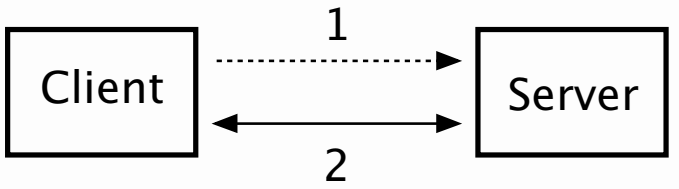
Networking

The whole point of a network is to allow two machines to connect and talk to each other. Once the two machines have found each other they can have two-way conversation.

Clients and Servers

One of the communicating machines-**server**-serves its services, while the second one-**client**-uses the services provided by the server. From the networking point of view the distinction is important only while the client is trying to connect to the server:

- the job of the server is to listen for a connection,
- the job of the client is to try to make a connection to a server.



Note: Once they have connected, it becomes a two-way communication process and it does not matter anymore that one happened to take the role of server and the other happened to take the role of the client.

Identifying a Machine

Machines are uniquely identified within the Internet by IP address, which can exist in two forms. Familiar associating machines with human readable form, for instance `java.cs.vsb.cz`. Alternatively, there is the **dotted quad** form, which is four numbers separated by dots, such as `158.196.149.94`.

> **Note:** In both cases, the IP address is represented internally as a 32-bit number (so each of the quad numbers cannot exceed 255).

The `java.net.InetAddress` class enables, among others, translation between the domain name and address, and vice versa.

```
String nameOrAddress = "...";
InetAddress address = InetAddress.getByName(nameOrAddress);
System.out.println("address: " + address.getHostAddress());
System.out.println("name: " + address.getHostName());
```

Identifying an Application

An IP address is not enough to identify a unique server, since many servers can exist on one machine. Each IP machine also contains **ports**, and when a client or a server is set up, a port must be chosen.

> **Note:** The port is not a physical location in a machine, but a software abstraction. Typically, each service is associated with a unique port number on a given server machine.

Uniform Resource Locator

A URL identifies resources via a representation of their primary access mechanism, i.e., their network **location**. The URL consists of several components:

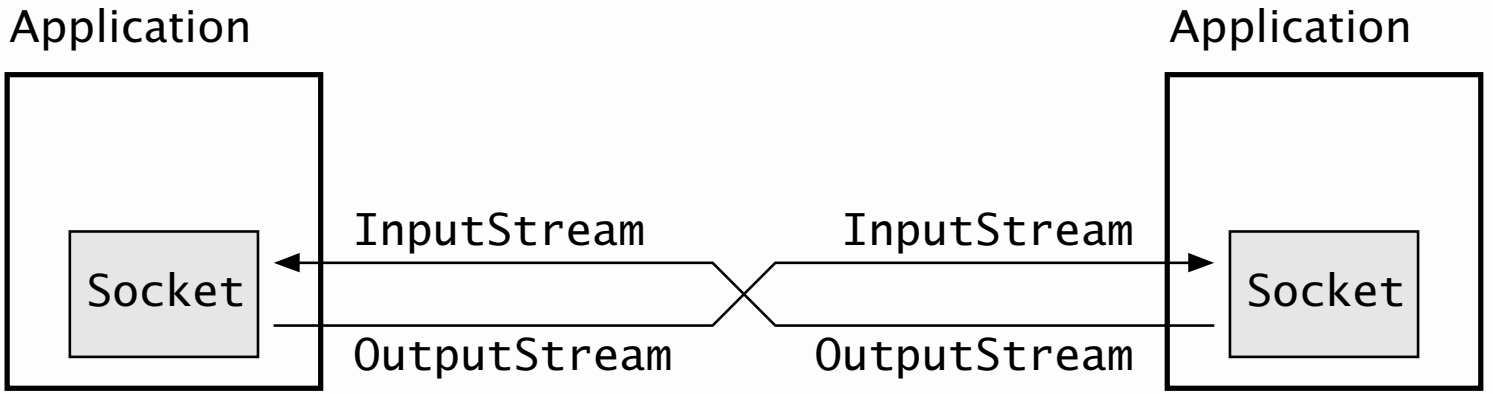
`protocol://hostname:port/path?query`

- `protocol` to be used to fetch the resource,
- `hostname` of the machine on which the resource lives,
- `path` to the file on the machine,
- `port` number to which to connect (typically optional),
- `query` is a reference to a named anchor within a resource.

```
try {
   URL url = new URL("http://java.cs.vsb.cz/docs/slides.pdf");
   InputStream slides = url.openStream();
   ...
}
catch (MalformedURLException e) {
   ...
}
```

Socket

The socket is the software abstraction used to represent the "terminals" of a connection between two machines. For a given connection, there is a socket on each machine.



Server

A java.net.ServerSocket instance waits for requests to come in over the network and returns a result to the requester.

```
int port = 1234;

ServerSocket server = new ServerSocket(port);

while (true) {
   Socket client = server.accept();
   Reader reader =
      new InputStreamReader(client.getInputStream());
   Writer writer =
      new OutputStreamWriter(client.getOutputStream());

   char c;
   while ((c = (char)reader.read()) != -1) {
      writer.write(Character.isLowerCase(c) ?
         Character.toUpperCase(c) : Character.toLowerCase(c));
      writer.flush();
   }
}
```

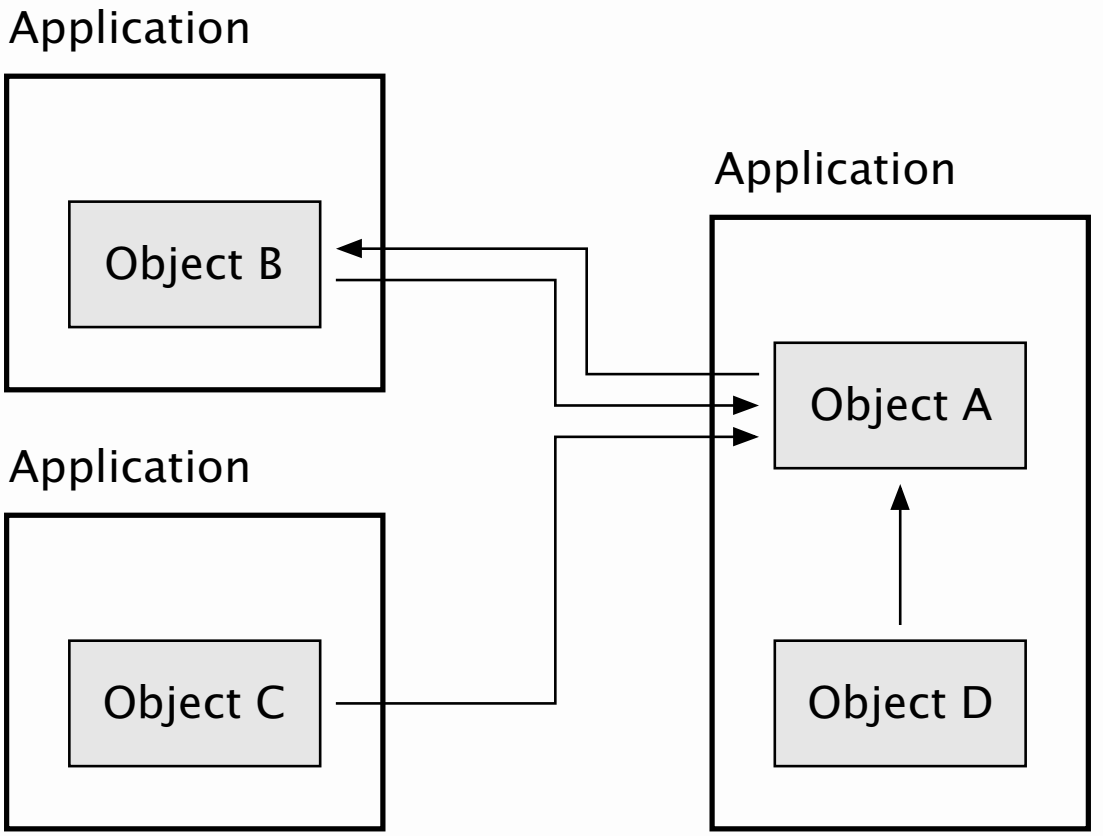
Client

A java.net.Socket is an endpoint for communication between two machines. Its instances are initialized with information needed to identify a remote application.

```
String hostname = "localhost";
int port = 1234;

String message = "aBcDeFgH";

Socket client = new Socket(hostname, port);
Reader reader =
   new InputStreamReader(client.getInputStream());
Writer writer =
   new OutputStreamWriter(client.getOutputStream());
writer.write(message);
writer.flush();

int length = 0, c;
while ((c = reader.read()) != -1 &&
   ++length < message.length()) {
   System.out.print((char)c);
}
```

Remote Method Invocation

The RMI provides the mechanism by which a server and a client communicate and pass information back and forth. Such an application is sometimes referred to as a **distributed object application**.



Remote Interface

When a remote object is created, the underlying implementation is masked by **remote interface**. The interface must have the following properties:

- the interface must be public,
- the interface must extend the interface `java.rmi.Remote`,
- every method in the interface must declare that it can throw `java.rmi.RemoteException`.

```
public interface Engine extends Remote {
   Object executeTask(Task task) throws RemoteException;
}
public interface Task extends Serializable {
   Object execute();
}
```

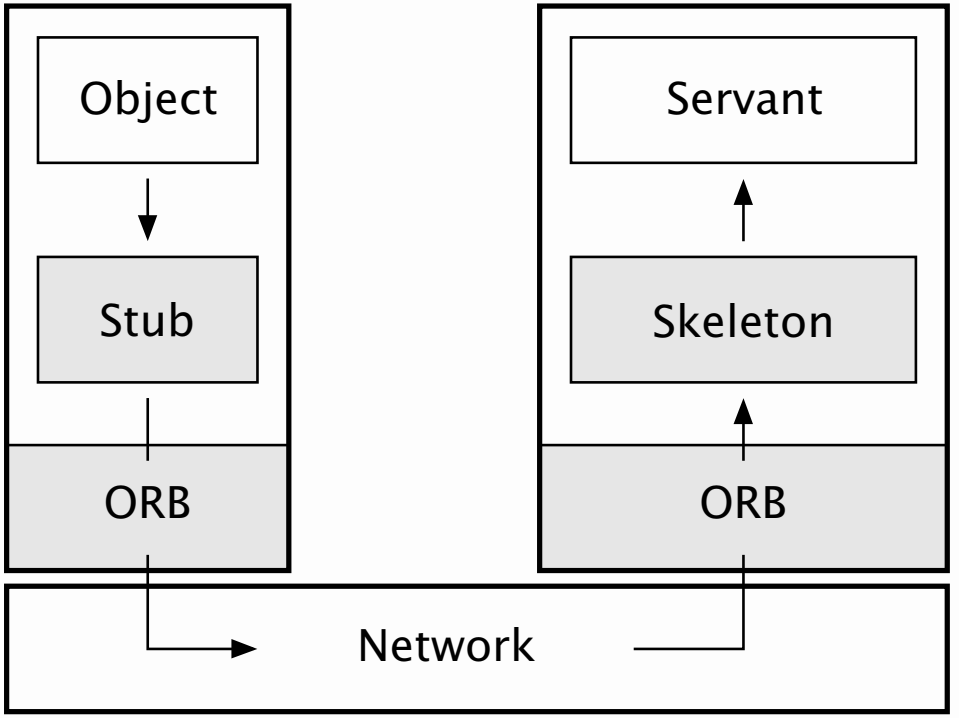
Implementing a Remote Object

A remote object must implement a remote interface and inherit from a `java.rmi.server.RemoteObject`. The `RemoteObject` class implements the `Object` behavior for remote objects.

```
public class EngineImpl
      extends UnicastRemoteObject implements Engine {
   public EngineImpl() throws RemoteException {
      super();
   }

   public Object executeTask(Task task) throws RemoteException
      return task.execute();
   }
}
public class TaskImpl implements Task {
   public Object execute() {
      Double result = new Double(45.6);

      System.out.println("calculating result...");

      return result;
   }
}
```

Object Request Broker

The ORB takes care of all of the details involved in routing a request from client to object, and routing the response to its destination.



Stubs and Skeletons

Stubs and skeletons that provide the network connection operations and allow to pretend that the remote object is just another local object on a machine. They are automatically performing serialization and deserialization as they marshal all of the arguments across the network and return the result.

- **Skeleton**
  Skeleton for a remote object is server-side entity that contains a method which dispatches calls to the actual remote object implementation.
- **Stub**
  Stub is a proxy for a remote object which is responsible for forwarding method invocations on remote objects to the server where the actual remote object implementation resides. A client's reference to a remote object, therefore, is actually a reference to a local stub. The stub implements only the remote interfaces, not any local interfaces that the remote object also implements.

Publishing Remote Object

Before a caller can invoke a method on a remote object, that caller must first obtain a reference to the remote object. The reference can be obtained from **RMI registry**.

```
> rmiregistry
```

- **Security Manager**
  All programs using RMI must install a security manager. The security manager ensures that the operations performed by downloaded code go through a set of security checks.

```
if (System.getSecurityManager() == null)
   System.setSecurityManager(new RMISecurityManager());
```

- **Binding the Remote Object**
  The `java.rmi.Naming` interface is used as a front-end API for binding, or registering, and looking up remote objects in the registry.

```
Engine engine = new EngineImpl();
Naming.rebind("engine", engine);
System.out.println("engine bound");
```

Passing Objects in RMI

Any entity of any type can be passed to or from a remote method as long as the entity is an instance of a type that is a **primitive** data type, a **remote** object, or a **serializable** object, which means that it implements the interface `java.io.Serializable`. The rules governing how arguments and return values are passed are as follows:

- Remote objects are essentially passed **by reference**. Any changes made to the state of the object by remote method calls are reflected in the original remote object,
- Local objects are passed **by copy**, using object serialization. Any changes to this object's state at the receiver are reflected only in the receiver's copy, not in the original instance.