Asynchronous Communication Between Network Processes

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

This paper concerns a project that provides hands-on exposure to students of a typical undergraduate data communication course. The project is implemented in C++. However, it is applicable to other programming languages as well.

The project deals with blocking and how a server in a TCP/IP network figures out which one of its many client sockets has data to read from. This is resolved using asynchronous communication between a client and a server. Briefly, the server can receive data from any of the clients it is connected to. However, the server does not know which client to wait on. The student has to employ nonblocking socket read and '*select*' mechanisms at the server to correctly identify the client it can read from.

1. Introduction

Many schools offer a data communications course at the undergraduate computer science program. Our program has a required core course in operating systems and networks (CS 327), followed by an elective course that is devoted to data communications and networks (CS 429).

In CS 327, we cover introductory topics including process management, concurrent processes, protocol architecture, TCP/IP suite, brief overview of broadband services, client-server communication and web enabling applications. We assign practical and useful projects that help students to gain an insight into the operating systems concepts and networking. We have assigned some useful projects in the past[1, 2, 4]. Most of our projects are assigned in a UNIX platform. Our students know UNIX and have done extensive programming in C/C++.

This paper concerns a project in CS 429. In this course we discuss transmission media and techniques for transmission and switching. Error detection and protocol techniques for reliable communication are also covered in the course. We also spend time on data compression techniques. Other topics covered include multiplexing techniques and switching technologies.

We spend about three weeks on TCP/IP networking and client-server communication techniques. There are many textbooks available for teaching data communication courses[3, 5, 6, 7]. We have used several different textbooks in our course.

In order to reinforce theoretical concepts taught in the course, we give a number of assignments spread evenly throughout the course. Some of the assignments are handwritten while others require programming.

The project discussed in this paper deals with client-server communication in a TCP/IP environment. Specifically, how the server in a TCP/IP network figures out which one of its many client sockets has data to read from. This is resolved using asynchronous communication between a client and a server. Briefly, the server can receive data from any of the clients it is connected to, in an asynchronous fashion. The server correctly figures out, each time, the client it is receiving data from.

In Section 2, we discuss the project in detail. In section 3, we discuss a program that attempts to solve this problem. In Section 4, we discuss additional extensions to the project and provide some concluding remarks.

2. Client-Server Communication With TCP/IP

During the course we typically assign two client-server projects. The client and server communicate using a message passing paradigm. In one project the client-server instances are created using fork. A pipe is used for communication between them. In the second project we employ TCP/IP form of communication. This paper concerns the second project.

In client-server communication, the client seeks a connection with the server. Then the client and server can communicate. While the first communication is still active another client can attempt communication with the server. Thus, it is conceivable that the server is engaged in communication with multiple clients. The server can manage the multiple clients using one of two different approaches.

One approach is to instantiate a server instance for each client. Thus each server instance, corresponding to a client, is logically disjoint from other server instances. Another approach is to have only one server instance which is common to all clients. In the latter case the server has to know which client endpoint it should wait on to read. This paper concerns a student project to address this problem.

To make the project simpler, we assume there are two client processes, $C1$ and $C2$, that are created using the *fork* system call. The server, S, waits for connection requests from $C1$ and $C2$; S accepts a connection from $C1$ and creates a new socket with the corresponding file descriptor $F1$. Similarly, S accepts a connection from $C2$ with $F2$ as the corresponding file descriptor (see Figure 1). As noted earlier, by using certain mechanisms the server has to correctly determine which one of these two descriptors, $F1$ or $F2$, has data to read from.

The students are required to implement both the client and the server parts. Which of the two clients sends data, next, to server is randomly decided by *C*1. *C*1 communicates this information to *C*2 via a local pipe (see Figure 1).

The project is implemented in C++. However, it is applicable to other programming languages as well.

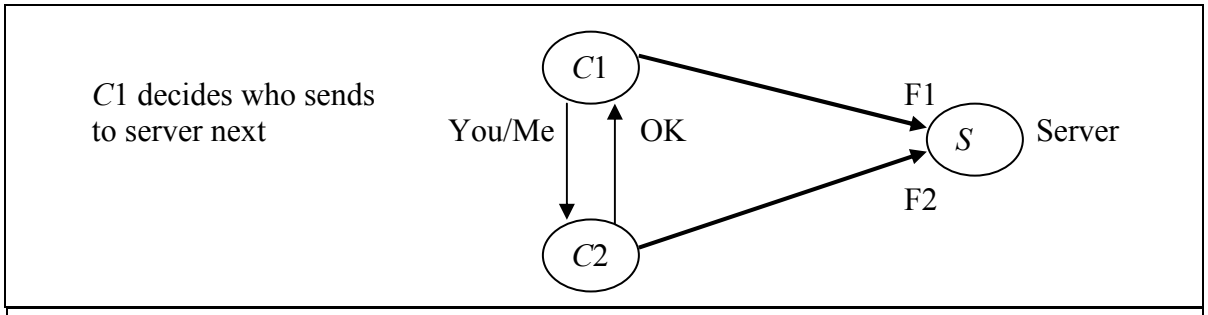

Figure 1: The relationship between clients and server

Recall that the server has two file descriptors, *F*1 and *F*2 that denotes the endpoint with client *C*1 and client *C*2 respectively. Suppose the server does a read on *F*1, and then on *F*2 in this order. This will work if both clients have sent data to the server. However, it fails if client *C*1 has not sent any data, but client *C*2 has. The server is blocked at read on *F*1 forever and the server program hangs since client *C*1 did not send any data at all and the server will never get to read from *F*2.

One might think that the server should be reading *F*2 first and then *F*1. This argument will not work either in the case when *C*2 has no data for the server but *C*1 has data for the server.

The reader will notice the programming dilemma here. The server does not know ahead of time which one of the two descriptors has data so it cannot wait on any one of them. What is required is a "selector" mechanism at the server. The server does not wait on either endpoint. Instead it waits on a selector. The selector condition will be true if one or both of the endpoints has data to read from. Then, additional probing of the selector is done to determine which one has data (*F*1 or *F*2 or both). Then, the server will read from the corresponding descriptor(s).

This is illustrated in the program in Figure 2. To avoid clutter we do not show other parts of the solution in this Figure. Function `getDataFromC1orC2orBoth` takes the endpoints of the connection as an argument. It starts by invoking the `select` function. Both the endpoints are supplied to the function and the function returns only when at least one of these two endpoints has data to read from. The function `FD_ISSET(endpoint, &selectSk)` is then used to check which one has the data. This is a boolean function. If endpoint has data it returns *true*, otherwise it returns *false*. We invoke this function on each of the two endpoints and if there is data we use `read` to read from that endpoint and display at the terminal.

```
int getDataFromC1orC2orBoth (int serverSideHandleClient[2])
{
   fd_set selectSk;
   int buff1[2], buff2[2];        // transferred data
   int howManyDidIGetFrom = 0;

   // prepare to use select()
   FD_ZERO(&selectSk);
   FD_SET(serverSideHandleClient[0], &selectSk);
   FD_SET(serverSideHandleClient[1], &selectSk);

   if ( select(serverSideHandleClient[1]+1, &selectSk, NULL, NULL, NULL) > 0)
   //Above call blocks until there is data from someone!

   {
      if ( FD_ISSET(serverSideHandleClient[0], &selectSk) )
      {
         read(serverSideHandleClient[0], buff1, sizeof(int) );
         cout << "C1 Data: " << buff1[0] << endl;
         howManyDidIGetFrom++;
      }

      if ( FD_ISSET(serverSideHandleClient[1], &selectSk) )
      {
         read(serverSideHandleClient[1], buff2, sizeof(int) );
         cout << "C2 Data: " << buff2[0] << endl;
         howManyDidIGetFrom++;
      }
   }

   else
   {
      cout << "server: select problem" << endl;
      exit(1);
   }

   // clear serverSideHandleClient in selectSK
   FD_CLR(serverSideHandleClient[0], &selectSk);
   FD_CLR(serverSideHandleClient[1], &selectSk);

   return howManyDidIGetFrom;
}
```

Figure 2: Receiving data from clients

3. Students Implementation

The students work in groups. Group size is restricted to 3 or less and the instructor decides the group membership. They are given a copy of Figure 1 and the code in Figure 2, which they will use in their implementation of client and server. We spend time in class explaining the details of Figure 2. The students implement client and server portion. The server will use the code supplied by the instructor (Figure 2).

The students brain storm the overall implementation in class with some help from the instructor. Their design consists of three modules: a main program that can function as a client or server; a server module (which will use Figure 2), and a client module. The students also determine that the two clients need a pipe to exchange control messages between the two pipes.

Now we will discuss the details of their implementation. The complete program is shown in Figure 3a-b. Now we run down the program details. The program starts with a command line argument, the number of times server will receive data. It prompts for option 1 or 2 so the program functions as a server or client respectively (see Figure 3a). If client, it invokes the `clientC1andC2` function.

The client function, `clientC1andC2`, forks and creates a child. Then the parent process connects with the server. Similarly, the child process gets a new socket and does the same. The endpoint at client side is *clientHandle1* for client 1 and *clientHandle2* for client 2 in the program.

```
/**********************************************************************
*                                                                     *
* Description:                                                        *
*  In this assignment, program is launched from two different windows.  *
*  Both programs specify the same number of runs.  One program is the  *
*  server end and the other program is the client end.  Client program  *
*  forks to create a child as client2, and the original parent as      *
*  client1.  Client1 uses socket1 to communicate with server and client2 *
*  uses socket2.  Client1 decides which client sends data to server first*
*  and server uses select() to decide which socket to be selected.     *
*  1. set the numberOfRandomNumbers in program command line;           *
*  2. in every run, client1 generates a random number and sends the    *
*     number and its pid to server, the same with client2;             *
*  3. in every run, who sends the random number next is decides by     *
*     client1(probabilistic: compare the random number with RAND_MAX/2), *
*     client1 communicates this over sp to client2 and receives a      *
*     response from client2;                                           *
*  4. that client then sends the number to server and receives a response*
*     in the usual manner;                                             *
*  5. repeat the loop till finishing all the runs.                     *
*                                                                     *
*  Server receives data up to X times. It displays them and stops!     *
*  The data is from any combination of C1 and C2.                      *
**********************************************************************/

#include <iostream>
#include <unistd>
#include <stdio>
#include <stdlib>
#include <sys/socket>
#include <time>
#include <sys/types>
#include <string>
#include <netinet/in>
#include <arpa/inet>

#define SIZE sizeof(struct sockaddr_in)

void server(int);
void clientC1andC2(int);
int getDataFromC1orC2orBoth (int serverSideHandleClient[2]);

int main(int argc, char *argv[] ) // get number of random numbers
                                  // for child from command line
{
   int clientOrServer;
   cout << "Please specify the terminal type" << endl;
   cout << "1. Server." << endl;
   cout << "2. Client." << endl << endl;
   cout << "Select 1 or 2: \t";
   cin >> clientOrServer;

   if ( clientOrServer == 1 )     // if clientOrServer=1, this window is server
      server( atoi(argv[1]) );

   if ( clientOrServer == 2 )
      clientC1andC2( atoi(argv[1]) );  // if clientOrServer=2, this window is client
   else
      cout << "Wrong input." << endl;

   return 0;
}
```

Figure 3a: Communication between client and server

```
void server(int stopCountForTxn)
{
   int i,j;
   int sk;                 // return value of sockets
                           // clientHandle1 for client1
                           // clientHandle2 for client2
   struct sockaddr_in client;
   struct sockaddr_in add = {AF_INET, 15220, INADDR_ANY}; // binding address
   int len=SIZE;
   fd_set selectSk;
   int serverSideHandleClient[2], connection;   // file descriptors for accept()
   int buff1[2], buff2[2];                      // transferred data
   int rp[1];                                   // responses to client

   // construct socket
   if ((sk = socket(AF_INET, SOCK_STREAM, 0)) < 0 )
   {
      cout << "server:  invalid socket" << endl;
      exit(1);
   }

   // bind socket
   if ( bind(sk, (struct sockaddr *)&add, SIZE) < 0 )
   {
      cout << "server:  invalid bind" << endl;
      exit(1);
   }

   // socket listens
   if ( listen(sk, 5) < 0 )
   {
      cout << "server:  invalid listen" << endl;
      exit(1);
   }

   // socket accepts the connection requests from clients
   j=0;

   do
   {
      if( ( connection = accept( sk,(struct sockaddr *) &client, &len ) ) == -1)
      {
         cout << "server:  invalid accept" << endl;
         continue;
      }

      // serverSideHandleClient[0] for client1, serverSideHandleClient[1] for client2
      serverSideHandleClient[j] = connection;
      j++;
      connection = -1;
   }while(j<2);

   // call the function getDataFromC1orC2orBoth to get the client data
   // stop after receiving data stopCountForTxn times
   int numberOfTimes = 0;
   do
   {
      numberOfTimes = numberOfTimes + getDataFromC1orC2orBoth
(serverSideHandleClient);
   } while (numberOfTimes < stopCountForTxn);

   close(serverSideHandleClient[0]);
   close(serverSideHandleClient[1]);
   close(sk);

   exit(0);
}
```

Figure 3b

The first client decides who gets to send data, this decision is conveyed to the other client through a pipe (see Figure 1). In any event, whichever client is to send data next, it sends through the corresponding descriptor, *clientHandle1* or *clientHandle2*, to the server. The client

quits when data has been sent `stopCountForTxn` times as specified in command line. For brevity, the module `clientC1andC2` is not shown in this paper.

The main work of the server is done in the `server` function. It establishes itself at a port number (15220) and receives a connection from both clients. Then, it calls the function `getDataFromC1orC2orBoth` by passing the endpoint identifiers to it. The function returns 1 or 2 depending on how many clients sent data (*C*1 or *C*2 or both). In any event, the server keeps a count of number of receptions and quits when this equals `stopCountForTxn`. The server then closes the endpoints before exiting (see Figure 3b).

Figure 4 shows a sample run of the complete project. Following compile instructions we invoke the server and then the client, both with command line option value of 5 for stopCountForTxn. The server output shows the client ID and the data from that client.

```
g++ fig2Select.cpp -c
g++ figure3.cpp fig2Select.o -lnsl -lsocket

// Server Run
a.out 5

Please specify the terminal type
1. Server.
2. Client.

Select 1 or 2:  1
C2 Data: 26261
C1 Data: 7605
C1 Data: 12726
C1 Data: 24444
C2 Data: 10331
exit

// Client Run
a.out 5
Please specify the terminal type
1. Server.
2. Client.

Select 1 or 2:  2
```

Figure 4: A sample run of the client-server project

4. Concluding Remarks

In this paper we discussed a hands-on project that can be assigned in a data communications course. The project uses the TCP/IP framework to demonstrate how a server can do non-blocking I/O.

This project is interesting because it integrates a number of concepts covered in data communications courses: message passing, interprocess mechanisms, TCP/IP networking calls, and asynchronous communication.
The project may easily be extended to provide other assignments that help reinforce classroom topics: (i) split the client and server modules into two different programs; (ii) manage the two client instances, *C*1 and *C*2, by two separate threads and not two separate processes.

We have found this exercise to be a valuable addition to classroom topics. After completing the project, asynchronous communication is more real to students. Incremental extensions to the project are easy to construct and build based on the previous knowledge base.

Bibliography

1. Dadfar, Mohammad B. and Ramakrishnan, Sub, "Systems Projects for a Computer Science Course" ASEE 2003 Annual Conference, 1620-04

2. Dadfar, Mohammad B. , Brachtl, Michael, and Ramakrishnan, Sub, "A Simulation of Processor Scheduling Algorithms" The American Society for Engineering Education (ASEE) J. Computers in Education, Vol. XIV, No. 1, January-March 2004.

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

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

5. Shay, W. A., "Understanding Data Communications & Networks," (Third Edition), Thomson - Brooks/Cole, 2004.

6. Stallings, W., " Data & Computer Communications," (Sixth Edition), Prentice-Hall, 2000.

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

MOHAMMAD B. DADFAR
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 and ASEE.

SUB RAMAKRISHNAN
Sub Ramakrishnan is a 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.