Lab 7: I/O (Socket Programming)

Yu-Lun Huang

2012-01-02

1 Objective

- Understand basic socket programming and client-server paradigms.

2 Prerequisite

- Read man pages of `socket()`, `read()`, `write()`, etc.

3 Client-Server Model

The client–server programming model partitions a task between the service providers (servers) and service requesters (clients). Generally, clients initiate communication sessions by sending requests to a server which await incoming requests.

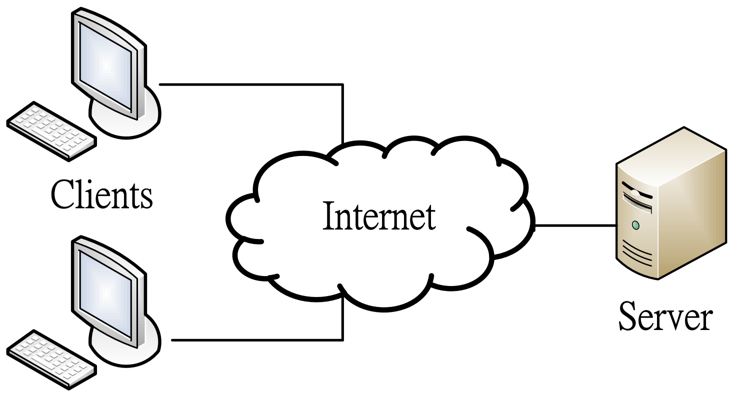

Figure 1: Client-Server Model

If clients and servers communicate over the network, then socket-based programming skills may be used to exchange messages in between. Figure 1 shows a schematic client-server interaction model.

4 Socket Fundamentals

Socket is an interface between an application process and transport layer. With the socket interface, the application process can send and receive messages to or from another application process running on a remote host. Similar to accessing an I/O device, a socket is accessible via a descriptor, as illustrated in Figure 2.

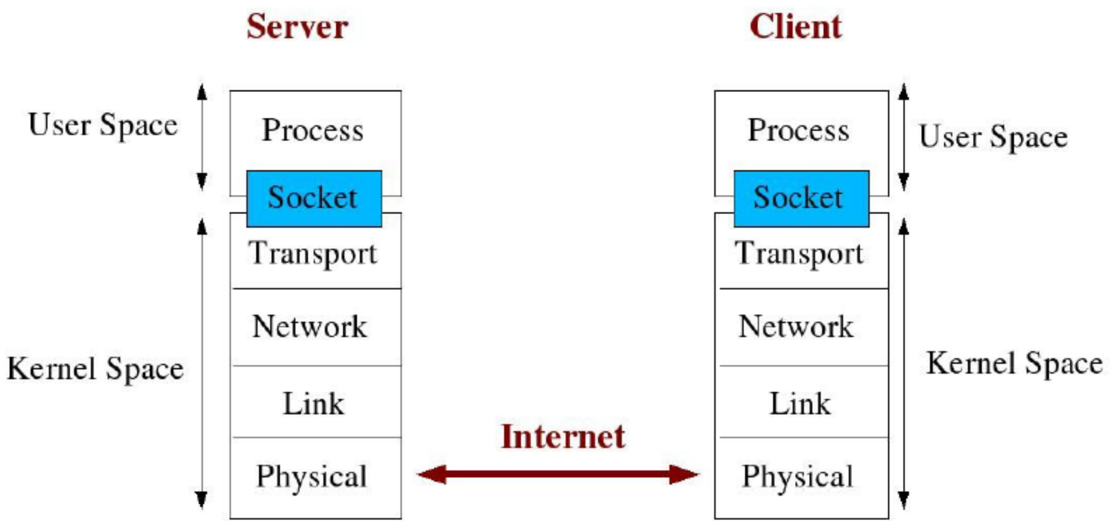

Figure 2: Socket Description.

There are several types of sockets, including internet socket, unix socket, X.25 socket, etc. Internet sockets can be further classified as STREAM socket and DATAGRAM socket, whcih is used for TCP protocol and UDP protocol. The STREAM socket, relying on TCP to provide reliable two-way communication, is used for connection-oriented protocols, as shown in Figure 3.

There are several important system calls for creating sockets for client/server programs:

- `socket()` creates an endpoint for communication and returns a descriptor.
- `connect()` initiates a connection on a socket. Normally used in a client program for sending requests.
- `bind()` binds a name to a socket. Normally, `listen()` is invoked after binding to socket. Used in a server program.
- `listen()` waits for connections on a socket. Used in a server program to wait for incoming requests.
- `accept()` accepts a connection on a socket. Used in a server program.

In a server program, you generally create a socket (using `socket()`, bind to it (`bind()` and listen `listen()` for incoming requests. Once a request is initiated by a client, `accept()` is invoked to accept the connection.

To simplify the sample programs, the common APIs for creating different types of sockets are listed in 'sockop.c':

- `passivesock()` creates a socket and binds to the corresponding socket descriptor to passively accept the incoming connection requests.
- `connectsock()` creates a socket and initiates a connection to the remote host (specified in **struct sockadd_in sin**).

Note that, we also create a new header file (`sockop.h` to include all header files required for socket programming. In `sockop.h`, `sys/socket.h` is necessary for those socket related APIs. In addition, the function prototypes of `passivesock()` and `connectsock()` are also defined in `sockop.h`.

```
/*
 * sockop.h
 */
```

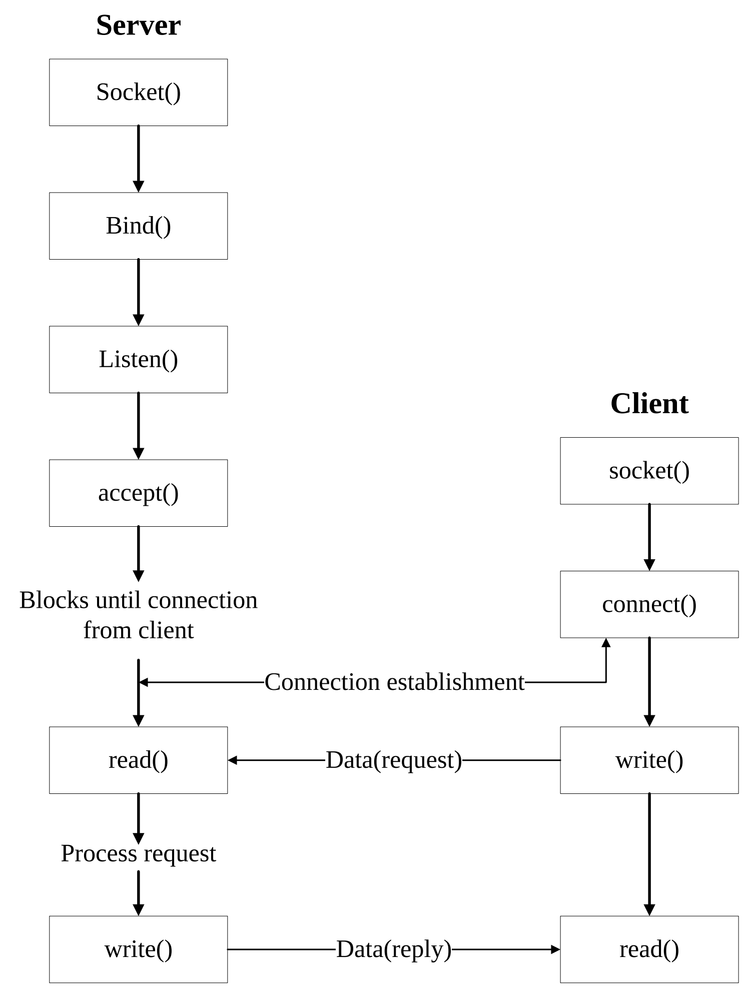

Figure 3: The design flow of Connection-Oriented Protocol.

```

#ifndef  _SOCKOP_H_
#define  _SOCKOP_H_

#include <arpa/inet.h>
#include <errno.h>
#include <netdb.h>
#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <sys/wait.h>

```

```
#define errexit(format,arg...) exit(printf(format,##arg))

/* Create server */
int passivesock(const char *service, const char *transport, int qlen);

/* Connect to server */
int connectsock(const char *host, const char *service, const char *transport);

#endif   /* _SOCKOP_H_ */
```

```
/*
 * sockop.c
 */

#include "sockop.h"

/*
 * passivesock - allocate & bind a server socket using TCP or UDP
 *
 * Arguments:
 *    service   - service associated with the desired port
 *    transport - transport protocol to use ("tcp" or "udp")
 *    qlen      - maximum server request queue length
 */
int passivesock(const char *service, const char *transport, int qlen)
{
   struct servent *pse; /* pointer to service information entry */
   struct sockaddr_in sin; /* an Internet endpoint address */
   int s, type;      /* socket descriptor and socket type */

   memset(&sin, 0, sizeof(sin));
   sin.sin_family = AF_INET;
   sin.sin_addr.s_addr = INADDR_ANY;

   /* Map service name to port number */
   if ((pse = getservbyname(service, transport)))
      sin.sin_port = htons(ntohs((unsigned short)pse->s_port));
   else if ((sin.sin_port = htons((unsigned short)atoi(service))) == 0)
      errexit("Can't␣find␣\"%s\"␣service␣entry\n", service);

   /* Use protocol to choose a socket type */
   if (strcmp(transport, "udp") == 0)
      type = SOCK_DGRAM;
   else
      type = SOCK_STREAM;

   /* Allocate a socket */
   s = socket(PF_INET, type, 0);
   if (s < 0)
      errexit("Can't␣create␣socket:␣%s\n", strerror(errno));

   /* Bind the socket */
   if (bind(s, (struct sockaddr *)&sin, sizeof(sin)) < 0)
      errexit("Can't bind to port %s: %s\n",
            service, strerror(errno));

   if (type == SOCK_STREAM && listen(s, qlen) < 0)
      errexit("Can't listen on port %s: %s\n",
            service, strerror(errno));

   return s;
}

/*
 * connectsock - allocate & connect a socket using TCP or UDP
 *
 * Arguments:
 *    host      - name of host to which connection is desired
 *    service   - service associated with the desired port
 *    transport - name of transport protocol to use ("tcp" or "udp")
 */
int connectsock(const char *host, const char *service, const char *transport)
{
   struct hostent *phe; /* pointer to host information entry */
   struct servent *pse; /* pointer to service information entry */
   struct sockaddr_in sin; /* an Internet endpoint address */
   int s, type;      /* socket descriptor and socket type */

   memset(&sin, 0, sizeof(sin));
   sin.sin_family = AF_INET;

   /* Map service name to port number */
   if ((pse = getservbyname(service, transport)))
      sin.sin_port = pse->s_port;
   else if ((sin.sin_port = htons((unsigned short)atoi(service))) == 0)
      errexit("Can't get \"%s\" service entry\n", service);

   /* Map host name to IP address, allowing for dotted decimal */
   if ((phe = gethostbyname(host)))
      memcpy(&sin.sin_addr, phe->h_addr, phe->h_length);
   else if ((sin.sin_addr.s_addr = inet_addr(host)) == INADDR_NONE)
      errexit("Can't get \"%s\" host entry\n", host);

   /* Use protocol to choose a socket type */
   if (strcmp(transport, "udp") == 0)
      type = SOCK_DGRAM;
   else
      type = SOCK_STREAM;

   /* Allocate a socket */
   s = socket(PF_INET, type, 0);
   if (s < 0)
```

```
        errexit("Can't create socket: %s\n", strerror(errno));

    /* Connect the socket */
    if (connect(s, (struct sockaddr *)&sin, sizeof(sin)) < 0)
        errexit("Can't connect to %s.%s: %s\n", host,
                service, strerror(errno));

    return s;
}
```

The following program shows the client side sample program (echoc.c).

```
/*
 * echoc.c
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#include "sockop.h"

#define BUFSIZE    1024

int main(int argc, char *argv[])
{
    int connfd; /* socket descriptor */
    int n;
    char buf[BUFSIZE];

    if (argc != 4)
        errexit("Usage: %s host_address host_port message\n", argv[0]);

    /* create socket and connect to server */
    connfd = connectsock(argv[1], argv[2], "tcp");

    /* write message to server */
    if ((n = write(connfd, argv[3], strlen(argv[3]))) == -1)
        errexit("Error: write()\n");

    /* read message from the server and print */
    memset(buf, 0, BUFSIZE);
    if ((n = read(connfd, buf, BUFSIZE)) == -1)
        errexit("Error: read()\n");
    printf("%s\n", buf);

    /* close client socket */
    close(connfd);

    return 0;
}
```

The server side sample program (echod.c)

```c
/*
 * echod.c
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#include "sockop.h"

#define BUFSIZE 1024

int main(int argc, char *argv[])
{
   int sockfd, connfd;  /* socket descriptor */
   struct sockaddr_in addr_cln;
   socklen_t sLen = sizeof(addr_cln);
   int n;
   char snd[BUFSIZE], rcv[BUFSIZE];

   if (argc != 2)
      errexit("Usage:␣%s␣port\n", argv[0]);

   /* create socket and bind socket to port */
   sockfd = passivesock(argv[1], "tcp", 10);

   while(1)
   {
      /* waiting for connection */
      connfd = accept(sockfd, (struct sockaddr *) &addr_cln, &sLen);
      if (connfd == -1)
         errexit("Error:␣accept()\n");

      /* read message from client */
      if ((n = read(connfd, rcv, BUFSIZE)) == -1)
         errexit("Error:␣read()\n");

      /* write message back to the client */
      n = sprintf(snd, "Server:␣%.*s", n, rcv);
      if ((n = write(connfd, snd, n)) == -1)
         errexit("Error:␣write()\n");

      /* close client connection */
      close(connfd);
   }

   /* close server socket */
   close(sockfd);

```

```
    return 0;
}
```

The above 'echod.c' is the simplest server code that accepts one client request at a time. To concurrently accept multiple client requests, you can either use the '`select()`' system call or create multiple threads/processes to handle the incoming requests.

5 PXA I/O: LCD Control

- Header files: asm-arm/arch-pxa/lib/creator_pxa270_lcd.h
- Function: ioctl(fd, command, data)
  - Device Name: `/dev/lcd`
  - Data Structure

```
/* Data structure for writing char to LCD screen */
typedef struct lcd_write_info {
    unsigned char Msg[512];        /* the array for saving input */
    unsigned short Count;          /* the number of input char */
    int CursorX, CursorY;          /* X, Y axis of cursor */
} lcd_write_info_t;

/* Data structure for writing a picture to LCD screen */
typedef struct lcd_full_image_info {
    unsigned short data[0x800];    /* the array for saving picture */
} lcd_full_image_info_t;
```

  - Command

```
/* Clear LCD data and move cursor back to the Upper-left corner */
#define LCD_IOCTL_CLEAR     LCD_IO ( 0x0 )

/* Write char to LCD */
#define LCD_IOCTL_WRITE     LCD_IOW( 0x01, lcd_write_info_t )

/* Turn On or Off cursor */
#define LCD_IOCTL_CUR_ON    LCD_IO( 0x02 )
#define LCD_IOCTL_CUR_OFF   LCD_IO( 0x03 )

/* Get and Set the position (X, Y) of cursor */
#define LCD_IOCTL_CUR_GET   LCD_IOR( 0x04, lcd_write_info_t )
#define LCD_IOCTL_CUR_SET   LCD_IOW( 0x05, lcd_write_info_t )

/* Write a picture to LCD */
#define LCD_IOCTL_DRAW_FULL_IMAGE LCD_IOW(0x06, lcd_full_image_info_t)
```

The following is the sample code of LCD control.

```
/*
 * lcd.c -- The sample code to print "Hello World" on LCD screen.
 */
```

```

#include <stdio.h>
#include <sys/fcntl.h>
#include <sys/ioctl.h>
#include <unistd.h>
#include "asm-arm/arch-pxa/lib/creator_pxa270_lcd.h"

int main()
{
   int fd;
   lcd_write_info_t display;  /* struct for saving LCD data */

   /* Open device /dev/lcd */
   if ((fd = open("/dev/lcd",O_RDWR) < 0))
   {
      printf("open /dev/lcd error\n");
      return (-1);
   }

   /* Clear LCD */
   ioctl(fd,LCD_IOCTL_CLEAR,NULL);

   /* Save output string to display data structure */
   display.Count = sprintf((char *) display.Msg, "Hello World\n");
   /* Print out "Hello World" to LCD */
   ioctl(fd, LCD_IOCTL_WRITE, &display);


   /* Get the cursor position */
   ioctl(fd, LCD_IOCTL_CUR_GET, &display);
   printf("The cursor position is at (x,y) = (%d,%d)\n",
         display.CursorX, display.CursorY);

   close(fd);
   return 0;
}
```

6 Reference Books

You can obtain more information about socket programming from the following books:

1. Douglas E. Comer, "Internetworking with TCP/IP Vol.1: Principles, Protocols, and Architecture (4th Edition)," Prentice Hall, 2000.

2. Douglas E. Comer and David L. Stevens, "Internetworking with TCP/IP Vol. II: ANSI C Version: Design, Implementation, and Internals (3rd Edition)," Prentice Hall, 1998.

3. Douglas E. Comer, David L. Stevens and Michael Evangelista, "Internetworking with TCP/IP, Vol. III: Client-Server Programming and Applications, Linux/Posix Sockets Version," Prentice Hall, 2000.

Lab 7 Assignments

Write a client and a server program to enable the embedded platform as a network echo server.

· The client program (running on your virtual machine) shows the echo outputs according to the keypad inputs when connected to the server.

· The server program (running on the development board) handles 1 client at a time and shows the echo message on the LCD. For example, upon pressing keypad key '1', you can show the predefined strings "ECHO: This is the echo outputs for key 1." on the LCD.

· (BONUS) Modify the server program so that it can handle multiple clients concurrently. (Hint: you can use 'multi-process', 'multi-thread' or 'select()' to realize the server and support multiple clients.)