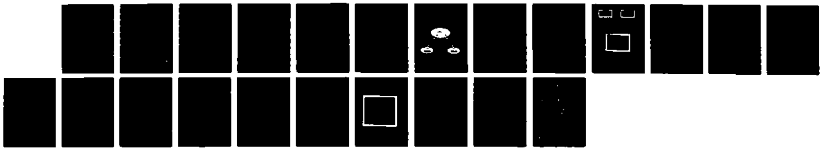
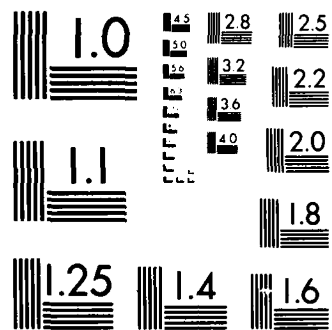
AD-A188 353 EXPERIENCE WITH BERKELEY UNIX INTERPROCESS COMMUNICATIONS(U) NORTH CAROLINA UNIV AT CHAPEL HILL DEPT OF COMPUTER SCIENCE H M ABDEL-WAHAB 1987 1/1

UNCLASSIFIED N0014-86-K-0680 F/G 12/5 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963 A

DTIC FILE COPY

4

AD-A188 353

Contract N00014-86-K-0680

Experience with Berkeley *UNIX** Interprocess Communications

Hussein M. Abdel-Wahab
Department of Computer Science
North Carolina State University
Raleigh, NC 27695

Abstract

At many universities and institutions throughout the world it is now very common to have a network of computers, each running the Berkeley *4.3BSD* version of *UNIX* or an equivalent version such as *ULTRIX*. This paper is to help users of these versions of *UNIX* to explore and experiment with the interprocess communications and networking facilities. We present a series of client/server programs that can be used as a model for writing distributed applications. We describe how users can test and experiment with these programs. Readers are assumed to be familiar with the C programming language and some version of *UNIX*.

...ment has been approved
... release and sale, its
...tion is unlimited.

DTIC
ELECTE
NOV 2 3 1987
A

*UNIX is a trademark of AT&T Bell Laboratories

1 Introduction

Currently, many universities and research organizations have a network of computers each running *4.3BSD* version of *UNIX* or a compatible version such as *ULTRIX* or *DG/UX*. A major addition to these versions of *UNIX* is the *Interprocess Communications* (*IPC*) facilities that allow unrelated processes on the same or different machines to communicate and exchange messages. This make it possible to write distributed applications and provide a better environment for users to collaborate and access remote resources and utilities (See for example, Abdel-Wahab et. al. [1,2]).

In this paper we introduce the important basic features of the *IPC* facilities and show how to write distributed applications based on these facilities. The users are assumed to be familiar with some version of *UNIX* (see for example, Kernighan and Pike [4]), and the C programming language (Kernighan and Ritchie[5]). A tutorial introduction to *IPC* is given in Coffield [3] and Sechrest [8]. For a comprehensive coverage see Leffler et. el. [6].

There are several *domains* of communication, the most frequently used ones are the *internet* domain (*INET*) and the *UNIX* domain. The *INET* domain employs the *DARPA* standard protocols *TCP/IP* and UDP/IP (Leiner et. el. [7]). The *UNIX* domain is basically used for efficient communication between unrelated processes residing on the same machine. The *INET* domain allows processes running on different machines (as well as on the same machine) to communicate. In most cases the *INET* domain is preferable to the *UNIX* domain, since applications written in the *INET* domain may run on any configurations of machines.

In each domain, there are two methods of communication: *stream* and *datagram*. In the stream method the two communicating processes have to establish a connection or a *virtual circuit* before they can exchange messages. This style of communication provides bidirectional, reliable, sequenced and unduplicated flow of data. On the other hand, datagram communication does not require any connection between the two processes as each message is addressed individually but

there is no guarantee regarding delivery, sequencing, or duplication of messages. The choice between the two methods is usually based on the application semantics and the required performance.

This paper is organized as follows. Section 2 is concerned with the *INET* domain, the first part deals with stream communication and the second with datagram communication. Each part is described by giving a pair of *server* and *client* programs. The programs are tested using two terminals (or two windows of a single terminal) connected to two different hosts or to a single host. Section 3 is devoted to the *UNIX* domain. Section 4 deals with the cases where a server communicates with two or more clients and input multiplexing is required. Section 5 is our conclusion.

2 Internet Domain

In this section we describe how two processes in the *INET* domain communicate with each other. The first step for a process to communicate with another is to create a *socket*. A socket is an end point of communication and is created with socket system call:

```
s = socket(domain, type, protocol);
```

For example, to create a stream socket in the internet domain we use:

```
s = socket(AF_INET, SOCK_STREAM, 0);
```

Here, we specify 0 in place of the protocol arguments so the system will use the default standard protocol.

We distinguish between the two communicating processes by calling one the server and the other the client. The server process has to "bind" its socket to a known "name" so that the client process can use this name to establish a connection with the server. The name format in the *INET* domain consists of two parts *host_address* and *port_number*. The host_address is a 32-bit network wide address assigned to the machine (for example, 128.109.135.1 is the address of a machine at

North Carolina State University called "ncsu" and 128.109.136.82 is for a machine at UNC-Chapel Hill called "unc"). The port_number is an integer in the range 0-50,000 (the subrange 0-1024 is reserved for privileged use, for example, the remote login program *rlogin* uses port number 513 and the file transfer program *ftp* uses port number 21). To bind a socket s to a name we use:

```
bind(s. name. sizeof(name)):
```

nama is a structure where its fields are to be filled out appropriately as in the following example.

```
struct sockaddr_in name;
........
name.sin_family = AF_INET;
name.sin_addr.s_addr = INADDR_ANY;
name.sin_port = 0;
bind(s. &name. sizeof(name));
```

Here, the constant INADDR_ANY, means any valid address of the host. If we know the host address, e.g, 128.109.136.82, we use it as:

```
name.sin_addr.s_addr = inet_addr("128.109.136.82");
```

and if the host name is known, e.g., "ncsu" it is used as:

```
struct hostent *hp;
.......
hp = gethostbyname("ncsu");
bcopy(hp -> h_addr. &(name.sin_addr.s_addr). hp->h_length);
```

To choose a port number, we either use:

```
name.sin_port = 0;
```

in which case, the system selects the next free port, or if you know the number of a particular free port, e.g., 1234, it is used as:

```
name.sin_port = 1234;
```

If the system selects a free port, the selected port number may be displayed as follows:

```
length = sizeof(name);
getsockbyname(s, &name, &length);
printf("%d", ntohs(name.sin_port));
```

After the server creates a socket and binds it to an *INET* name, it executes:

```
listen(s, n);
```

to specify, as n, the maximum number of outstanding connections to be queued awaiting acceptance by the server. This system call is non-blocking, it just sets up the socket and makes it ready for accepting connections.

The client process creates its own socket, sc, but it does not need to bind it to a name like the server. Then it initiates a connection to the server process using the connect system call:

```
connect(sc, &server_name, sizeof(server_name));
```

When the connection request arrives at the server, it is accepted using:

```
s1 = accept(s, 0, 0);
```

Where s1 is an "auxiliary" socket descriptor to be used in communicating with the client process. This system call causes the server process to be blocked waiting for the client process to be connected. The server may accept other connections as shown in Fig. 1.

Once a connection is established between a client and a server, data may flow in both directions. The client may send data using:

```
send(sc, data, sizeof(data), 0);
```

and the server may receive the arrived data using:

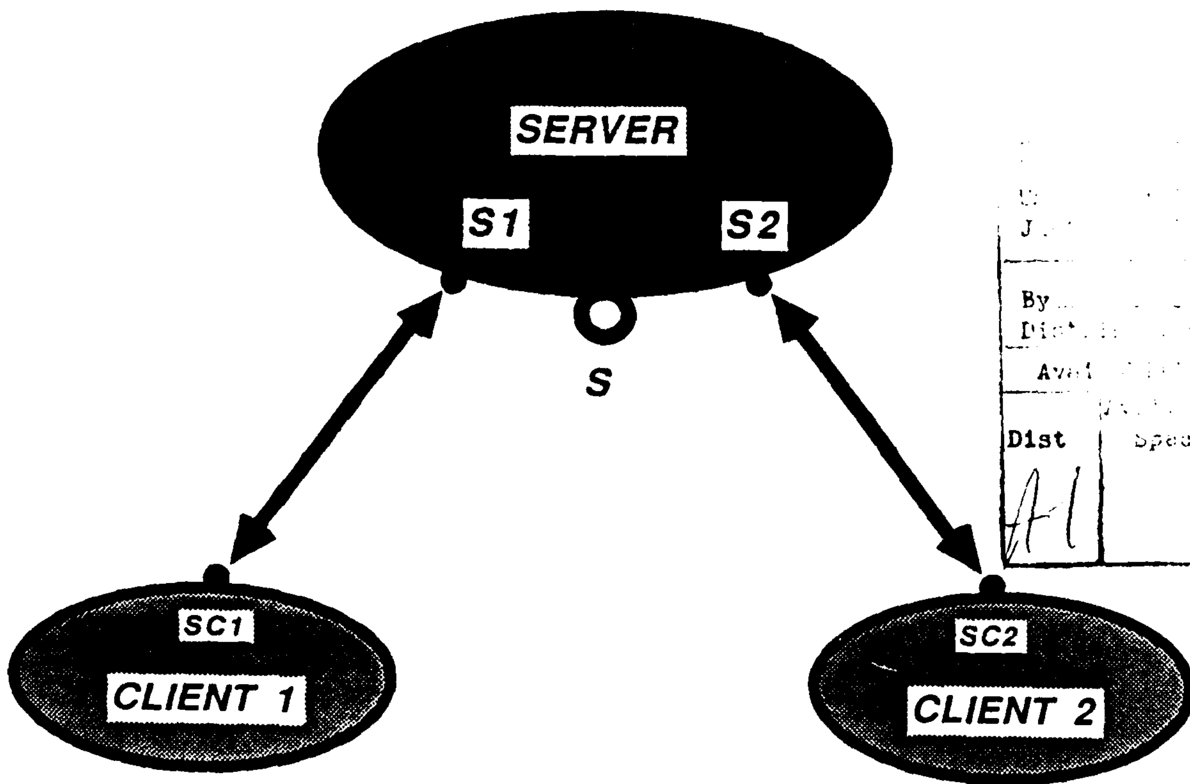
```
recv(s1, buffer, sizeof(buffer),0);
```

When the server and the client are done with their conversation, each may close its end of the connection using the close call:

```
close(socket);
```

Example Programs

In this paper we choose the names of our example programs to reflect their functions using the following abbreviations:



By
Dist
Special

Fig. 1: Server and two clients

I : for *INET*,

U : for *UNIX*,

V : for Virtual circuit,

D : for Datagram

S : for Server,

C : for Client

Each program includes a "def" file that contains:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <stdio.h>
#include <netinet/in.h>
#include <netdb.h>
#include <signal.h>
#include <sys/time.h>
#define   TRUE 1
```

2.1 *INET* Virtual-circuit programs

In order to test and practice with the concept of communication between two processes in the *INET* domain using a virtual circuit connection, we wrote two programs in Fig. 2: a server program *IVS* and a client program *IVC*.

The server process *IVS* starts by creating a socket ss (lines 10-14), then binds it to any free port in the host machine (lines 16-23), advertises the selected port number by displaying the message: `socket has port # ....` and finally listens for a connection attempt by the client process *IVC* (line 33). When the client process attempts to connect to the server process, the connection is accepted in socket s1 (line 34). Following the connection establishment, the server enters a

```
0 #include "def"
1 main()
2 {
3       int ss, s1;
4       struct sockaddr_in name;
5       charbuf[1024];
6       int cc;
7       intlength;
8       struct   sockaddr from;
9       /* Create socket from which to read. */
10      ss = socket (AF_INET,SOCK_STREAM,0);
11      if ( ss<0 ) {
12            perror("opening virtual circuit socket");
13            exit(-1);
14      };
15      /* form INET socket name and bind it to ss */
16      name.sin_family = AF_INET;
17      name.sin_addr.s_addr = INADDR_ANY;
18      name.sin_port = 0;
19      if ( bind( ss, &name, sizeof(name) ) ) {
20            close(ss);
21            perror("binding name to virtual circuit socket");
22            exit(-1);
23      }
24      /* Find out assigned port number and print it out */
25      length = sizeof(name);
26      if ( getsockname (ss,&name,&length) ) {
27            perror("getting socket name");
28            exit(0);
29      }
30      printf("Socket has port #%d\n", ntohs(name.sin_port));
31      /* listen for a connection from the client process vic */
32      printf("... waiting for connection ...\n");
33      listen(ss,1);
34      s1 = accept(ss, 0, 0);
35      printf ("connected to client\n");
36      printf ("... waiting for messages ... \n");
37      do {
38            if ( (cc=recv(s1,buf,sizeof(buf),0) ) < 0 )
39                  perror("receiving virtual circuit packet");
40            if (cc > 0) {
41                  buf[cc] = NULL;
42                  printf("message received: %s\n", buf);
43            }
44            else {
45                  printf("message received: EOF ");
46                  close (s1);
47                  close (ss);
48                  printf("... disconnect\n");
49                  exit (0) ;
50            }
51      }
52      while (TRUE);
53 }
```

Fig. 2a: *IVS* (server)

```
0 #include "def"
1 main( argc, argv )
2 int argc;
3 char *argv[];
4 {
5       int       sc;
6       struct      sockaddr_in server;
7       struct   hostent *hp, *gethostbyname();
8       char buf[512];
9       int rc;
10      /* Create socket on which to send. */
11      sc = socket (AF_INET,SOCK_STREAM,0);
12      if (sc<0) {
13            perror("opening virtual circuit socket");
14            exit(-1);
15      }
16      /* Construct name of socket to send to and connect to it */
17      server.sin_family = AF_INET;
18      if ( (hp = gethostbyname(argv[1])) == NULL ) {
19            close(sc);
20            fprintf(stderr, "Can't find host %s\n", argv[1]);
21            exit(-1);
22      }
23      bcopy ( hp->h_addr, &(server.sin_addr.s_addr), hp->h_length);
24      server.sin_port = htons(atoi(argv[2]));
25      if ( connect(sc, &server, sizeof(server)) < 0 ) {
26            close(sc);
27            perror("connecting stream socket");
28            exit(0);
29      }
30      printf("Connected to the server\n");
31      printf("type message (to finish type CTRL-D)\n");
32      /* Send message. */
33      do {
34            rc=read(0,buf, sizeof(buf));
35            if (send(sc, buf, rc, 0) <0 )
36                  perror("sending virtual circuit message");
37      }
38      while (rc > 0);
39      printf ("EOF... disconnect\n");
40      close(sc);
41      exit (0);
42 }
```

Fig. 2b: *IVC* (client)

Fig. 2: INET Virtual-circuit programs

```
%IVS
Socket has port# 1234
...waiting for connection...
connected to client
...waiting for messages...
message recieved: Hello world!
message recieved:EOF ... disconnect
```

terminal Ts: running the server

```
%IVC unc 1234
Connected to server
type message(CTRL-D to finish)
Hello world!
EOF ... disconnect
```

terminal Tc: running the client

Fig. 3a: Testing using two terminals

```
%IVS
Socket has port# 1234
...waiting for connection...
connected to client
...waiting for messages...
message recieved: Hello world!
message recieved:EOF ... disconnect
```

```
%IVC unc 1234
Connected to server
type message(CTRL-D to finish)
Hello world!
EOF ... disconnect
```

Fig. 3b: Testing using two windows

Fig. 3: Testing the server and the client programs

loop (lines 37-52), receiving (lines 38-39) and displaying (lines 40-43) any message that arrives on the connection until the client sends an `EOF` signaling the end of conversation[1]. Upon receiving the `EOF` message, the *IVS* process terminates (lines 44-50).

In Fig. 2b, the client process *IVC* is initiated by typing:

```
% IVC host_name port_number
```

where `host_name` is the name of the machine running the server process *IVS*, and `port_number` is the number displayed by the server process. The *IVC* process starts by creating a socket sc (lines 11-15), forming the server_name (lines 17-24) from the host name (argv[1]) and the port number (argv[2]). Then it connects the socket sc to the server_name (lines 25-29) and enters a loop (lines 33-38) reading any message typed on the keyboard (line 34) and sending it to the server process (lines 35-36) until `CTRL-D` is typed to end of conversation with the server.

To test how the two programs interact with each other, we may use two terminals T_s and T_c beside each other as shown in Fig. 3a. We use terminal T_s to run the server program. The server process displays the selected socket number (e.g., `Socket has port # 1245`) and waits for a connection by the client process. We use the second terminal T_c to run the client process *IVC* . To run *IVC* we provide two arguments: the remote machine name and the port number where the server is waiting. After the client process is connected to the server process it reads any character typed on the keyboard and sends it out to the server process *IVS*. Whatever typed on T_c will appear instantly at T_s. For example, if we type the message "`Hello World!`" on T_c it will appear on T_s as "`Message received: Hello World!`". To end the session we type `CTRL-D` on T_c and both processes will terminate after displaying the message "`EOF ... disconnect`"

If we have access to only one machine, we can still conduct the above test, since in the *INET* domain processes in the same machine or in different machines can communicate with each other. We use terminal T_s to run the server *IVS* and

[1] In *UNIX* an End-Of-File is signaled by typing `CTRL-D`

after it displays the port number, use terminal T_c to run the client, where the machine name is the one running both the server and the client processes.

If we like to use a single terminal for this test, we need to create two "windows" as shown in Fig. 3b. Windows can be created using any available window management system such as the Berkeley UNIX 4.3BSD *window* program that runs on any ASCII terminal, or the MIT *X-windows* that runs on a variety of workstations including *SUNs* and *DEC VAXs*. In our test, the top window is used as terminal T_s while the bottom window is used as terminal T_c.

2.2 *INET* Datagram Programs

To create a datagram socket in the *INET* domain we use:

```
s = socket (AF_INET, SOCK_DGRAM, 0);
```

Fig. 4a shows the server program *IDS* and Fig. 4b shows the client program *IDC*. The server process *IDS* starts by creating socket ss (lines 11-15), binds it to an *INET* port (lines 17-24) and advertises the chosen port (lines 26-31). Then enters a loop (lines 34-46) where it receive messages sent by the client process. In contrast to the virtual circuit program (*IVS*), this program does not execute listen to queue connections and does not *accept* connections. Also, instead of using recv, it uses recvfrom to read the incoming messages (line 35).

In Fig. 4b, the client process *IDC* creates a socket sc (lines 11-15), forms the server name from the command line arguments argv[1] and argv[2] (lines 17-24). Then sends every line as an individual datagram (line 26-32) using:

```
sendto(sc, message, message_length, 0, &server, sizeof(server));
```

Note that the server name &server is attached with every message going from process *IDC* so that the server process *IDS* can identify the source of the message. The two processes terminate when CTRL-D is typed.

To test these two program we follow the same procedure we described earlier for testing the virtual circuit programs *IVS* and *IVC*.

```
0 #include "def"
1 main()
2 {
3       int ss;
4       struct     sockaddr_in name;
5       char      buf[1024];
6       int       cc;
7       int       length;
8       struct    sockaddr from;
9       int       fromlen;
10      /* Create socket from which to read. * /
11      ss = socket (AF_INET,SOCK_DGRAM,0);
12      if ( ss<0 ) {
13            perror("opening datagram socket");
14            exit(-1);
15      };
16      /* form INET socket name and bind it to ss * /
17      name.sin_family = AF_INET;
18      name.sin_addr.s_addr = INADDR_ANY;
19      name.sin_port = 0;
20      if ( bind( ss, &name, sizeof(name)) ) {
21            close(ss);
22            perror("binding name to datagram socket");
23            exit(-1);
24      }
25      /* Find out assigned port number and print it out * /
26      length = sizeof(name);
27      if ( getsockname (ss,&name,&length) ) {
28            perror("getting socket name");
29            exit(0);
30      }
31      printf("Socket has port #%d\n", ntohs(name.sin_port));
32      /* Read from the socket * /
33      printf("... waiting for messages ...\n");
34      for (;;) {
35            if ( (cc=recvfrom( ss,buf,sizeof(buf), 0,0,0)) < 0 )
36                  perror("receiving datagram packet");
37            if (cc > 0) {
38                  buf[cc] = NULL;
39                  printf("message received: %s\n", buf);
40            }
41            else{
42                  printf("message received: EOF ... exit");
43                  close(ss);
44                  exit(0);
45            }
46      }
47 }
```

Fig. 4a: *IDS* **(server)**

```
0 #include "def"
1 main( argc, argv )
2 int argc;
3 char *argv[];
4 {
5       int       sc;
6       struct     sockaddr_in server;
7       struct    hostent *hp, *gethostbyname();
8       char buf[512];
9       int rc;
10      /* Create socket on which to send. * /
11      sc = socket (AF_INET,SOCK_DGRAM,0);
12      if (sc<0) {
13            perror("opening datagram socket");
14            exit(-1);
15      }
16      /* Construct name of socket to send to. * /
17      server.sin_family = AF_INET;
18      if ( (hp = gethostbyname(argv[1])) == NULL ) {
19            close(sc);
20            fprintf(stderr, "Can't find host %s\n", argv[1]);
21            exit(-1);
22      }
23      bcopy ( hp->h_addr, &(server.sin_addr.s_addr), hp->h_length);
24      server.sin_port = htons(atoi(argv[2]));
25      /* get data from user and send it to the server process * /
26      do {
27            rc=read(0,buf, sizeof(buf));
28            if (sendto(sc, buf, rc, 0, &server, sizeof(server)) <0 ) {
29                  perror("sending datagram message");
30            }
31      }
32      while (rc > 0);
33      printf ("EOF... exit\n");
34      close(sc);
35      exit (0);
36 }
```

Fig. 4b: *IDC* **(client)**

Fig. 4: INET datagram programs

3 Unix Domain

One difference between the *UNIX* and the *INET* domains is the address format. In *UNIX* domain, addresses are file path names, e.g., /tmp/file1. To create a stream socket in the *UNIX* domain we use:

s = socket (AF_*UNIX*, SOCK_STREAM, 0);

and to create a datagram socket we use:

s = socket (AF_*UNIX*, SOCK_DGRAM, 0);

3.1 *UNIX* Virtual-circuit Programs

Fig. 5 shows the code for the server program *UVS* and the client program *UVC*.

The *UVS* process creates a socket ss (lines 13-16), binds it to a *UNIX* file name (lines 18-24). Then it listens to socket ss (line 27), and when a client process initiates a connection, it is accepted into an auxiliary socket s1 (line 28). After that it enters a loop (lines 31-46) to receive (lines 32-33) and display messages sent by the client process (lines 34-37) until an `EOF` is received. Upon receiving `EOF`, it closes the socket and deletes the associated file (lines 41-42).

The client process *UVC* in Fig. 5b creates socket sc (lines 8-12), and forms the server name (lines 14-15). Contrast this with the *INET* program *IVC*, where the server_name is not known until execution time and is formed from the command line arguements: argv[1] and argv[2]. After connecting to the server process, it repeatedly accepts the user input and sends it to the server process until `CTRL-D` is typed.

To test the interaction between *UVS* and *UVC*, we may use two terminals (connected to the same host), or create two windows on one terminal as described before. In invoking the client process *UVC*, we provide no arguments, since the host name is the same for both the client and the server and the socket is bound to a path name known to the client process in priori.

```
0 #include "def"
1 main()
2 {
3       struct      sockaddr_un name;
4       struct  sockaddr from;
5       char      buf[1024];
6       int         cc;
7       int       fromlen;
8       int         ss;
9       int         s1;
10      char       *pathname = "/tmp/unxsock";
11      /* Create socket from which to read. */
12      ss = socket (AF_UNIX,SOCK_STREAM,0);
13      if ( ss<0 ) {
14            perror("opening virtual circuit socket");
15            exit(-1);
16      };
17      /* form name and bind it to ss */
18      name.sun_family = AF_UNIX;
19      strcpy( name.sun_path, pathname);
20      if ( bind( ss, &name, sizeof(struct sockaddr_un)-1 ) ) {
21            close(ss);
22            perror("binding name to ss");
23            exit(-1);
24      }
25      /* listen for a connection from the client process uvc */
26      printf("... waiting for connection ...\n");
27      listen(ss,1);
28      s1     = accept(ss, 0, 0);
29      printf ("connected to client\n");
30      printf ("... waiting for messages ... \n");
31      do {
32            if ( (cc=recv(s1,buf,sizeof(buf),0) ) < 0 )
33                  perror("receiving virtual circuit message");
34            if (cc > 0) {
35                  buf[cc] = NULL;
36                  printf("message received: %s\n", buf);
37            }
38            else {
39                  printf("message received: EOF ");
40                  printf("... disconnect\n");
41                  close (ss);
42                  unlink(pathname);
43                  exit(0);
44            }
45      }
46      while (TRUE);
47 }
```

Fig. 5a: *UVS* (server)

```
0 #include "def"
1 main()
2 {
3       int       sc;
4       struct       sockaddr_un name;
5       char buf[512];
6       int rc;
7       /* Create socket on which to send. */
8       sc = socket (AF_UNIX,SOCK_STREAM,0);
9       if (sc<0) {
10            perror("opening virtual circuit socket");
11            exit(-1);
12      }
13      /* Construct name of socket to send to. */
14      name.sun_family = AF_UNIX;
15      strcpy( name.sun_path, "/tmp/unxsock");
16      if ( connect(sc, &name, sizeof(struct sockaddr_un)-1 ) < 0 ) {
17            close(sc);
18            perror("connecting stream socket");
19            exit(0);
20      }
21      printf("Connected to the server\n");
22      printf("... type any message, to finish type CTRL-D...\n");
23      /* Send message. */
24      do {
25            rc=read(0,buf, sizeof(buf));
26            if (send(sc, buf, rc, 0) <0 )
27                  perror("sending virtual circuit message");
28      }
29      while (rc > 0);
30      printf ("EOF... disconnect\n");
31      close(sc);
32      exit (0);
33 }
```

Fig. 5b: *UVC* (client)

Fig. 5: UNIX Virtual-circuit programs

3.2 *UNIX* Datagram Programs

To experiment with datagrams in the *UNIX* domain, see Fig. 6 for the server program *UDS* and the client program *UDC*. The server process *UDS* creates a socket (lines 11-15), binds it to name ((lines 17-23) , and without waiting for connections, enters a loop (lines 26-41) where it receives the messages sent by the client process and displays it until an EOF is received.

The client process in Fig. 6b creates a socket (lines 8-12) and sends messages with the server name attached to each message(line 20). To test the interaction between these two programs we follow the same procedure for testing the virtual circuit programs in the *UNIX* domain.

4 Input Multiplexing

If a server process is connected concurrently to more than one client process, it needs to multiplex the input received from all of the clients. In this section we present an example in the *INET* domain.

Fig. 7 shows the code for a server program called *IV_SEL* which is essentially the *IVS* program except that it serves two clients instead of just one. The program uses the select system call to monitor the arrival of data from any of the clients. As usual, the program starts by creating a socket (lines 9-13), binds it to a port (lines 15-20), announces the selected port (lines 22-27) and listens for connections (line 30). It accepts the first connection on socket s1 (line 31) and the second connection on socket s2 (line 34). Fig. 1 illustrates the relationship between the server and its two clients.

To monitor the two sockets s1 and s2 for the arrival of data, we use a bit mask called `read_template` (line 3). The mask is cleared using the FD_ZERO macro (line 40) and we add s1 and s2 to the mask using the FD_SET macro (line 41-42). The select call:

```
nb = select (FD_SETSIZE, &read_template, 0, 0, &wait);
```

```
0 #include "def"
1 main()
2 {
3       struct      sockaddr_un  sockname;
4       char        buf[1024];
5       int         cc;
6       struct      sockaddr  from;
7       int         fromlen;
8       int         ss;
9       char        *pathname = "/tmp/unxsock";
10      /* Create socket from which to read. */
11      ss = socket (AF_UNIX,SOCK_DGRAM,0);
12      if ( ss<0 ) {
13            perror("opening datagram socket");
14            exit(-1);
15      };
16      /* form sockname and bind it to ss */
17      sockname.sun_family = AF_UNIX;
18      strcpy( sockname.sun_path, pathname);
19      if ( bind( ss, &sockname, sizeof(struct sockaddr_un) -1) ) {
20            close(ss);
21            perror("binding sockname to ss");
22            exit(-1);
23      }
24      /* Read from the socket */
25      printf ("... waiting for messages ... \n");
26      do {
27            if ( (cc=recv(ss,buf,sizeof(buf),0) ) < 0 )
28                  perror("receiving virtual circuit message");
29            if (cc > 0) {
30                  buf[cc] = NULL;
31                  printf("message received: %s\n", buf);
32            }
33            else {
34                  printf("message received: EOF ");
35                  printf("... exit\n");
36                  close (ss);
37                  unlink(pathname);
38                  exit(0);
39            }
40      }
41      while (TRUE);
42 }
```

Fig. 6a: *UDS* **(server)**

```
0 #include "def"
1 main()
2 {
3       int       sc;
4       struct    sockaddr_un name;
5       char buf[512];
6       int rc;
7       /* Create socket on which to send. */
8       sc = socket (AF_UNIX,SOCK_DGRAM,0);
9       if (sc<0) {
10            perror("opening datagram socket");
11            exit(-1);
12      }
13      /* Construct name of socket to send to. */
14      name.sun_family = AF_UNIX;
15      strcpy( name.sun_path, "/tmp/unxsock" );
16      printf("... type any message, to finish type CTRL-D...\n");
17      /* Send message. */
18      do {
19            rc=read(0,buf, sizeof(buf));
20            if (sendto(sc, buf, rc, 0, &name,
21                                    sizeof(struct sockaddr_un)-1) <0 )
22                  perror("sending datagram message");
23      }
24      while (rc > 0);
25      printf ("EOF... exit\n");
26      close(sc);
27      exit (0);
28 }
```

Fig. 6b: *UDC* **(client)**

Fig. 6: UNIX Datagram programs

is blocked for an amount of time equal to the value of the variable wait (in Fig. 7, lines 37-38, the value is set to 1 second). When the select call returns, then either data has arrived on one of the sockets, the wait time-out has been exceeded, or an error has occurred. If *nb*, the value returned by *select*, is greater than zero then new data has arrived on at least one of the multiplexed sockets. We use the FD_ISSET macro to test whether there is data available at a particular socket (line 49 and 62). For example, if socket s1 has data, it is received (lines 50-51) and displayed (lines 52-55).

When a client terminates the conversation with the server by sending an EOF, the corresponding socket should not be added to the bit mask. This is achieved by setting a flag (e.g., eofl in line 59) when EOF is received. When a flag is set, the corresponding socket is not monitored any more (e.g., line 41). When all clients have sent an EOF, the server terminates (lines 77).

It should be noted that if the value of wait is 0, the select statement will return immediately, i.e., it behaves as if it is "polling" the sockets. On the other hand if &wait is replaced with the constant NULL, then the call is blocked indefinitely until data arrives on one of the monitored sockets.

To test the *IV_SEL* program, we may use three different terminals (alternatively, we may use three windows of one terminal) connected to either one, two or three machines. Assume that we have access to three machines named: "ncsu", "mcnc" and "unc" and we are using three windows of one terminal as shown in Fig. 8. We use the top window to run *IV_SEL* process on "ncsu", the middle window to run a client process *IVC* on the "mcnc" machine and the bottom window to run another client process *IVC* on "unc".

We should notice that any message typed on the middle or the bottom window will instantly appear at the top window. To terminate a client type CTRL-D in the corresponding window. The *IV-SEL* program terminates when all the clients have sent EOF messages (lines 78-81). The test may be conducted using other combinations of terminals, windows and machines.

```
0 #include "def"
1 main()
2 {
3       fd_set read_template;
4       struct timeval wait;
5       struct      sockaddr_in server;
6       char      buf[102];
7       int      sock, length, s1,s2, nb, cc, ready,eof1,eof2;
8       /* Create socket */
9       sock = socket (AF_INET,SOCK_STREAM,0);
10      if ( sock < 0 ) {
11            perror("opening stream socket");
12            exit(0);
13      }
14      /* Name socket using wildcards */
15      server.sin_family = AF_INET;
16      server.sin_addr.s_addr = INADDR_ANY;
17      server.sin_port = 0;
18      if ( bind (sock,&server,sizeof(server)) ) {
19            perror("binding stream socket");
20      }
21      /* Find out assigned port number and print it out */
22      length = sizeof(server);
23      if ( getsockname (sock,&server,&length) ) {
24            perror("getting socket name");
25            exit(0);
26      }
27      printf("Socket has port #%d\n", ntohs(server.sin_port));
28      printf("...waiting for connection ...\n");
29      /* Start accepting connections */
30      listen (sock, 5);
31      s1= accept(sock,0,0);
32      eof1=0;
33      printf("connected to first client\n");
34      s2 = accept(sock,0,0);
35      eof2=0;
36      printf("connected to second client\n");
37      wait.tv_sec = 1;
38      wait.tv_usec = 0;
39      do {
40            FD_ZERO(&read_template);
41            if (!eof1) FD_SET(s1,&read_template);
42            if (!eof2) FD_SET(s2,&read_template);
43            nb = select(FD_SETSIZE, &read_template, (fd_set *) 0, (fd_set *) 0, &wait);
44            if (nb <0) {
45                  perror("select");
46                  exit(1);
47            }
48            if (nb != 0) {            /* 0 time out */
49                  if(FD_ISSET(s1, &read_template)){
50                        if ( (cc=recv(s1,buf,sizeof(buf),0) ) < 0 )
51                              perror("receiving virtual circuit packet");
52                        if (cc > 0) {
53                              buf[cc] = NULL;
54                              printf("message from s1:\n %s", buf);
55                        }
56                        else {
57                              printf("message from s1: EOF\n ");
58                              close(s1);
59                              eof1=1;
60                        }
61                  }
62                  if(FD_ISSET(s2, &read_template)){
63                        if ( (cc=recv(s2,buf,sizeof(buf),0) ) < 0 )
64                              perror("receiving virtual circuit packet");
65                        if (cc > 0) {
66                              buf[cc] = NULL;
67                              printf("message from s2:\n %s", buf);
68                        }
69                        else {
70                              printf("message from s2: EOF\n ");
71                              close(s2);
72                              eof2=1;
73                        }
74                  }
75            }
76      }
77      while ( !eof1 && !eof2 );
78      printf("...exit..\n");
79      close(s1);
80      close(s2);
81      exit(0);
82 }
```

Fig. 7: IV_SEL program

```
% IVS_SEL
  Socket has port# 1234
... waiting for connection ...
connected to first client
connected to second client
message from s1: Hello, first client
message from s2: Hello, second client
message from s1: EOF
message from s2: EOF
... exit...
```

```
% IVC ncsu 1234
Connected to server
type message(CTRL-D to finish)
Hello, flsrt cllent
EOF... disconnect
```

```
% IVC ncsu 1234
Connected to server
type message(CTRL-D to finish)
Hello, second cllent
EOF... disconnect
```

Fig. 8: Testing input multiplexing from two clients

5 Conclusion

We have surveyed the basic features of the Interprocess Communications facilities available in the popular Berkeley version of *UNIX*. Then we presented a series of simple server/client programs that can be used as a model of writing practical distributed applications over a network of computers each running *4.3BSD* or compatible version of *UNIX*. We also show how to test and experiment with the programs in a variety of configurations. After the user masters the concepts and techniques presented in our programs, we recommend looking at the source code of some widely used utilities such as the remote login program (*rlogin*) and the file transfer program (*ftp*). While the paper focuses on a specific system, the experience and knowledge gained are valuable in dealing with other distributed systems and environments.

Acknowledgements

Thanks are due to Dean Brock, Sheng-Uei Guan, John Menges and J. Nievergelt for their helpful comments. This work was done when the author was visiting the Department of Computer Science at the University of North Carolina - Chapel Hill and was partly supported by ONR under contract N00014-86-K-0680.

REFERENCES

1. H. Abdel-Wahab, Sheng-Uei Guan, and J. Nievergelt, 'Shared Workspaces for Real-Time Collaboration in Distributed Networks: Concepts, Techniques, Problems" Department of Computer Science, University of North Carolina, Chapel-Hill (1987).

2. H. Abdel-Wahab, Sheng-Uei Guan, and J. Nievergelt, 'A Prototype for Remotely Shared Textual Workspaces', Department of Computer Science, University of North Carolina, Chapel-Hill (1987).

3. D. Coffield and D. Shepherd, 'Tutorial guide to UNIX sockets for network communications', *Computer Communications,* 10, (1), 21-29 (1987).

4. B. W. Kernighan, and R. Pike, *The UNIX Programming Environment,* Prentice-Hall, (1984).

5. B. W. Kernighan, and D. M. Ritchie, *The C programming Language,* Prentice-Hall, (1978).

6. S.J. Leffler, R.S. Fabry, W.N. Joy, P. Lapsley, S. Miller, and C. Torek, 'An Advanced 4.3BSD Interprocess Communication Tutorial', Computer System Research Group, Department of Electrical Engineering and Computer Science, University of California, Berkeley (1986).

7. B. M. Leiner, R. Cole, J. Postel, and D. Mills, 'The DARPA Internet Protocol Suite' *IEEE Communications Magazine,* 23, (3), 29-34 (1985)

8. S. Sechrest, 'An Introductory 4.3BSD Interprocess Communication Tutorial', Computer System Research Group, Department of Electrical Engineering and Computer Science, University of California, Berkeley (1986).

END
3-88
DTIC