Remote Procedure Calls and Java Based Interprocess Communication

Sub Ramakrishnan, **Mohammad B. Dadfar**

Department of Computer Science
Bowling Green State University
Bowling Green, Ohio 43403
Phone: (419)372-2337 Fax: (419)372-8061
Email: datacomm@cs.bgsu.edu

Abstract

The growth and expansion of the internet has created opportunities and a need for exploring internet technologies in a classroom setting. Techniques such as remote procedure calls which were established more than a decade ago are being revisited as client-server networks become popular. Developers are also scrambling to build Java based applications that can be deployed on any desktop.

In the last offering of our data communications course, we did a pilot study and made some changes to our traditional offering of this course. The focus was more hands-on experience, exposure to modern technology, and less on theory. The classroom setting was informal and projects were group-oriented. In this paper we describe two projects that were assigned during this offering. The first project uses Java to build a client-server application and attempts to compare the tradeoffs between Java and C++. The second project is an extension of the local programming paradigm. The students build a network file service protocol using remote procedure calls.

1. Introduction

It is well-known that data communications and computer networking has become one of the most important areas in computer science. Computer Science departments offer one or more courses in this area in their undergraduate/graduate programs. We at Bowling Green State University have offered a Data Communications and Networks course in our undergraduate program since early 1980s. This course has become increasingly popular among students and as a result we have increased the number of sections offered. Moreover, we have introduced an additional course in operating systems and networks at the junior level which is now a required course for all students majoring in computer science.

To make the Data Communications course more useful, the practical part of the course is carefully designed so that students could incorporate the theoretical concepts with current issues in real-world networking. Since the area is changing rapidly the choice of projects has become a major task for instructors.

The growth and expansion of the internet has created opportunities to design practical projects in this area. Computer Science departments around the country react to such technology trends and attempt to integrate these concepts in their curriculum. Instructors are motivated to expose internet-related technologies in a classroom setting. Remote Procedure Calls (RPC), which were established more than a decade ago are being revisited as client-server networks become popular. Java based networked applications that can be deployed on any desktop have recently gained the attention of developers and are also candidates for course projects.
Traditionally, we have covered a considerable amount of theoretical concepts, and some practical projects that help enhance some of these concepts, in our offering of the data communications course. In our last offering of this course, we did a pilot study and made some changes. The focus was more hands-on experience and less on theory. The classroom setting was informal and projects were group-oriented. In this paper we describe two projects that were assigned during this offering.

We discuss the motivation in assigning these projects, specific details of the projects, the problem solving phase and the impact of teamwork on the final product. We also describe certain extensions to these projects that might be of use to other institutions.

In Section 2, we give an overview of the topics in our data communications course and preliminary details of the two course projects. These projects are discussed in Section 3 and Section 4 respectively. Concluding remarks appear in Section 5.

2. Data Communications Course

Our Data Communications and Networks course (CS 429) is a 3 credit hour course and runs for 15 weeks for a total of 45 hours of classroom instruction. (The summer offering is 8 weeks long and the class meets about six hours per week.) Both senior undergraduates and some graduate students enroll in this course. The students have already had at least two courses on programming and data structures, and have been introduced to UNIX systems.

The topics covered in the recent offering of the course include the following: basics of networking, benefits of resource sharing, network topologies, comparison of packet and circuit switching, multiplexing, transmission techniques, bandwidth limitation, and transmission impairments. These topics take up about 30% of the course. Approximately 50% of the class time is spent on the following topics: OSI model, physical connectivity, ethernet and token ring standards, TCP/IP protocol suite and related RFCs, domain name system issues, client-server architectures and remote procedure calls paradigm. About 10% of the class time is spent on ISDN and broadband ATM networks. The remaining 10% is taken up by three exams giving during the term and a final exam. A good discussion of the topics covered in the course can be found in standard textbooks [1, 5, 8, 10, 12].

Usually, we assign about four programming projects and three to five homework assignments in the course. The projects are designed to provide students with hands-on experience in data communications and network applications. Many class projects have been proposed in the literature [2, 3, 6, 7, 9]. This paper describes two projects, offered during the Summer 1997 term, that relate to the middle 50% of the classroom instruction.

The first project is to respond to students' interests in Java and to compare the tradeoffs in building client-server applications in C++ vs. Java. Java networking mechanisms are direct and

easy to understand and provide a level of abstraction that is simple for first time network programmers. Unlike C++, Java error handling is separate from the main code and provides a number of packaged components to build simple GUI interface without much programming. A client-server application was used as a model to demonstrate some of these features. The students built a client application in Java that communicated with the *finger* daemon on a remote machine. The machine/port number were all parameterized making the client a general purpose client. The students were extremely enthusiastic about this project and some students even suggested useful extensions to the project.

The motivation behind the second project is given briefly. Local procedure calls paradigm is well known in the computer science literature [4]. RPC is an extension of this concept and is applicable for client-server architectures. Sun proposed this concept and built Network File Service (NFS) using the RPC paradigm [11]. RPC interface provides a higher level abstraction than socket based TCP/IP programming. It also bridges the gap between data formats across heterogeneous computer systems. RPC integrates well within the course framework.

The second project is an extension of the local programming paradigm. The students built an application to provide remote file service with functionality similar to that of NFS. It included primitives for remote file manipulation: open, read, write, and close. Like NFS, the application is stateless. This project helped them appreciate the need for a network representation and the simplicity and power of stateless servers. Further, they realized that the classroom instruction provides practical concepts useful in building real world applications comparable to popular services like NFS.

3. A Client-Server Project in Java

Classroom instruction provides the necessary background in client-server networking. Topics in this area include: TCP/IP and IP addressing, host and network part, concept of port number, preliminaries for setting up a stream oriented connection between a client and a server, and the difference between clients and servers.

In this project students write a client-server communication application in Java. The client communicates with a well known server such as the *finger* daemon. The client application provides a graphical user interface for entering the information the client needs in order to finger the server. This includes: the server machine name or IP address, server port number, and users to be fingered.

Students are exposed to Java for the first time in this course. Java instruction takes under two class periods for our students who are already familiar with C++. We first discuss a simple *hello world* program, then a program to compute the average of the numbers entered at the command line (these programs are not shown here). Syntax and semantics of useful statements such as `System.out.println`, extracting command line arguments, computational statements and flow of control are discussed briefly. Now the students write a few other simple programs on their own to become familiar with the Sun JDK and debugging tools [11].

In the second class period we discuss how networking is done in Java: the format of `Socket` primitive, how to chain Socket's input stream to `DataInputStream` and to read from a `DataInputStream`.

Briefly, the client does the following: i) connects to the server and creates an instance of `Socket` by passing, the server machine name and the port number the server is available on, to the constructor of `Socket` and ii) creates an instance of `DataInputStream` by chaining the `getInputStream` method of `Socket`. At this point, the client can read a line from server by invoking the method `readLine()` of the instance that was just created. We also discuss the methods of the class `ServerSocket` and how to chain the Socket's output stream to `PrintStream`.

Briefly, the server does the following: i) creates an instance of `ServerSocket` by passing the port number to the constructor of `ServerSocket`, ii) accepts connection from client by invoking the `accept` method of `ServerSocket`, iii) creates an instance of `PrintStream` by chaining the `getOutputStream` method of `Socket`. At this point, the server can use `println` method of `PrintStream` to send data to the client.

Note that the client can also write to the server or the server can read from the client by using the `DataInputStream` and `PrintStream` in both the client and server. It needs to be pointed out that the server modules are discussed in class but the students do not actually implement them.

For completeness, we discuss the Java exception mechanisms and the specific exceptions raised by these calls, and how to handle them. In fact, during the explanation of these primitives, the students readily figure out how the primitives should be sequenced to get the job done. A sample code for the finger client is given in Appendix A.

Couple of observations are in order. The client and server code of Java are much simpler than comparable C++ client-server code. The C++ client code is far too complex. Though the `socket` primitive in C++ is easy to understand, the `connect` primitive is not. It requires understanding of a suite of structures, network to host formats and carefully marshaling arguments into these structures. The C++ server code is also complicated. It involves a number of primitives including `socket`, `bind`, `listen`, `accept`. The `read`/`write` primitives are quite intuitive. The students do the programming in C++ as well and are in agreement with the fact that Java client-server mechanisms are straightforward and more intuitive. For brevity, we do not include the C++ client-server code. The only subtle part of Java is the chaining of input or output streams that facilitates simpler primitives for reading and writing (`readln` and `println`). But the students seem to get over this difficulty quickly.

4. RPC Client-Server

While RPC concept is easy to understand there are some cumbersome details the students have to master before building successful RPC applications. Specifically, the following issues need to be explained in class in order to get a handle on RPC: i) interface specification - format of data that will be exchanged between client and server, program number, and procedure names and parameter types of server routines that are callable by the client, ii) compilation mechanisms and client and server level stubs, iii) slight differences in the way one writes local vs. remote procedures, iv) how the server registers its presence with the operating system so that the client can subsequently invoke server procedures. Item (i) provides opportunities for comprehensive

discussion of this topic in the classroom. It also provides the necessary justification why RPC is a higher level abstraction with primitives that are easier to use as compared to socket level calls.

Once these details are resolved in class students may be given a simple but complete example of an RPC code that works. Appendix B is an RPC based sort program; the client passes an array to be sorted and the server returns sorted output. The interface specification file (B.1) provides a program number, version number, and a number for each procedure in this program. Further, it defines an integer array with 5 elements and defines the parameter type of the procedure `sortnum`. The client makes a TCP connection with the server program (`SORTPROG`) using the primitive `clnt_create` (see B.2). It then invokes the procedure `sortnum` whose input is the numbers to be sorted and the output is the sorted array (note that it is consistent with the interface specification file, B.1). The client then prints the numbers.

It is worth noting that the server program (B.3) is quite similar to a 'local' sort procedure. The only difference is that the procedure has a number that is associated with it. By convention, pointers are used for input and output. We usually go through a program like this in class to make sure students understand the complete details of RPC. Now we are ready to discuss the RPC project.

The students write an RPC based remote file server. The server responds to file open, read, write, and close calls from an RPC client. The server performs the requested operation at the server side and returns the result. Naming conventions for these primitives are kept as close as possible to local file access primitives. The following primitives are to be supported:

- `ropen`: Open a file on the server. Input arguments are: file name, mode (read/write). Semantics are similar to open call. Returns the file descriptor for the file that was opened.
- `rread`: Read off of a file on server. Input arguments are: file descriptor and number of bytes. Returns the string read.
- `rwrite`: Writes to the file. Input arguments are: file descriptor, buffer to write, and length of the buffer. Returns none.
- `rclose`: Close the file.

We spent some time in class going over the details of this assignment. We also gave some hints on the interface specification file. There is a great deal of interaction in class when students are asked to figure out the description of the interface specification file: primitives, parameters and data types. This topic also provides opportunities for a discussion of stateful vs. stateless servers. The assignment is a bit of challenge initially but most students like the challenge. Once they code one primitive completely other primitives are straightforward and easily accomplished. The assignment is particularly well suited for a group-project since there are several interacting but somewhat independent modules. The students spend about 15 hours to complete this assignment.

5. Concluding Remarks

In this paper we examined a recent offering of our Data Communications and Networks course and explored hands-on project options for this course. We discussed a Java based client-server project and a project based on RPC. The projects integrate well with classroom discussion of

client-server architectures. The projects are practical and is of moderate complexity and help students understand how networked applications actually work. These are assigned as group projects and students like the opportunity to interact with their peers and learn from each other.

Couple of observations are in order. Exposure to modern technology that integrates well with real world applications provides a big motivation for students and keeps their level of interest high. By involving students in the material being covered, a broad range of topics can nicely be covered in the course. Since students have different learning styles, some additional help outside of regular office hours was also given for those in need to get everyone on board. The overall reaction from students was quite favorable.

These projects can be easily expanded to meet other needs. For example the client-server project can focus on host to IP address translation, keeping the connection a bit longer and exchanging multiple messages (as in the case of an SMTP protocol) and robust exception handling. The RPC assignment may include mechanisms for dynamic program registry, for the server to call back the client when the work is done instead of the client being blocked while work is in progress at the server, and for the client to automatically discover the services offered at a specific machine.

Appendix A: Java Based *finger* Client

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

//  finger client.
//  Command line arguments: port # of finger server, server machine
//  User to be fingered is hard coded -- may be changed
//  Print message received from host -- the time on host
//  Read from server may be expanded to a loop

public class finger
{
  public static void main(String[] args)
  {
    Socket theSocket;
    String hostname;
    DataInputStream receiveDataStream;
    PrintStream sendDataStream;
    if (args.length == 0) {
        System.out.println("Usage: progName serverDayTimePort serverHost" +
            "  Purpose: Call server @serverDayTimePort & get time");
        System.exit(1);
    }

    if (args.length == 2) {
      hostname = args[1];
    }
    else {
      hostname = "localhost";
    }

    try {
      theSocket = new Socket(hostname, Integer.parseInt(args[0]));
//    Chain socket's input stream to DataInputStream
//    DataInputStream is better for reading ASCII text than getInputStream

      receiveDataStream = new DataInputStream(theSocket.getInputStream());

//     Chain socket's output stream to DataOutputStream
//     DataInputStream is better for writing ASCII text than getOutputStream

      sendDataStream = new PrintStream(theSocket.getOutputStream());
      sendDataStream.println("Hello");

//    Read a line off of server
      String theTime = receiveDataStream.readLine();
      System.out.println("It is " + theTime + " at " + hostname);
    }  // end try

    catch (UnknownHostException e) {
      System.err.println(e);
    }
    catch (IOException e) {
      System.err.println(e);
    }
  }  // end main
}    // end daytimeClient
```

Appendix B: RPC Based Sort Program

B.1: Interface Specification File

```
typedef int Array[5];
program SORTPROG
{
    version SORTVERS
    {
        Array  SORTNUM(Array) = 1;  // procedure name
    } = 1;                          // program version number
} = 0x20000099;                     // program number
```

B.2: Client Code

```
#include <stdio.h>
#include <rpc/rpc.h>
#include "sort.h"
//                  C L I E N T: Written in C
main(argc, argv)
int  argc;
char **argv;

/*   This sample program does the following:
     (1) Client gives an array (hard coded) to server.
     (2) Server sorts the array and passes it back to client.
     (3) Client displays the sorted array on the terminal.
*/

{
   CLIENT *cl;
   int i, *result;
   char  *server;
   Array array;
   array[0] = 4;
   array[1] = 5;
   array[2] = 9;
   array[3] = 6;
   array[4] = 2;

   server = argv[1];
   cl = clnt_create(server, SORTPROG, SORTVERS,"tcp");
   if(cl == NULL) {
      clnt_pcreateerror(server);
      exit(1);
   }

   result = sortnum_1(array, cl);
   if(result == NULL) {
      clnt_perror(cl, server);
      exit(1) ;
   }

   for (i = 0; i < 5; i++)
      printf("\n%d\n", *result++);
}
```

B.3: Server Code

```
#include <stdio.h>
#include <rpc/rpc.h>
```

```
#include "sort.h"
//                    S E R V E R: Written in C
int  *sortnum_1(numary)
int  *numary;
{
   int iCtr1, iCtr2, Temp;
   static Array result;

   for(iCtr1 = 0; iCtr1 < 5; iCtr1++)
      result[iCtr1] = *numary++ ;
   for(iCtr1 = 0; iCtr1 < 5; iCtr1++)
      for(iCtr2 = iCtr1 + 1; iCtr2 < 5; iCtr2++)
         if(result[iCtr1] < result[iCtr2])
         {
            Temp = result[iCtr2];
            result[iCtr2] = result[iCtr1];
            result[iCtr1] = Temp;
         }
   return(result);
}
```

References

[1] Comer, Douglas, "Computer Networks and Internet," Prentice-Hall, 1997.

[2] Dadfar, Mohammad B. , Francis, Jeffrey, and Ramakrishnan, Sub, "Application of Client/Server Paradigm and web Technologies in a Networking Course," ASEE 1997 Annual Conference, 2220-04.

[3] Dadfar, Mohammad B. and Evans, Stephen, "An Instructional Token Ring Model on the Macintosh Computer," ASEE Computers in Education Journal, Vol. IV, Number 1, January-March 1991, pp. 28-32.

[4] Deitel, H., and Deitel, P., "C++: How to Program," Prentice-Hall, 1994.

[5] Halsall, Fred, "Data Communications, Computer Networks and Open Systems," (Fourth Edition), Addison-Wesley, 1996.

[6] Hughes, Larry, "Low-Cost Networks and Gateways for Teaching Data Communications," ACM SIGCSE Bulletin, Vol. 21, Number 1, February 1989, pp. 6-11.

[7] Kamel, K. and Riehl, A., "An Instructional Model to Build a Computer Network by Adding Nodes," ASEE Annual Conference Proceedings, June 1992, pp. 1107-1111.

[8] Orfali, R., Harkey, D., and Edwards, J., "Essential Client/Server Survival Guide," Wiley & Sons, 1994.

[9] Ramakrishnan, Sub and Dadfar, Mohammad B., "Client/Server Communication Concepts for a Data Communications Course," ASEE 1997 Annual Conference, 2520-02.

[10] Stevens, Richard, TCP/IP Illustrated Volume 3, Addison-Wesley, 1994.

[11] SUN Microsystems Inc. JDK 1.1, http://www.javasoft.com/products/JDK/1.1/index.html

[12] Tanenbaum, A. S., "Computer Networks," (Third Edition), Prentice-Hall, 1996.

SUB RAMAKRISHNAN is an Associate Professor of Computer Science at Bowling Green State University. From 1985-1987, he held a visiting appointment with the Department of Computing Science, University of Alberta, Edmonton, Alberta. Dr. Ramakrishnan's research interests include distributed computing, performance evaluation, parallel simulation, and fault-tolerant systems.

MOHAMMAD B. DADFAR is an Associate Professor in the Computer Science Department at Bowling Green State University. His research interests include Computer Extension and Analysis of Perturbation Series, Scheduling Algorithms, and Computers in Education. He currently teaches undergraduate and graduate courses in data communications, operating systems, and computer algorithms. He is a member of ACM, IEEE, ASEE, and SIAM.