Network Computing Lecture 7

Professor Louise E. Moser

Spring 2012

Java Remote Method Invocation

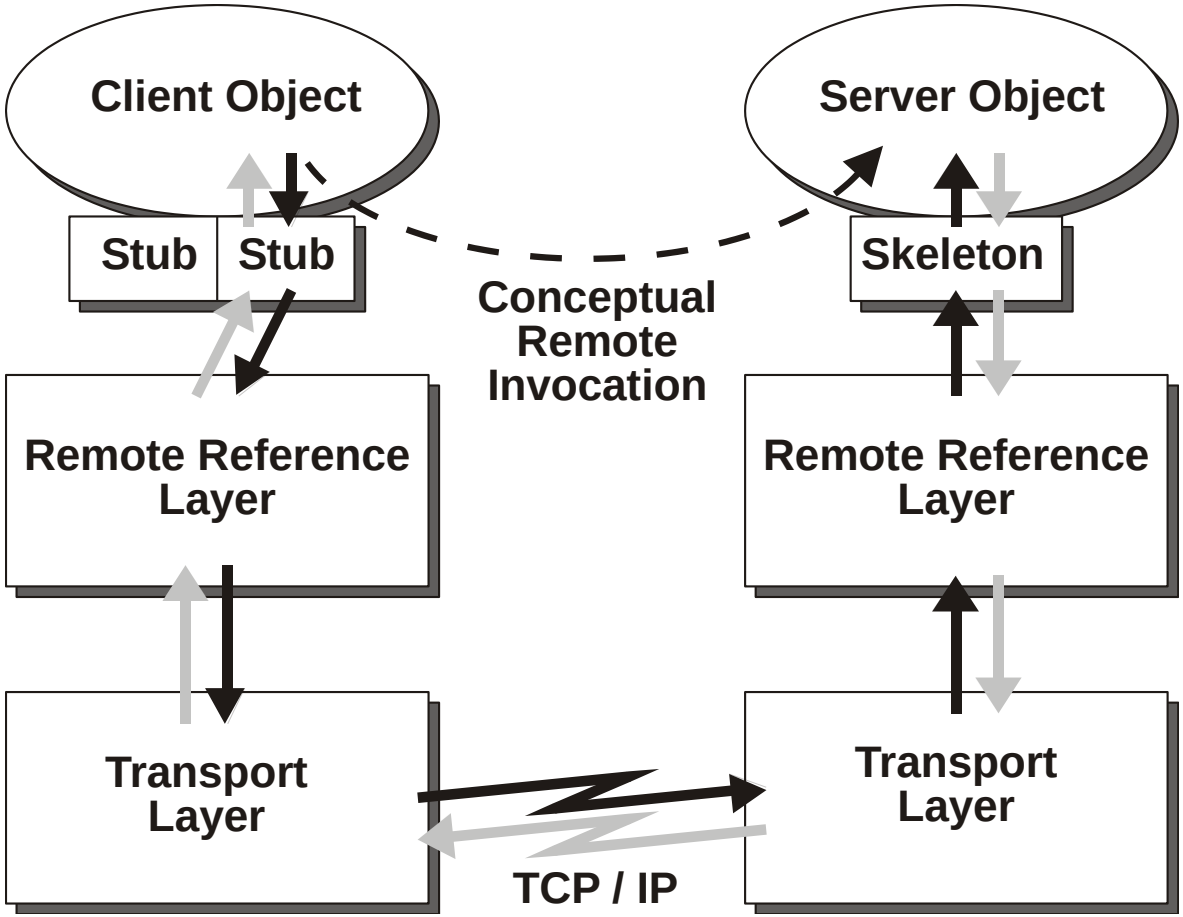
- Java Remote Method Invocation (RMI) allows objects to invoke methods of objects located on other computers, as if they were local
- The method invocation is packaged into a TCP/IP message and sent to the other computer, where the method of the remote object is invoked
- The result comes back in another TCP/IP message
- Java RMI is important because
  - Method invocation is easier to program than message passing
  - Method invocation is uniform for both local and remote objects
  - The Java program is determined by its logic, rather than by the allocation of objects to processors

A First Look at How Java RMI Works

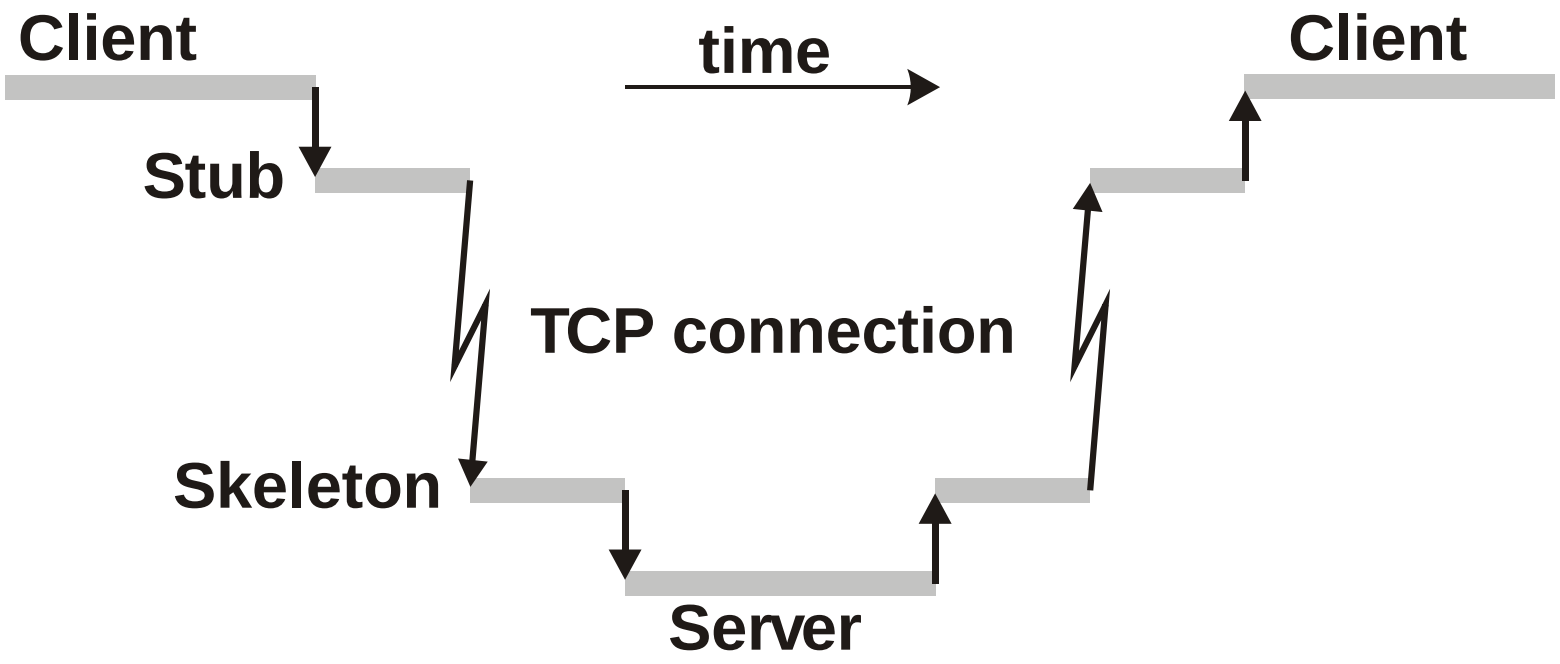
- The client object invokes a method of a remote object, as if it were local, but there is no such local method for it to invoke
- Instead, there is a **stub** that the client object can invoke
  - The stub has the same interface as the remote object
- When the stub is invoked, it passes the call (method name and parameters) to the **Remote Reference Layer (RRL)**
- The RRL packages the call into a message, and transmits the message to the remote computer by TCP/IP
- At the remote computer, the RRL unpacks the message and passes the call to a **skeleton** that invokes the method of the server
- The results go back to the client in the reverse order, i.e.,

  server to **skeleton** to **RRL** to **TCP/IP** to **RRL** to **stub** to **client**

A Client Invoking a Remote Method Using RMI



Another Way of Looking at Invocation of a Remote Method Using RMI



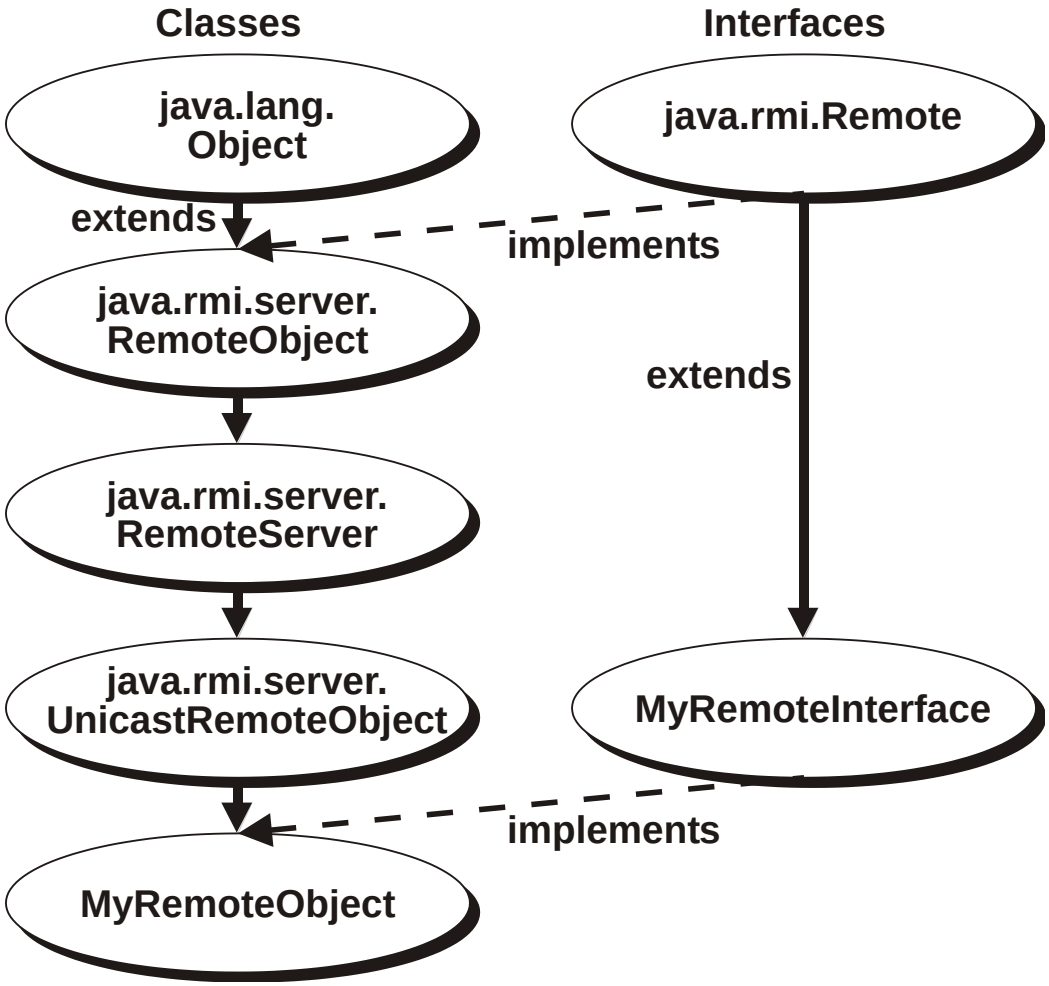
How Does Java RMI Really Work?

- There are lots of questions that must be answered before we can understand how Java RMI really works
  - How do we make a server object invocable remotely?
  - How does a client object find a remote object?
  - How does a client object get a stub?
  - How does RMI pass parameters and results?
  - How does RMI handle exceptions?

How Do We Make a Server Object Invocable Remotely?

- The server object must extend *java.rmi.server.UnicastRemoteObject*
- The methods that are invoked remotely must have their signatures defined in an interface that extends *java.rmi.Remote*
- The server object must be registered with
  - The Java RMI Registry, or
  - A private registry maintained by the application
- These registries (typically) run on the same machine as the server object

The Java RMI Class/Interface Hierarchy



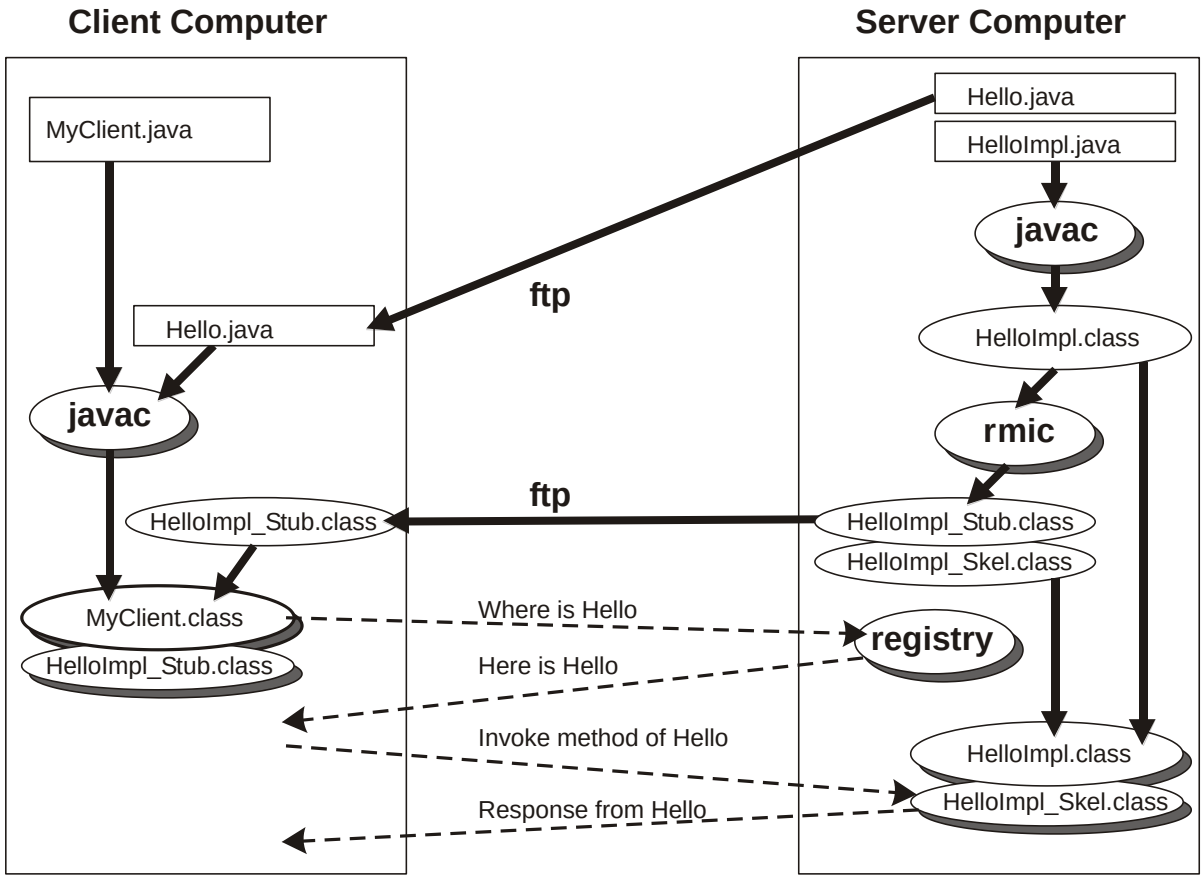
How Does a Client Object Find a Remote Object?

- There are two ways to do this:
  - The client can invoke the *lookup()* method of the Registry on the server machine, passing it a URL as a parameter, indicating the name of the machine and the server object, as in *rmi://foo.bar.edu/TimeServer*
    - The Registry returns a reference to a remote object (actually, a socket connection to the RRL, which connects to the skeleton)
    - This is what is done for the initial contact
  - The client invokes a method of a remote object, which returns a reference to another remote object
    - This is how most other references to remote objects are obtained

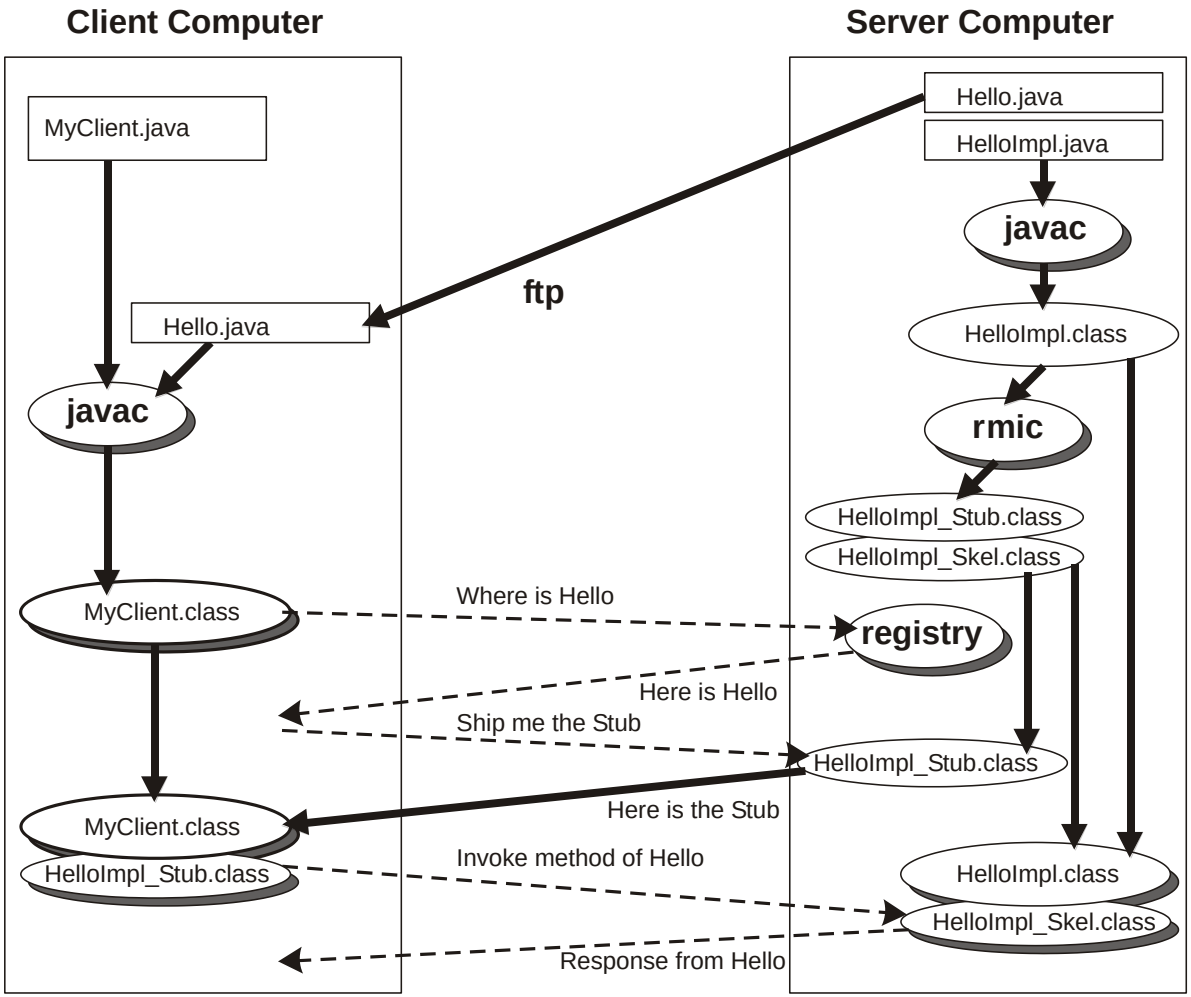
How Does a Client Get a Stub for the Server?

- The RMI compiler, *rmic*, processes the server source code and generates a set of *.class* files for the stub and the skeleton
- If the server source code is *XXX.java*
  - The stub is *XXX_Stub*, which is stored in the *XXX_Stub.class* file
  - The skeleton is *XXX_Skel*, which is stored in the *XXX_Skel.class* file
- The programmer of the client object needs the interface for the server object, to know how to invoke the server methods
- The user on the client-side needs the interface for the server object and the *XXX_Stub.class* file, to run the client
  - The *XXX_Stub.class* file can brought to the client-side before the client is compiled and run
  - Alternatively, the client can be compiled using only the interface for the server object, and the *XXX_Stub.class* file can be retrieved from the server machine at run time

How Does a Client Get a Stub for the Server?



How Does a Client Get a Stub for the Server?



How Does RMI Pass Parameters and Results?

- RMI must package the parameters into a message
  - This is easy for simple types such as *int*
- *Strin*gs and *vectors* are packaged as a length, followed by that many elements
- Some objects contain other objects
  - Those must be sent across the network too
- Everything in *java.lang*, *java.util*, and *java.swing* can be sent across the network
- What cannot be sent across the network?
  - *Threads*, *InputStreams*, *OutputStreams*, and *java.sun.**

How Does RMI Handle Exceptions?

- It is possible that the network connection is down, the remote object is gone, or the remote object generates an exception
- All remote methods should be defined with *throws java.rmi.RemoteException*, as in

  public String remoteMethod() throws java.rmi.RemoteException;
- All invocations should be placed within a *try/catch* block, as in

```
try {
  remoteObject.remoteMethod();
}
catch (java.rmi.RemoteException e) {
  System.err.println(e);
}
```

- For more precise handling of remote exceptions, there are 16 subclasses that extend *java.rmi.RemoteException*

The TimeServer RMI Example

- This simple remote object provides a method *getTime()*, which returns the current time and date as a string
- First we define the interface

```
package rmiexample.timeserver;
public interface TimeServer extends java.rmi.Remote {
    public String getTime() throws java.rmi.RemoteException;
}
```

- The interface must be public so that remote clients can get it, and must extend *java.rmi.Remote* or a subclass of *java.rmi.Remote*
- The interface defines one method *getTime()* that must also be public
  - *getTime()* requires no parameters and returns a string
  - *getTime()* is defined with *throws java.rmi.RemoteException*
- The interface is saved in the file *rmiexample.timeserver.TimeServer.java*

The Implementation of TimeServer

```
package rmiexample.timeserver;
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

public class TimeServerImpl
                extends UnicastRemoteObject
                        implements TimeServer {
  ...
```

- The implementation of the *TimeServer* interface is named *TimeServerImpl*
- *TimeServerImpl* extends the *UnicastRemoteObject* class and implements the *TimeServer* interface
- *TimeServerImpl* is saved in the file *rmiexample.timeserver.TimeServerImpl.java*

Initializing TimeServerImpl

```
public TimeServerImpl() throws RemoteException
  {                    // Constructor
  // Initialize the TimeServerImpl object
 }
```

- In this example, nothing needs to be done to initialize the new object
- Most objects do need some initialization

The getTime() Remote Method

```
public String getTime() {
  //  Send the date and time
  return new java.util.Date().toString();
}
```

- This is the easy part
- It looks almost exactly like a local method, which is good because, in a complex program, this is what would be complex
- The *new* and *toString()* ensure that you have a new *String* object, rather than a pointer to something owned by *java.util.Date*

The main() Method of TimeServerImpl

```
public static void main(String args[]) {
    System.setSecurityManager(new RMISecurityManager());
    try {
        TimeServerImpl tsi = new TimeServerImpl();
        Naming.rebind("TimeServer", tsi);
    }
    catch ...
```

- The *main()* method creates *tsi*, a new *TimeServerImpl*
- It then invokes *rebind()* to register *tsi* in the Registry (*Naming* service) with the name "*TimeServer*"
  - The *main()* method has an array of strings, *args[]*, as a parameter
  - There are a bunch of exceptions that might need to be caught
  - We will cover the stuff about the *RMISecurityManager* next time

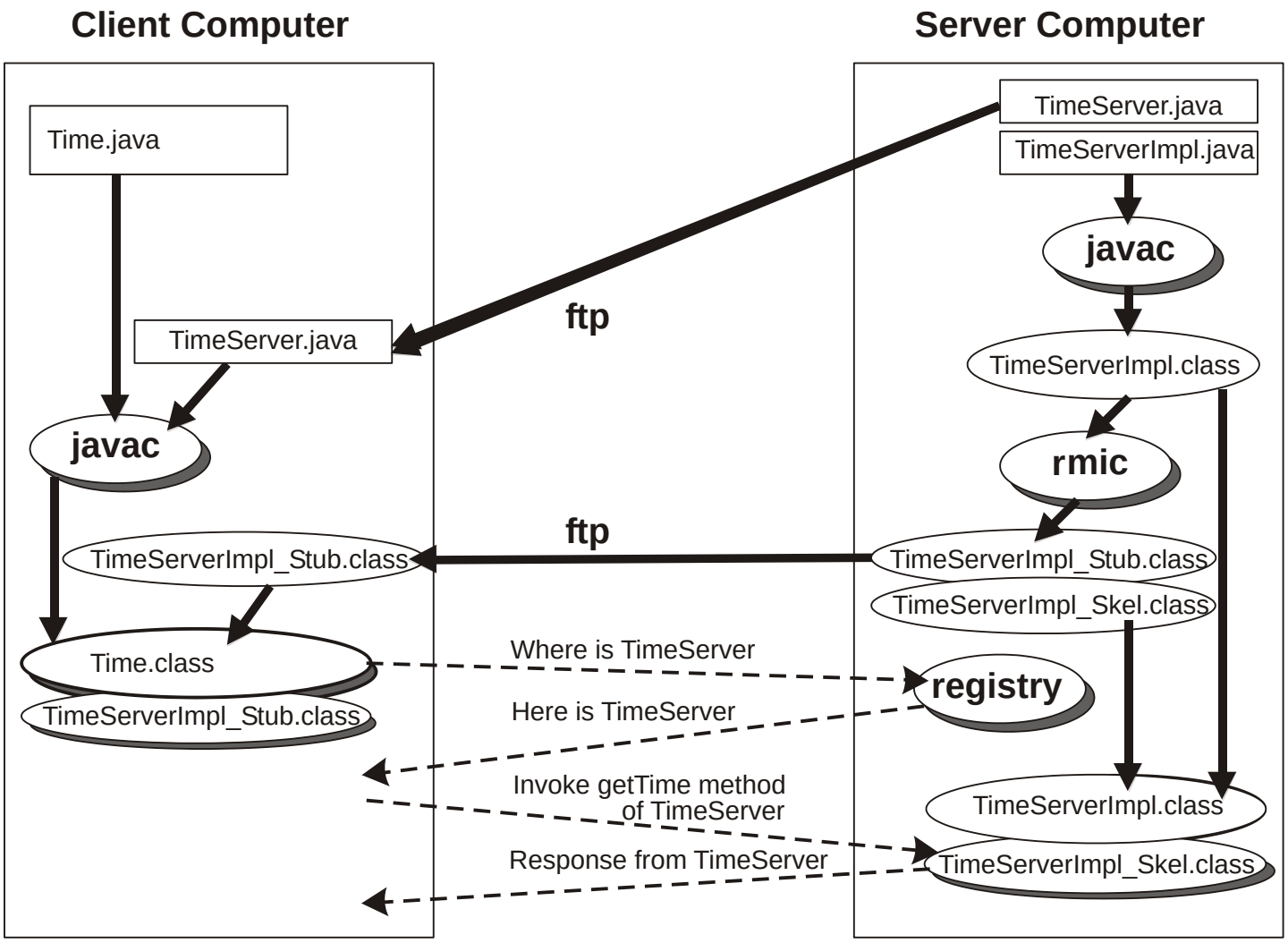
The Time Client That Invokes the TimeServer

```
package rmiexample.timeserver;
import java.rmi.*;
public class Time {
  public static void main(String arg[]) {
    String time = null;
    try {
      TimeServer ts = (TimeServer)Naming.lookup(
                                    "rmi://foo.bar.edu/TimeServer");
      time = ts.getTime();
    }
    catch {
    ...
    }
    if (time!=null)
      System.out.println("The Time is " + time);
} }
```

The Time Client That Invokes the TimeServer

- The Time Client declares a remote object, *ts*, through which it has access to the Time Server
- It invokes the *lookup()* method of the Registry to obtain a reference to the Time Server (actually, a socket connection to the RRL which connects to the skeleton)
  - It uses the *URL rmi://foo.bar.edu/TimeServer* as a parameter of the *lookup()* method
  - The *lookup()* method returns an object, which is cast into a *TimeServer* object
  - The *TimeServer* object is assigned to *ts*
- It invokes the *getTime()* method of the *TimeServer* object *ts*, which returns the time to the Time client
- There are a bunch of exceptions that you might need to handle

The Time Client and the TimeServer



The Time Client and the TimeServer