Chapter 10

IPC - IV

Chapter Objectives

To understand concepts of IPC used over the duration of the course.

Objectives

For this chapter, the following are the objectives:

- UDP Sockets,

Slide #10-1

Notes

In this chapter, we examine UDP sockets.

The objective of this chapter is to provide an understanding of concepts on UDP sockets.

Chapter Organization

1. **Objective**: Introduction to
   - UDP sockets,
2. **Description**: This an introduction to IPC mechanisms in Linux.
   This chapter provides an introduction to the concepts used in course.
3. **Concepts Covered in Chapter**:
   - UDP sockets
4. **Prior Knowledge**:
   same as Chapter #1
5. **Teaching & Learning Strategy**:
   Discussion questions are,
   - What are these tools for?
6. **Teaching Format**:
   Theory + Homework Assignments
7. **Study Time**: 120 Minutes (Lecture & Theory)
   + ~45 minutes (Homework Assignments)
8. **Assessment**: Group Homework Assignments
9. **Homework Eval**: Group
10. **Chapter References**:

UDP Sockets

UDP .. overview

- **Connectionless protocol**

UDP socket

```
   int socket (AF_INET, SOCK_DGRAM, PF_UNSPEC)
```

UDP uses
- sendto() and
- recvfrom()

Slide #10-2

Notes:

1. UDP servers do not fork() with every new client.
2. Connectionless Protocol – because clients do not use connect() and accept() to establish a dedicated port to communicate.
3. All clients send their messages to the same port.
4. The benefit of using a connectionless protocol is that a single server process can receive messages from many clients. For managing concurrent access to centrally stored data, UDP allows one process to coordinate and queue all transactions.
5. Sending a message over UDP does not guarantee that the message will be delivered to the destination address.
6. Single UDP messages are also limited in size. The absolute upper limit is 8k.
7. Instead of using read() and write() calls on socket descriptors, UDP sockets use sendto() and recvfrom() for sending and receiving messages.

gethostbyname()
getservbyname()
copy to network format

Programming the UDP client

`socket ()` ... create file descriptor

bind()

•

•

`shutdown()`

`close()`

Slide #10-3

Notes:

1. `socket()`
   `int socket(address_family, sock_stream, protocol_family)`
   1.1. where the `address_family` is typically AF_INET
   1.2. `sock_stream` is SOCK_DGRAM
   1.3. `protocol_ family` can be PF_INET or PF_UNSPEC
2. `bind()`
   The client socket descriptor must be bound to a port, in order for the UDP client program to use the sendto(). The server process, receives the client port number with every message from the client, therefore, the client port number does not need to be known ahead of time. Typically, '0' is passed to bind(), which then picks an available port.

```
#define _POSIX_SOURCE 1

#include <stdio.h>
#include <errno.h>
#include <string.h>

#define  MYSERVER_UDP_PORT_NUMBER 29786

char *Program="udp_client";


int getUDPsocket(int port_number, char *errbuf) {

      char  myhostname[1024];
      struct sockaddr_in dgram_addr[1];
      int         udp_sd;

      char  localbuf[256];

      int   status;

      gethostbyname(myhostname,1024);

      status=InetAddressInit(dgram_addr, myhostname, port_number, errbuf);

      if (status != 0) {
            sprintf(errbuf, "couldn't get inet address: %s", localbuf);
            return status;
      }

      udp_sd = socket(AF_INET, SOCK_DGRAM, PF_UNSPEC);

      if (udp_sd < 0) {
            sprintf(errbuf,"socket() error: %s\n",strerror(errno));
            return -1;
      }

      status=bind(udp_sd,(struct sockaddr *) dgram_addr, sizeof(struct
sockaddr_in));

      if (status==-1) {
            sprintf(errbuf,"bind() error: port# %d:
%s\n",port_number,strerror(errno));

            close(udp_sd);
            return -1;
      }

      return udp_sd;
}

int InetAddressInit(struct sockaddr_in *saddr, char *hostname,
      int portnum, char *errbuf) {

      int nitems;
      int var;

      memset(saddr, 0, sizeof(struct sockaddr_in));
```

```
	sock_addr_ptr->sin_family = AF_INET;
	sock_addr_ptr->sin_port = htons((unsigned short)portnum);

	nitems = sscanf(hostname, "%*d.%*d.%*d.%d", &var);

	if (nitems == 1) {
		unsigned long addr;

		addr=inet_addr(hostname);

		memcpy (&(sock_addr_ptr->sin_addr), &addr, sizeof(struct
in_addr));
	}
	else {

		struct hostent *hostinfo;

		hostinfo=gethostbyname(hostname);

		if (hostinfo == 0) {
			sprintf(errbuf,"gethostbyname() error: %s\n",
strerror(errno);

			return -2;
		}

		if (hostinfo->h_addrtype != AF_INET) {
			sprintf(errbuf,"address type error: %s\n", strerror(errno);

			return -3;
		}

		memcpy (&(sock_addr_ptr->sin_addr), hostinfo->h_addr,
sizeof(struct in_addr));

	}

	return 0;

}

int main(int argc,char **argv) {

	struct sockaddr_in	server_addr[1];
	struct sockaddr_in	reply_addr[1];

	int	reply_addr_len;

	char *remote_host;
	int	sd, bytes_sent, flags;
	char *msgbuf;
	int	msglen, reply_length;
	char	rcvbuf[1024];	// receive buffer
	char	errbuf[1024];	// error buffer
	it	status;

	ProgramName=*argv;

	if (*argv == 0){
		fprintf(stderr, "Err: Need name of Remote Host as argument\n");
		exit(1);
	}

	remote_host=*argv;
	sd=getUDPsocket(0,errbuf);

	if (sd <0) {
		fprintf(stderr, "Err: Couldn't get UDP socket : %s\n", errbuf);
		exit(1);
	}

	status=InetAddressInit(server_addr, remote_host,
MYSERVER_UDP_PORT_NUMBER, errbuf);

	if (status!=0) {
		fprintf(stderr, "Err: Couldn't get inet address : %s\n", errbuf);
		exit(1);
	}

	msgbuf="This is a test!";
	msglen=strlen(msgbuf);
	flags=0x0;

	bytes_sent=sendto(sd,msgbuf,msglen,flags,
		(struct sockaddr *) server_addr, sizeof(struct sockaddr_in));

	if (bytes_sent < msglen) {
		fprintf(stderr,"sendto() err: %s\n",
			(bytes_sent < 0) ? strerror(errno) : "too few bytes");

		exit(1);

	}

	reply_addr_len = sizeof(reply_addr);

	reply_length = recvfrom(sd, recv_buf, sizeof(recv_buf), flags,
		(struct sockaddr *) reply_addr, &reply_addr_len);

	if (reply_length == -1) {
		fprintf(stderr,"recvfrom() err: %s\n", strerror(errno) );

		exit(1);
	}

	recv_buf[reply_length] = 0;

	printf("\nReceived %d bytes from '%s':\n\t%s\n",
		ProgramName,reply_length, remote_host, recv_buf);

	exit(0);
}
```

Programming the server

`socket ()` ... create file descriptor

- `bind()` => get `hostbyname()`, get `servbyname()`, copy to network format

`shutdown()`

`close()`

Slide #10-4

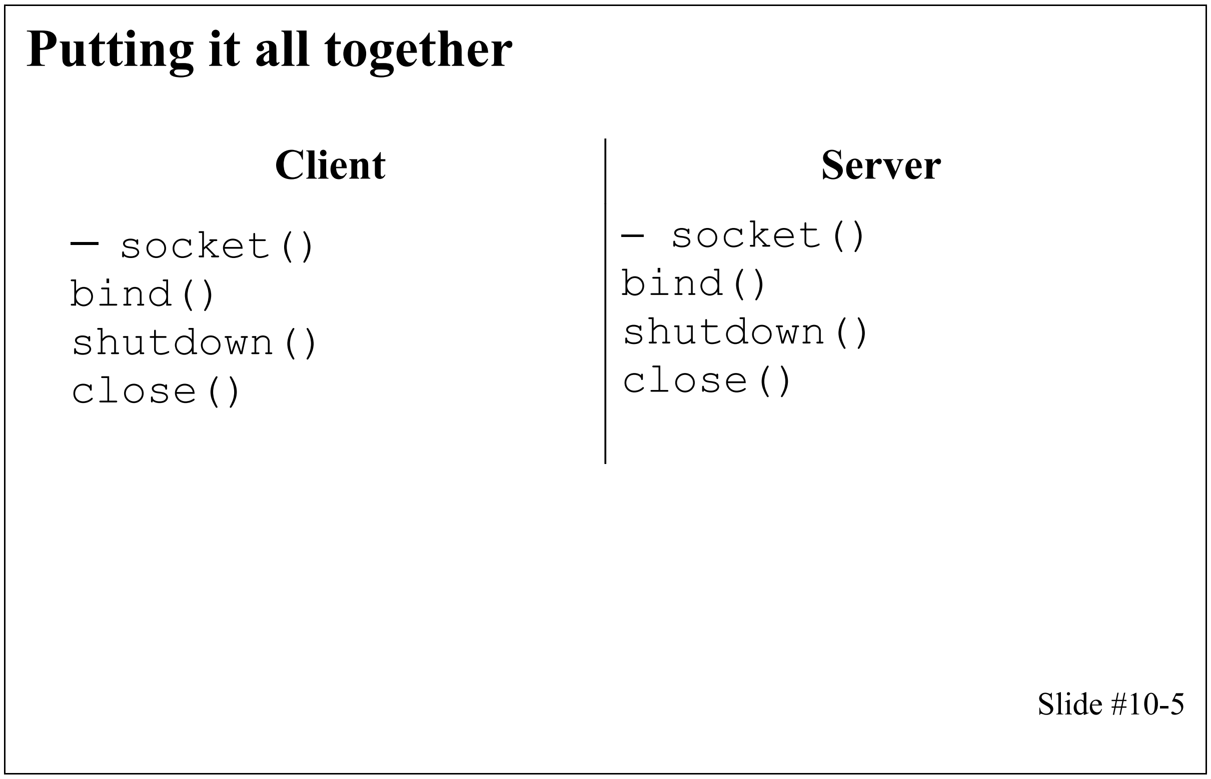
Notes:

1. `socket()`
2. `bind()`
3. `shutdown()`
4. `close()`

```
#define _POSIX_SOURCE 1

#include <stdio.h>
#include <errno.h>
#include <string.h>

#define  MYSERVER_UDP_PORT_NUMBER 29786

char *Program="udp_server";

int getUDPsocket(int port_number, char *errbuf) {

      char  myhostname[1024];
      struct sockaddr_in dgram_addr[1];
      int         udp_sd;

      char  localbuf[256];

      int   status;

      gethostbyname(myhostname,1024);
```

```
	status=InetAddressInit(dgram_addr, myhostname, port_number, errbuf);

	if (status != 0) {
		sprintf(errbuf, "couldn't get inet address: %s", localbuf);
		return status;
	}

	udp_sd = socket(AF_INET, SOCK_DGRAM, PF_UNSPEC);

	if (udp_sd < 0) {
		sprintf(errbuf,"socket() error: %s\n",strerror(errno));
		return -1;
	}

	status=bind(udp_sd,(struct sockaddr *) dgram_addr, sizeof(struct 
sockaddr_in));

	if (status==-1) {
		sprintf(errbuf,"bind() error: port# %d: 
%s\n",port_number,strerror(errno));

		close(udp_sd);
		return -1;
	}

	return udp_sd;
}

int InetAddressInit(struct sockaddr_in *saddr, char *hostname,
	int portnum, char *errbuf) {

	int nitems;
	int var;

	memset(saddr, 0, sizeof(struct sockaddr_in));

	sock_addr_ptr->sin_family = AF_INET;
	sock_addr_ptr->sin_port = htons((unsigned short)portnum);

	nitems = sscanf(hostname, "%*d.%*d.%*d.%d", &var);

	if (nitems == 1) {
		unsigned long addr;

		addr=inet_addr(hostname);

		memcpy (&(sock_addr_ptr->sin_addr), &addr, sizeof(struct 
in_addr));
	}
	else {

		struct hostent *hostinfo;

		hostinfo=gethostbyname(hostname);

		if (hostinfo == 0) {
			sprintf(errbuf,"gethostbyname() error: %s\n", 
strerror(errno);
```

```
                return -2;
            }

            if (hostinfo->h_addrtype != AF_INET) {
                sprintf(errbuf,"address type error: %s\n", strerror(errno));

                return -3;
            }

            memcpy (&(sock_addr_ptr->sin_addr), hostinfo->h_addr,
sizeof(struct in_addr));

      }

      return 0;

}

int main(int argc,char **argv) {


      char errbuf[10000];

      int   server_sd;

      struct sockaddr_in      reply_addr[1];
      int   reply_addr_length;
      char  address_str;

      char  recv_buf[10000];
      int         msglen, flags;

      char *      msgbuf;

      int         reply_length, bytes_sent;

      ProgramName=*argv;

      server_sd=getUDPsocket(MYSERVER_UDP_PORT_NUMBER,errbuf);

      if (server_sd <0) {
            fprintf(stderr, "Err: Couldn't get UDP socket : %s\n", errbuf);
            exit(1);
      }

      while (1) {

            flags=0;

            reply_addr_length = sizeof(struct sockaddr_in);

            msglen = recvfrom(sd, recv_buf, sizeof(recv_buf), flags,
                  (struct sockaddr *) reply_addr, &reply_addr_len);

            if (msglen == -1) {
                  fprintf(stderr,"recvfrom() err: %s\n", strerror(errno) );

                  exit(1);
            }

            recv_buf[mesg_length]=0;

            mesg_buf = "This is a Test, too!";
            reply_len=strlen(mesg_buf);


            bytes_sent=sendto(sd,msgbuf,msglen,flags,
                  (struct sockaddr *) server_addr, sizeof(struct
sockaddr_in));

            if (bytes_sent < msglen) {
                  fprintf(stderr,"sendto() err: %s\n",
                        (bytes_sent < 0) ? strerror(errno) : "too few bytes");

                  exit(1);

            }
      }

      exit(0);
}
```



Putting it all together

Client

– socket()
bind()
shutdown()
close()

Server

– socket()
bind()
shutdown()
close()

Slide #10-5

Makefile

```
all:  tcp_server tcp_client gendata myserver.dat

tcp_server: tcp_server.c
      gcc -o tcp_server -lsocket -lnsl tcp_server.c

tcp_client: tcp_client.c
      gcc -o tcp_client -lsocket -lnsl tcp_client.c

myserver.dat gendata: gendata.c
      -rm -f myserver.dat myserver.txt
      gcc -o gendata gendata.c
      ./gendata
      chmod 644 myserver.dat
      ls -l myserver.dat

clean:
      rm -f myserver.dat tcp_server tcp_client tcp_sample.tar
tcp_sample.tar.gz gendata

move: clean
      tar -cvf tcp_sample.tar tcp_server.c tcp_client.c gendata.c Makefile
      gzip tcp_sample.tar
```

gendata.c

```c
#define MAXLINES 3

#include <errno.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

struct Example {
      int linenum;
      char data[1024];
};

typedef struct Example Example;

int main() {

      Example ExampleData[1];

      int wfd, i=1, j=0;

      umask(777);

      if ((wfd=open("myserver.dat",O_CREAT|O_TRUNC|O_WRONLY,0x644)) < 0) {
            fprintf(stderr,"open() Failure: %s\n", strerror(errno));
            return -1;
      }

      while (i<=MAXLINES) {
            ExampleData->linenum=i;
            ExampleData->data[0]=0;
            sprintf(ExampleData->data,"This is Line '%d'", i);

            j=sizeof(struct Example);
            if ((write(wfd,ExampleData, j)) < j) {
                  fprintf(stderr,"write() Failure: %s\n", strerror(errno));
                  return -1;
            }
            i++;
      }

      fprintf(stderr,"Wrote %d lines.\n",i);

      return 0;
}
```

Chapter 10
IPC - IV
Assignment Questions

Questions:

10.1 Modify “mysh” (from chapter 9) to implement the ‘get’ and ‘put’ commands to get/put a single file, using socket communication

- 10.1.1 use TCP style sockets for communicating commands between client and server, and
- 10.1.2 use UDP style sockets for bulk transfers of files across the network
- 10.1.3 Implement the “-s full” option to communicate between the requester and the server.
- 10.1.4 It should also
    - 10.1.4.1 Be in an rpm/apt package to install client and server
    - 10.1.4.2 Include static/shared libraries, and man pages
    - 10.1.4.3 support mywc, mycat and myls, also lmywc, lmycat and lmyls