Hands-on Projects Based on Message Passing Between Processes

Mohammad B. Dadfar, Sub Ramakrishnan

Department of Computer Science
Bowling Green State University
Bowling Green, Ohio 43403

Abstract

Message passing plays a critical role in all areas of data and computer communications course offerings. Examples include: TCP/IP and OSI suites, physical network architectures, client-server applications and remote procedures. This paper describes hands-on classroom projects that use the message passing paradigm for a course in data communications.

1. Introduction

The growth of Internet and related technologies has significantly increased the demand for skilled, Information Technology (IT), workforce. However, the supply chain has not grown proportionately. Computer Science departments around the country are trying to be responsive to industry needs in these emerging areas of the computing discipline. Exposure to application issues in the realm of computer communications and client-server computing are especially desirable skills for the graduating student population. Upper level courses on data and computer communications and web architectures have been seeing a surge in student enrollment. There are many textbooks in the area of data communications and computer networks. We have used several different textbooks in our course offerings. Depending on the topics covered in the course, and the level of students taking the course, some textbooks may be more appropriate than the others.

The computer networking area is changing rapidly and the choice of classroom projects has become a major task for instructors. To make the data communications course more useful, the practical part of the course is carefully designed so that students could relate the theoretical concepts to current issues in real-world networking. Many class projects have been proposed in the literature. In this paper we describe several hands-on classroom projects that use the message passing paradigm. The first three projects are simple and may be assigned during the first part of the course. The fourth project (discussed in Section 4 and Section 5) can be assigned after the fifth week of a fifteen-week course.

Message passing plays a critical role in all areas of data and computer communications course offerings. Examples include: TCP/IP and OSI suites, physical network architectures, client-server applications and remote procedures. In the following sections we describe these projects. We also discuss several extensions to these projects. Our students are encouraged by these projects, find them to be a worthwhile learning experience, and feel that the projects integrate...
nicely with the topics covered in classrooms and textbooks.

2. Projects Overview

The general theme of the projects deals with creation and management of a set of processes and providing for a message communication facility among them. Process creation is achieved using the `fork` mechanism and message passing is done through the `pipe` facility. The communication layout attempts to represent some kind of (physical) topology, for example, a family genealogical tree, star, ring, bus and others.

The projects described in this paper have practical relevance and should help reinforce the theoretical concepts introduced in data communication courses. They would be applicable to any Unix platform with a C/C++ compiler, for example, Sun Solaris with a GNU `gcc` compiler.

The projects are suitable for a sophomore offering of a course that deals with data communications and possibly operating systems. Prerequisites include a programming course in C/C++ and data structures. Students are also expected to be proficient working in a Unix environment. The projects introduce students to process management, communication between two processes and handshake mechanisms.

We begin with a simple project on process creation, and work our way up to communication and acknowledgement mechanisms. Where possible, we will provide the code for the programs and suggest possible variations.

3. Projects

a. Use of System Calls

In this assignment the program does some computation (such as generating a certain number of random numbers). It then prints its process ID, the CPU time it has used and finally terminates. Command line argument specifies the number of random numbers to be generated.

The objective of the project is to ensure that students know how to retrieve command line arguments, read Unix `man` pages and use system calls. This can be assigned in the first week of classes.

b. Process Creation and Process Attributes

In this project students fork a child process off of the main process. The child process does some computation (such as generating a set of random numbers). The child prints its process ID and the CPU time it has used. The objective of this assignment is to understand the system call, `fork`, and how to identify the parent and child copies, following `fork`.

Prior to assigning the assignment, it is helpful to spend one class period on the use of `fork` since students generally have difficulty working with concurrent processes. A quiz that tests mastery of forking would be valuable too.

The project solution is given in Figure 1. The `atoi` statement extracts the command line
argument and stores it as an integer. The parent forks a child process. The if statement is satisfied by the child process. The child generates a number of random numbers. It then invokes the `getrusage` system call to print the CPU time it has used.

Suggested variations to the project include the following: (1) Use `times` system call instead of `getrusage`, (2) Parent and child both generate random numbers and print their respective CPU times, (3) Print the starting and ending times for the two processes, (4) Allow the parent to wait for the child before exiting.

```c
// The number of Random Numbers is given as command line argument
// Child generates 'command line specified' random numbers, prints CPU time it has used.
#include <sys/types.h>
#include <sys/resource.h>
#include <sys/time.h>
#include <iostream.h>
#include <unistd.h>
#include <stdlib.h>

int main (int argc,char *argv[]) {
    struct rusage cputimes;
    int pid, number, randnum, test;
    if (argc < 2) {
        cout << "Usage programName    numberOfRandomNumbersToGenerate" << endl;
        exit(1);
    }
    number = atoi(argv[1]);
    pid = fork();  /* create a child process */
    if (pid == 0) {
        /* child process */
        for (int i = 0; i < number; i++)
            randnum = rand();
        test = getrusage(RUSAGE_SELF, &cputimes);
        cout << "I am the child; my ID " << getpid() << " seconds and " << cputimes.ru_utime.tv_usec << " microseconds " << endl;
        exit(0);
    }
}

//Run of the program
 g++ programName.cpp  -o programName
programName 123456
I am the child; my ID 11905   User time: 0 seconds and 47824 microseconds
```

**Figure 1: A Sample Solution for Project (b)**

c. Communication Between Parent and Child

The parent forks a child process. Parent sends a `Hello` message to child. Child responds with another message to the parent. Each process prints the message it receives and exit. Use `pipe` system call to do the project.

The project can be completed by creating a `pipe` system call. The pipe has two ends, one designated for reading and the other for writing. Both processes take turns in writing and
reading to the pipe. A sample solution is shown in Figure 2.

```c
#include <signal.h>
#include <fcntl.h>
#include <iostream.h>           /* c++ I/O headers */
#include <unistd.h>

int main (int argc,char *argv[]) 
{
    char buffer[512];
    int pid, pipeName[2];
    pipe (pipeName);  /* [0] is 'read' end; [1] is 'write' end */
    pid = fork();
    if (pid == 0)
    {
      /* child process */
      read (pipeName[0], buffer, 512);
      cout << "Child print: " << buffer << endl;
      write (pipeName[1], "Hello From Child to Parent ", 512);
      exit(0);
    }
    else
    {
      /* parent process */
      write (pipeName[1], "Hello From Parent to Child", 512);
      cout << buffer;                 // print to term window
      read (pipeName[0], buffer, 512);
      cout << "Parent print: " << buffer << endl;
      write (pipeName[1], "Hello From Child to Parent", 512);
      exit(0);
    }
}
```

```bash
g++  programName.cpp  -o  programName
programName
Child print: Hello From Parent to Child
Parent print: Hello From Child to Parent
```

**Figure 2: A Sample Solution for Project (c)**

Other variations to this project include the following: (1) Use a command line argument to determine who writes to the pipe first. (2) Use a command line argument to specify who writes to the terminal first! This can be implemented using another pipe for synchronization: If it is parent’s (child’s) turn to print to the terminal, it prints the *Hello* message to the terminal and writes an *I am done* message to the synchronization pipe. The other reprocess then reads the message from the synchronization pipe before printing its *Hello* message to the terminal.

This project can also be extended to include a collection of processes that communicate in a certain fashion as explained in the next section.

### 4. A Comprehensive IPC Project

We first describe the specific details of the project and then share the problem-solving phase. The project deals with message passing among a group of four processes (*P*₁, *P*₂, *P*₃ and *P*₄) connected as a family genealogical tree. Here, *P*₁ is the message originator. The message may be addressed to one of *P*₂, *P*₃, or *P*₄. However, *P*₂ acts as a router (intermediate node) and consumes messages addressed to it or passes to either *P*₃ or *P*₄ depending on the destination address. In addition to handling the incoming data frames from *P*₁, process *P*₂ may also send its
own messages to processes $P_3$ and/or $P_4$. The acknowledgement travels similarly in the reverse direction. Flow control is enforced by using a *stop-and-wait* protocol. For example, $P_1$ may only send the next message after it receives acknowledgment for the previous one. This project introduces students to certain aspects of communications protocols including how a router (intermediate node) handles incoming messages.

A formal statement of this project is given in Figure 3.

![Diagram of communication between processes](image)

The purpose of this project is to establish a communication between four processes, $P_1$, $P_2$, $P_3$, and $P_4$. There would be three bi-directional communication links (pipes) between processes as shown in the following diagram. Use `socketpair` calls to create bi-directional communication links.

Process $P_1$ prepares $n$ data frames intended for one of the other processes. Each data frame has three fields, source address, destination address, and the message (data). The addresses of these processes are $1$, $2$, $3$, and $4$ respectively. $P_1$ attaches its address (i.e. 1) as the first field and the address of the final destination (i.e. 2, 3, or 4 chosen randomly) as the second field. The message (e.g. a random number $n1$ in the range 0 to 100) would be the third field. Then $P_1$ sends the frame to $P_2$. Prior to sending the frame, $P_1$ displays the data and the final destination as follows:

**Process $P_1$: Message $n1$ Sent to Process $P_i$**

where ‘$i$’ is 2, 3, or 4. Process $P_2$ checks the destination address and if it is 2 (i.e. the frame is intended for $P_2$), displays the data and sends an acknowledgment to $P_1$. Otherwise (without looking at the data) sends the frame to the intended destination process ($P_3$ or $P_4$) and waits for an acknowledgment from that process. After receiving an acknowledgment from the destination process, $P_2$ sends an acknowledgment to $P_1$.

Process $P_2$ will also generate and send $m$ data frames to either $P_3$ or $P_4$. Prior to sending each frame, $P_2$ displays the data and the final destination as follows:

**Process $P_2$: Message $n1$ Sent to Process $P_j$**

where ‘$j$’ is either 3 or 4. Process $P_3$ and $P_4$ wait to receive a data frame from $P_2$. Upon receiving a frame, they display the data and the identity of the original sender ($P_1$ or $P_2$) and send an acknowledgment to $P_2$.

The destination process ($P_2$, $P_3$, or $P_4$) after receiving the data frame, extracts the data (e.g. a random number in the range 0 to 100) and displays a message similar to:

**Process $P_i$: Message $n1$ Received from Process $P_j$**

Processes $P_1$ and $P_2$ send a total of $n + m$ data frames ($n$ and $m$ are accepted as command line arguments). The output of $P_2$, $P_3$, and/or the output of $P_1$ or $P_4$ should be interleaved. After sending all of its data frames ($n$ data frames) $P_1$ sends the final frame containing the message *Bye* to $P_2$ and upon receiving the acknowledgement it terminates itself. After receiving the final data frame from $P_1$, process $P_2$ may need to send more (its own) data frames to $P_2$, and/or $P_4$. At the end process $P_2$ sends a final data frame (containing the message *Bye*) to each of the processes $P_1$ and $P_4$ and terminates itself. Upon receiving the final frame, each of the processes $P_1$ and $P_4$ will terminate themselves.

Figure 3: Problem Statement for the IPC Project
5. Discussion of the IPC Project Solution

A sample solution for the IPC project is given in Figure 4. Process $P_1$ uses a pipe (pipeName[2]) to communicate with $P_2$. (See doProcessP1 of Figure 4.) $P_1$ generates $n$ random numbers in the range 0 to 100. These are the messages that $P_1$ are sending to other processes. For each message $P_1$ generates a random destination address, 2, 3, or 4, which designates whether the message is for $P_2$, $P_3$ or $P_4$. It also assigns the source address as 1 to represent process $P_1$. These three numbers are placed in a frame and passed to $P_2$ through the pipe. The frame has three fields, the source address, the destination address, and the message.

When process $P_2$ receives a message from $P_1$, it first determines the destination address. If the destination is $P_2$, process $P_2$ consumes it. If the destination is $P_3$, $P_2$ forwards the message to $P_3$. A second pipe (pipeName[3]) allows $P_2$ to send the message to $P_3$. If the destination is $P_4$, $P_2$ uses a third pipe (pipeName[4]) to send the message to $P_4$. $P_3$ and $P_4$ send acknowledgements of receipt of their messages to $P_2$ and if the original source is $P_1$, then $P_2$ will send an acknowledgement to $P_1$.

Process $P_2$ also generates $m$ random numbers in the range 0 to 100 and for each of these numbers it generates a random destination address, 3 or 4, which designates whether the random number is for $P_3$ or $P_4$. It also designates the source address as 2 to represent $P_2$. If the destination is $P_4$, $P_2$ sends the message to $P_3$ through the second pipe (pipeName[3]). If the destination is $P_4$, $P_2$ sends the message to $P_4$ through the third pipe (pipeName[4]). The functionality of $P_2$ is accomplished in the module doOtherPartsOfP2 (see Figure 4).

Process $P_2$ does not wait on $P_1$ to finish before generating its own messages. $P_1$ and $P_2$ generate their messages independently. The idea is to interleave the terminal output as seen by the end user.

The functionality of $P_3$ or $P_4$ is given in the module doProcessP3orProcessP4 (see Figure 4). Note that $P_3$ (or $P_4$) receives a message from $P_2$ and sends an acknowledgment back to $P_2$. The process continues in a loop until it receives the termination frame, Bye, from $P_2$.

The main module creates the three pipes, the three processes, and calls the other modules. To run this program, type: programName n m where $n$ is the number of messages sent by $P_1$ and $m$ is the number of messages generated by $P_2$.

A sample run of the program is shown in Figure 5. The output shows the random numbers that were sent by $P_1$ and either consumed or forwarded by $P_2$, and the identity of the recipient process that the number is forwarded to. It also includes the random numbers generated by $P_2$ that were intended for either $P_3$ or $P_4$.

Extensions to this basic message passing assignment include varying the topology, frame (message) format, and flow control options.
#include <sys/types.h>
#include <sys/socket.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>
#include <fcntl.h>

int pipeName[5][2];
int source, dest, P1num, P2num, P3num, P4num, numFromP1;
int i, n, m, turn, message[3], Bye = -1;

main (int argc, char *argv[]) {
    int i, pid2, pid3, pid4;    /* the ID of processes P2, P3, and P4 */
    /* check the number of arguments, if it is not 2 terminate the program */
    if (argc != 3) {
        perror("You need to supply two arguments: i.e. a.out n m");
        exit(1);
    }
    pipePairCreation (pipeName[2]);   /* create a pipe between P1 and P2 */
    pipePairCreation (pipeName[3]);   /* create a pipe between P2 and P3 */
    pipePairCreation (pipeName[4]);   /* create a pipe between P2 and P4 */

    pid2 = fork();
    if (pid2 != 0) doProcessP1 ( atoi ( argv[1] ));
    pid3 = fork();
    if (pid3 != 0)            /* P2 continues */
        { pid4 = fork();
            if (pid4 != 0) /* P2 continues */
                { doOtherPartsOfP2 (atoi(argv[2]));
                    else /* do process P4 */
                        doProcessP3orProcessP4 ( 4 );
                }    /* end of for loop */
        }    /* end of for loop */
    if (pid3 == 0)            /* do process P3 */
        doProcessP3orProcessP4 (3);
    exit(0);
}

pipePairCreation (int pipeNameDummy[2])
{
    if ((socketpair(AF_UNIX, SOCK_STREAM, 0, pipeNameDummy) ) == -1)
        { perror("unsuccessful pipe:");
            exit(1);
        }
}

writeMessage (pipeEnd, source, destination, numberToWrite)
{
    int message [3];
    message[0] = source;
    message[1] = destination;
    message[2] = numberToWrite;
    write(pipeEnd, message, sizeof(message));
}

doProcessP1 (n) 
{
    /* Create n random numbers. Send to P2, P3, or P4 through P2. */
    for (i = 1; i <= n; i++)
        { P1num = rand() % 101;
            dest = 2 + rand() % 3;
            source = 1;
            switch (i, P1num , " Sent to ", dest);
            writeMessage (pipeName[2][0], source, dest, P1num);
            read(pipeName[2][0], message, sizeof(message));
        }    /* end of for loop */
    /* P1 is finished. Now send the final frame */
    writeMessage (pipeName[2][0], source, 2, Bye);
    /* wait for acknowledgement from P2 */
    read(pipeName[2][0], message, sizeof(message));
    exit(0);
}

Figure 4: A Sample Solution for the IPC Project (Part 1)
doOtherPartsOfP2 (m) {
  i = 1;               /* index to the loop for P2 */
  numFromP1 = 0;       /* initialize the data from P1 */
  while ( (numFromP1 != Bye) || (i <= m) )
  {
    if ( (numFromP1 != Bye) && (i < m) )
      turn = 1 + rand() % 2;  /* decide whose turn it is */
    else if (numFromP1 != Bye)
      turn = 1;               /* P2 is done, P1’s turn */
    else if (i < m)
      turn = 2;               /* P1 is done, P2’s turn */
    if ((turn == 1) && (numFromP1 != Bye))       /*** P1’s turn ***/
    {
      /* read message from P1 */
      read(pipeName[2][1], message, sizeof(message));
      dest = message[1];  /* determine the destination */
      if (dest == 2)   /* message is meant for P2 */
      {
        numFromP1 = message[2];
        if (numFromP1 != Bye)
          TermOutput(2, numFromP1,* Received from *,message[0]);
      }
      else if ( (dest == 3) || (dest == 4) )  
        write(pipeName[dest][0], message, sizeof(message));
    }
    /* return acknowledgement to P1 */
    else                                     /***** P2’s turn *****/
    {
      if (numFromP1 != Bye)
        write(pipeName[2][i], "I am ready ", 12);
    }      /* end of if {for p1’s turn} */
    i = i + 1;    /* increment P2’s loop index */
  }                      /* end of while  */
/* P2 is finished, P2 sends final frame to P3, P4 and P1 */
writeMessage (pipeName[3][0], 2, 3, Bye, sizeof(message));
writeMessage (pipeName[4][0], 2, 4, Bye, sizeof(message));
writeMessage (pipeName[2][1], 2, 1, Bye, sizeof(message));
exit(0);     /* P2 is done */
}

doProcessP3orProcessP4 ( processNO) {
  /* Code is nearly the same for P3 & P4.  Argument specifies 3 or 4 */
  /* receive a frame from P2 */
  read(pipeName[processNO][1], message, sizeof(message));
  /* if the number is -1, the process should terminate */
  while (message[2] != Bye)
  {
    TermOutput(processNO,message[2]," Received from ",message[0]);
    /* send an acknowledgement to P2 */
    write(pipeName[processNO][1], "I am ready", 12);
    /* receive the next frame from P2 */
    read(pipeName[processNO][1], message, sizeof(message));
  }
  exit(0); /* P3/P4 is done */
}

TermOutput ( char * printedBy, int ID1, char * msg, int ID2) {
  printf ("Process %d: Message %d %s Process %d 
",printedBy,ID1,msg,ID2);
}

Figure 4: A Sample Solution for the IPC Project (Part 2)
6. Concluding Remarks

We introduced a number of simple projects suitable for a sophomore course in data communications and possibly operating systems. A description of such a course is given in [2]. The projects are simple enough to be completed in a week or less. Yet, each one builds upon the foundations from the previous project. Our students are encouraged by these projects, find them to be worthwhile learning experiences and feel that the projects nicely integrate with the course content.

Bibliography


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, IEEE, ASEE, and SIAM.

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.