
CIS578 – Advanced Operating Systems

Project

A Middleware for Remote Method Invocation

DUE DATE: Dec. 1, 2004

Background

This project consists of designing and implementing the main components of a middleware that enables remote method invocation (RMI). Figure 1 illustrates the main components involved in an RMI framework. The numbered arrows indicate the order of events that take place when a remote object is invoked by another object (here indicated by *local obj*).

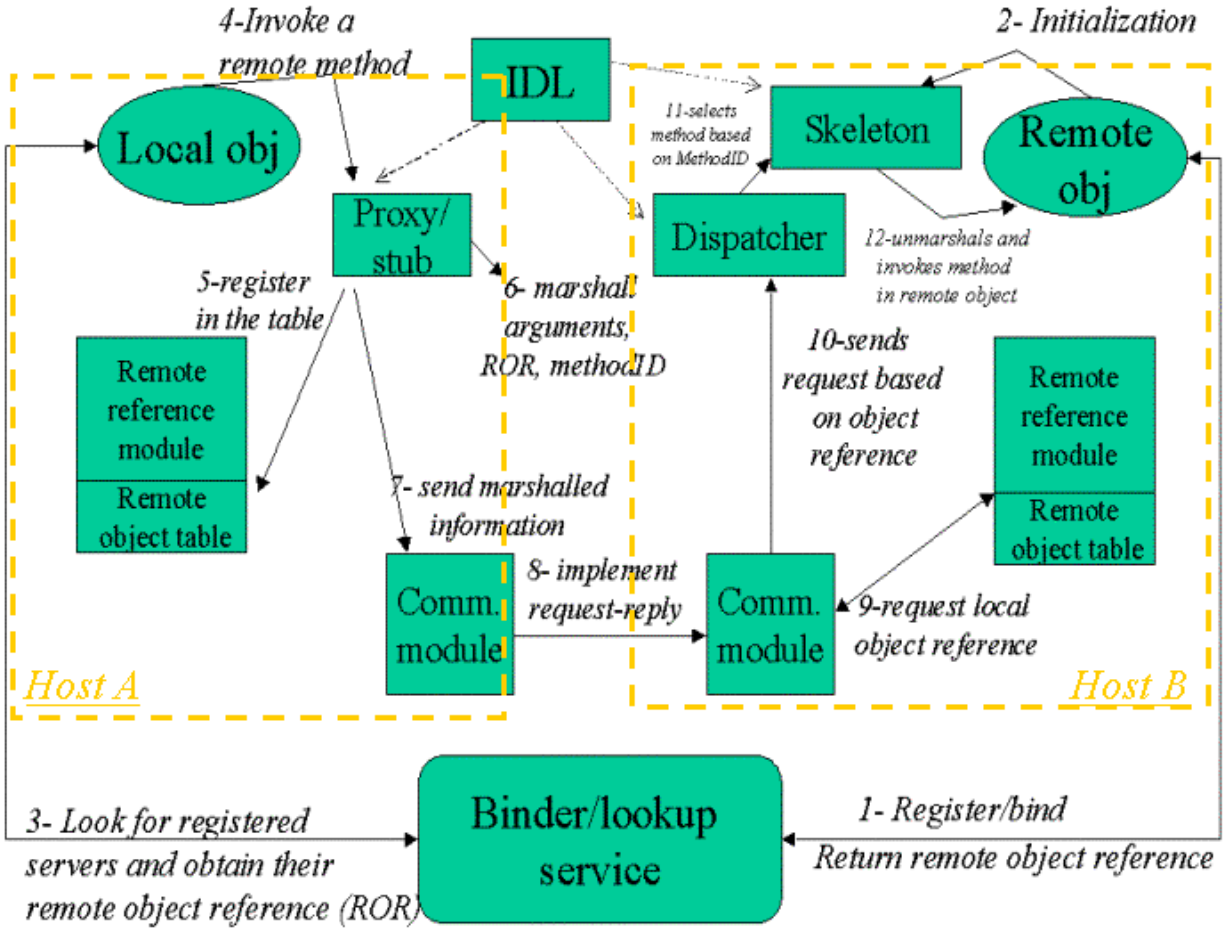

Figure 1 - Main components of an RMI framework

The first step, however, is the generation of the proxy, dispatcher and skeleton for each interface defined in an IDL file. The compilation of the IDL file generates the code for those three components as well as a header file

containing the name of all the methods that can be remotely accessed by other objects. The detailed description and design of each of the above components can be found in the course textbook (Coulouris – chapter 5). You can also get more details and specific implementation details in OMG's CORBA, Java RMI and Sun RPC specifications.

Your Implementation

You are to implement the following components from Figure 1:

- Communication modules
- Remote reference module
- Proxy
- Dispatcher
- Skeleton
- Binder/lookup service

You can assume that an IDL program has already been compiled, generating the proxy, dispatcher and skeleton that you created. In other words, you do not have to write a parser to read any given IDL file and create the corresponding proxy, dispatcher and skeleton files. Your middleware, therefore, will only be a "testbed" for invoking a given set of methods on a remote object. For example, you can pre-define a set of methods that the objects can invoke from a remote object: get_bank_balance(in account), deposit_account(in account, in amount, out status), withdraw(in account, in amount, out status). You can then create a proxy, skeleton and dispatcher for the interface consisting of these methods.

You will have different implementation options when designing the modules. For example, you can choose *at-least-once* and *at-most-once* semantics for the communication module. The option is yours, as long as you justify/explain your implementation. Similarly, you have various options when implementing the binder/lookup service: a global table, distributed storage, etc.. Again, it is your option, just make sure to explain and document your design and implementation.

Underlying Communication Layer

As we reviewed in class, the middleware layer runs on top of the communication layer. The implication of this is that your middleware will make use of the services of the communication layer to execute its functions. Thus, after the request/reply is marshalled and ready to be transmitted to the remote object, the communication module of the middleware will send the request/reply to the remote object via the underlying network.

We will assume that the underlying communication layer offers TCP/UDP services. The data will therefore be encapsulated in a TCP/UDP segment and transmitted via sockets created at each end of the communication channel.

Socket programming (or TCP programming) can be done both in C and Java (references and examples are included in this project description).

Assumptions/Requirements

1. You can assume that an IDL file has already been defined and compiled. Your stub/proxy, skeleton/dispatcher should work for the following interfaces: *Student* and *Student-List*. The *Student-List* is the interface to the server. The server maintains a list of students that can be manipulated through the following functions:

- addstudent (in Student, out OK)
- delstudent (in number, out OK)
- listall (out Vector)
- find(in number, out Student)
- setGrade(in number, in Grade)
- getGrade(in number, out Grade)

Where:

- *number* is string of characters representing student numbers
- *OK* is a return value indicating if the operation was successful
- *Vector* is the Vector holding all the Students in Student-List
- *Student* is an instance of the Student class, which has the following fields:
  - *Name* – a string holding the student's last name
  - *Age* – an int holding the student's age
  - *Address* – a string holding student's address
  - *Program* – a string indicating the program the student is enrolled in.
  - *Grade* – an instance of the Grade class, representing students' grade.

- *Grade* is an instance of the Grade class, which has the following fields:
  - *CourseI* – grade for Course I
  - *CourseII* – grade for Course II
  - *CourseIII* – grade for Course III
  - *GPA* – some total value

2. Level of transparency: the user should be able to start the communication between two objects by starting two separate programs: *client* (or object 1) and *server* (or object 2) in any given location. The user does not have to know where the server is running. Client and server must be able to communicate while running in different platforms, i.e., a client can run on a PC while the server is running under UNIX, and vice-versa.

3. The client program is single-threaded, whereas the server is multithreaded when dealing with more than one client. The server should therefore create a new thread for each new client that makes a request.
   Only one thread waits on the server's TCP/UDP socket for receiving messages. When a message arrives, the main server thread has to notify the thread corresponding to the client that sent the message.

4. Since multiple clients can access the server simultaneously, you must implement some form of mutual exclusion to avoid inconsistent results.

5. Your middleware should marshall all method parameters and return values. See Coulouris Section 4.3. Marshalling should be done such that *Grade* and *Student* are passed by value.

6. Your middleware should be fault-tolerant, through active or passive replication.

7. When requested, your middleware should provide a consistent global state of the system, i.e., a consistent ordering of events (e.g., ordering of requests received by the server), including the status of communication channels.

8. All the above functionality should be "testable". For example, in he case of marshalling, the data being transmitted should be displayed, before and after serialization. In the case of fault-tolerance, a message should be displayed whenever data is being replicated. There is no need for a GUI, but ease of use is required for all functions.

Deliverables

1 – Source code for all the implemented components in a zip file uploaded to the VLT site. The zip file should also include:

A- Manual describing how to compile and run your middleware.

B- Documentation:

a. Description of the main components and their functions
b. The interface between all components
c. Design issues justifying the choices made (e.g., choice of an invocation semantics or the mutual exclusion method chosen)
d. A user interface section describing how to use your middleware (I will use this to test your project)
e. References: list of additional sources used in the project
f. Task list: how was the work distributed in your group? Who did what?

References on Socket Programming

You can implement sockets either in C or in Java. For instructions on programming sockets in 'C', check www.ecst.csuchico.edu/~beej/guide/net/

The following are sample codes for a client and server using Java sockets (Coulouris Book, 4.2.2).

The Client

```
import java.io.*;
import java.net.*;
class TCPClient{
      public static void main(String argv[]) throws Exception
      {
            String sentence;
            String modifiedSentence;
            BufferedReader inFromUser = new BufferedReader(
                                          new InputStreamReader(System.in));
            Socket clientSocket = new Socket("hostname",6789);
            DataOutputStream ouToServer =
                      new DataOutputStream(clientSocket.getOutputStream()));
            BufferedReader inFromServer = new BufferedReader(
                        new InputStreamReader(clientSocket.getInputStram()));
            sentence = inFromUser.readLine();
            outToServer.writeBytes(sentence + '\n');
            modifiedSentence = inFromServer.readLine();
            System.out.println("FROM SERVER " + modifiedSentence);
```

```
            ClientSocket.close();
      }
}
```

The Server

```
import java.io.*;
import java.net.*;
class TCPServer{
      public static void main(String argv[]) throws Exception
      {
            String clientSentence;
            String capitalizedSentence;
            ServerSocket welcomeSocket = new Server Socket(6789);
            while (true) {
                  Socket connectionSocket = welcomeSocket.accept();
                  BufferedReader inFromClient =
                        new BufferedReader(new
                            InputStreamReader(connectionSocket.getInputStream()));
                  DataOutputStream outToClient =
                        new DataOutputStream(connectionSocket.getOutputStream());
                  clientSentence = inFromClient.readLine();
                  capitalizedSentence = clientSentence.toUpperCase() + '\n';
                  outToClient.writeBytes(capitalizedSentence);
            }
      }
}
```