AC 2008-2882: NETWORK PROCESSES COMMUNICATION: CLASS PROJECTS

Mohammad Dadfar, Bowling Green State University

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.

Network Processes Communication: A System Project for Students

Mohammad B. Dadfar, Ray Kresman

Department of Computer Science
Bowling Green State University
Bowling Green, Ohio 43403
Phone: (419)372-2337 Fax: (419)372-8061
email: {dadfar, rama}@cs.bgsu.edu

Abstract

In this paper we discuss two projects dealing with communication between network processes. They are assigned in our undergraduate data communications course. The implementation language is C/C++ and the platform is Unix. We introduce a project where students create processes using Unix utilities such as *fork* that includes different levels of processes such as parent, children, and sibling processes. We also describe a practical client-server application students are already familiar with.

1. Introduction

Data communications and networking courses have been among the most popular courses in computer science departments during the past two decades. Most students try to complete at least one course in this area. Instructors assign different types of projects for their data communications and networking courses[1, 2]. In our department we have offered a sophomore level mandatory course (Operating Systems and Networks) that introduces both operating systems and data communications concepts. Following this course, we have elective courses in each of operating systems and data communications. A variety of topics including protocol architecture, transmission technologies, data compression, data integrity, flow control, client server communication and remote procedure calls are covered in the data communications course. There are several text books that address these topics[3, 4, 5].

We consider practical projects as a main component of this course. Hands-on programming projects are used to enhance the learning process and to gain additional insight into specific topics. The projects are implemented in C/C++. However, they are applicable to other programming languages as well.

We start with simple projects where students create child processes using Unix utilities including *fork* statement. There are different levels of processes such as parent, children, sibling, and grand children processes. The communication between the parent-child and sibling processes would be interesting, and at the same time challenging for some students.

We then concentrate on the communication between independent processes and discuss popular concepts such as message passing, TCP/IP networking calls, and asynchronous communication. By this time students have had the opportunity to work with UNIX commands and they are able to develop programs that use communication between processes on different machines.

2. Communication between Processes (Inter-process Communication)

The purpose of this assignment is to establish a communication between several processes (inter-process communication; *IPC*). In the past we have assigned other projects related to different aspects of communication for this course[6, 7]. We simulate an environment where a number of processes communicate with each other.

The processes are created by a parent process P_0. There would be bidirectional communication links (pipes) between processes. One of the objectives of this assignment is to demonstrate the functions performed by intermediate nodes in a data network. When a node P_i receives a message (frame) it checks the destination address and if it is not intended for P_i it will send the frame to the destination node directly or through other nodes.

The number of the processes (nodes) is given as an input. Processes send and receive data frames containing source address, destination address, frame type, and data. A frame is either a *data* frame that contains a message or an *acknowledge* frame that contains either an acknowledgement or a negative acknowledgement. A sending process prepares a *data* frame (or an *acknowledge* frame) and attaches its address as well as the address of the final destination, and the message. Then it sends the frame to the next process. A process that receives a frame checks the destination address and if the frame is intended for this process, it handles the frame (e.g. reads the data and sends an acknowledgment to the sender). Otherwise it sends the frame to another process as mentioned above until the intended destination process eventually receives the frame. A process that sends a data frame expects an acknowledgment from the destination process. The following is a more detailed description of the project.

The parent process P_0 creates `n` processes P_1, P_2, … , P_n (using *fork* statement). Then these `n` + 1 processes communicate with each other using bidirectional pipes (using *socketpair* call). The program should create `m` pipes (`m` >= `n`) between these processes as shown in Figure 1. The bidirectional *Pipe i* connects the parent process P_0 to processes P_i. As Figure 1 shows there are `n` such pipes. There may be additional bidirectional pipes that connect other processes to each other. In general each process P_i has the information about the pipe that connects P_i to P_0 (*Pipe i*) and any additional pipes that connect P_i to other processes. In Figure 1 process P_1 is directly connected only to process P_0 while process P_2 is directly connected to process P_0 and process P_3. It is possible that a process is directly connected to more than two other processes.

Each process P_i has two roles. It may communicate directly with other processes by sending and receiving messages to/from them. It may also act as an intermediate node. The program should accept three numbers (`n`, `m`, and `k`) as command line arguments. Each process P_i will send `k` messages (for example random numbers in the range 1 to 1000) to other processes. At the end process P_0 sends a final frame containing a `-1` to each of the other processes and then each of the processes will send a final frame containing a `-1` to P_0.

Process P_0 will terminate when it has sent its k messages as well as n final frames to each of the other processes and has received the final frames from those processes. Each of the n processes $P_1, P_2, \ldots, P_n$ will terminate when it has sent its k messages and has received the final frame from P_0 and has responded to P_0.

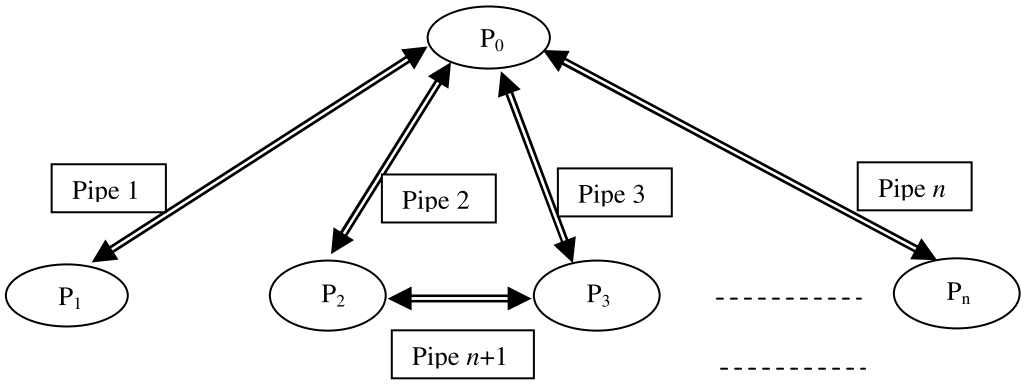

Figure 1: The n+1 processes and the m connecting pipes

There are two types of frames: a *data* frame (contains a 1 as its frame type) and an *acknowledgment* frame (contains a 0 as its frame type). Each frame contains four fields: *source address*, *destination address*, *frame type*, and *data*. The format of a frame is as follows.

Source Address	Destination Address	Frame Type	Data

The source address and destination address could be 0, 1, 2, …, n. If the frame type is acknowledgement, then the last field (*data* field) is checked for acknowledgement (0) or negative acknowledgement (-1). When a process sends a message to another process, it waits until it receives an acknowledgment from that process. A data frame will also serve as an acknowledgment. For example when P_i sends a message to P_j, it waits to receive an acknowledgment from P_j. However, P_i may receive a data frame from P_j and in this case P_i will consider this frame as an acknowledgment from P_j.

Each process may act as a source, a destination and also as an intermediate node. In the latter case a process P_i forwards the received frames to another process. If there is a pipe connecting P_i to the destination process, this can be done directly. Otherwise P_i will send the frame to the parent process P_0 or another connecting process.

A sample solution for this project is given in Figure 2. Due to the length of the actual program we use a pseudo code to describe the solution. Readers may request a copy of the program from the authors.

The main module creates m bidirectional pipes, n processes, and calls the other modules. Each process sends its messages independently. The idea is to interleave the terminal output as seen by the end user. Each process P_i generates k random numbers (messages) that sends to other processes. For each message the sending process P_i generates a random destination address `j`, (`j` = 0, 1, 2, `i`-1, `i`+1, …, `n`), and indicates the source address as `i` (to represent process P_i) and the frame type (1 for a *data* frame). These four items are placed in a data frame and passed to the destination process P_j through a connecting pipe if there is one or through Pipe `i` to parent process P_0.

When process P_j receives a message from P_i it checks the destination address. If the destination is P_j, it handles the message and sends an acknowledgement to P_i. If the destination is not P_j, it forwards the message to the intended destination either directly or through another process.

Extensions to this basic message passing assignment include varying the topology, frame (message) format, and flow control options.

```
/* List the required header files. */
/* Declare the global variables.   */

main (int argc, char *argv[])
 {
   /* Declare the ID of processes P1, P2, P3, … Pn and other variables. */

   /* Check the number of arguments, if it is not correct terminate the program.  */

   /* Create the m >= n pipes.  This can be done by calling a small function that also
      verifies the status of the operation.  */

   /* The parent process P0 creates n processes.  For example, after creating the first
      process P1, the parent receives the process ID for P1 and continues with its
      operation.  The following is an example:*/

      Pid1 = fork();       /* Process P0 create a new process P1.  The ID of P1 is
                              returned to parent.*/
      if (pid1 = 0)        /* The new process P1 receives a 0 for variable Pid1. */
        doProcessP1 (  );  /* P0 will supply the required argument to each process. */

      pid2 = fork();)      /* Parent process P0 continues. */
      if (pid2 = 0)
      .
      .

 }


doProcessP1 (k, and other arguments  )
 {
      /* Create k random numbers and place them in data frames. Send these frames
         to processes P0, P2, P3, … or Pn randomly through bidirectional pipes.
         For each frame wait for an acknowledgement.

         If a received frame is not intended for this process, forward the frame
         to destination process if there is a direct connection to that process.
         Otherwise, send the frame to the parent process P0 or some other process.

         When the process is done and has received the final frame from P0, it
         sends its final acknowledgement to P0 and quits.
      */
 }
```

Figure 2: The Description of A Sample Solution for Process Communication

3. A Practical Application

The previous section discussed inter-process communication among multiple processes. In this section we discuss a project that provides a more real-life context of an actual application that the students are already familiar with. Request For Comments (RFCs) are documents available over the web that describe the inner workings of various protocols and applications[8]. One such document, 'rfc821' relates to the Simple Mail Transfer Protocol (SMTP)[9].

The project is to build a client application to communicate with an SMTP server. The client application is run on command line. The application sends a "hello world" email message to a specific user on a specific machine by talking to an SMTP server, all given as command line arguments or in some other form. The RFC, noted above, describes messages and their formats for transmission between the client and the SMTP server. Students read it on their own and try to implement a few basic primitives.

The instructor may specify a class prototype that captures the client server communication and have the students build the body to complete the application. A typical prototype is given in Figure 3. If necessary Figure 3 may be explained in class. The constructor is called to initiate a connection with the SMTP server on machine, *machineName*, at port number, *portNumber*. We identify the sender and receiver by invoking the method *senderAndReceiver*. The RFC allows the actual body of the email to be sent in multiple packets. When the server receives a '.' on a line by itself, it assumes the end of the message body. The client application invokes the method, *messageBodyPrefix*, as many times as are necessary to send the different parts of the message. And, at the end *messageSuffix* is invoked once to complete the transmission, that causes '.' to be sent to server. Method *logout* closes connection with the server. For each item that is sent to server, the server responds with a message. The student can examine the correctness of her/his solution by checking what the client sent, what the server response is, and finally checking whether the receiver actually received the email that was sent through the application. All of these items along with the program may be submitted for grading. The prototype of Figure 3 involves implementation of a few primitives from *rfc821*: `Helo`, *mail from*, *rcpt to*, *data* and *quit*.

```
Class smtpClient
{
   public String smtpClient (String machineName, int portNumber);
   public String senderAndReceiver (String sender, String receiver);
   public String messageBodyPrefix (String message);
   public String messageSuffix ();
   public void logout ();
}
```

Figure 3: Class prototype for SMTP Client Application

Figure 4 shows a sample interaction between the client and the SMTP server. Actual machine and domain names are not shown. As can be noted from the SMTP rfc, lines that begin with a 3 digit number are server responses, while the others are client transmissions.

As an additional complexity one may implement a number of other primitives that the RFC supports, for example, sending the message as an attachment, verifying the receiver, etc.

```
Connected to smtpServerMachineName.

220 serverDomainName ESMTP timeStampOfConnection   //response
helo                                               //client txn

501 5.0.0 helo requires domain address
helo domainName

250 smtpServerMachineName.domainName               //response
    Hello clientMachine clientIPAddress, pleased to meet you
send from:<senderUserName@domainName>

502 5.5.1 Command not implemented: "send
from:<senderUserName@domainName>"
MAIL FROM:<senderUserName@domainName>

250 2.1.0 <senderUserName@domainName>... Sender ok
RCPT TO:<receiverUserName@ReceiverDomainName>

250 2.1.5 <receiverUserName@ReceiverDomainName>... Recipient ok
data

354 Enter mail, end with "." on a line by itself
hello world
.
250 2.0.0 Message accepted for delivery
quit

221 2.0.0 smtpServerMachineName.domainName closing connection
```

Figure 4: SMTP and Server Communication

4. Concluding Remarks

We described two simple projects for a data communications course. Due to the length of the program, the actual solution was not listed in this paper. The listed short description of the solution should give enough information to develop a program. Some of our students find the projects challenging and others believe these are worthwhile learning experiences and they are excited by completing the projects. The complexity of the projects could be modified by changing the input parameters (number of processes, number of bidirectional pipes, and relationship between processes). For example the first project could be assigned in several different stages from a simple limited number of processes to a more complex set of processes involving parent, children, sibling, and grand children processes connected with many bidirectional pipes.

Bibliography

1. DeHart, J., Kuhns, F., Parwatikar, J., Turner, J., Wiseman, C., and Wong, K., "The Open Network Laboratory," Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education, 2006 (pp 107-111).

2. Elsharnouby, T., Udaya Shankar, A., "Using SeSFJava in Teaching Introductory Network Courses," Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education, 2005 (pp 67-71).

3. Halsall, F., "Computer Networking and the Internet," (Fifth Edition), Addison-Wesley, 2005.

4. Kurose, J., and Ross, K., "Computer Networking," (Third Edition), Addison-Wesley, 2005.

5. Shay W., "Understanding Data Communications and Networks," (Third Edition), Brooks/Cole, 2004.

6. Ramakrishnan, Sub and Dadfar, Mohammad B., "Data Compression and Data Integrity: Projects for Data Communication Courses," ASEE 2005 Annual Conference, 3420-05.

7. Dadfar, Mohammad B. and Ramakrishnan, Sub, "Systems Project for a Computer Science Course", The American Society for Engineering Education (ASEE) J. Computers in Education, Vol. XIV, No. 3 (July - September 2004), pp 2-9.

8. Internet RFC/STD/FYI/BCP Archives. http://www.faqs.org/rfcs/

9. RFC 821 - Simple Mail Transfer Protocol. http://www.faqs.org/rfcs/rfc821.html

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.

RAY KRESMAN
Ray Kresman 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. Kresman's research interests include distributed computing, performance evaluation, parallel simulation, and fault-tolerant systems.