1. Explain lack of flow control with UDP

#include "unp.h"

#define NDG 2000 /\* datagrams to send \*/

#define DGLEN 1400 /\* length of each datagram \*/

void

dg\_cli(FILE \*fp, int sockfd, const SA \*pservaddr, socklen\_t servlen)

{

int i;

char sendline[DGLEN];

for (i = 0; i < NDG; i++) {

Sendto(sockfd, sendline, DGLEN, 0, pservaddr, servlen);

}

}

Dg\_cli function that writes fixed no of datagrams to server

#include "unp.h"

2 static void recvfrom\_int(int);

3 static int count;

4 void

5 dg\_echo(int sockfd, SA \*pcliaddr, socklen\_t clilen)

6 {

7 socklen\_t len;

8 char mesg[MAXLINE];

9 Signal(SIGINT, recvfrom\_int);

10 for ( ; ; ) {

11 len = clilen;

12 Recvfrom(sockfd, mesg, MAXLINE, 0, pcliaddr, &len);

13 count++;

14 }

15 }

Dg\_echo function that counts received datagrams

The interface’s buffers were full or they could have been discarded by the sending host. The counter “dropped due to full socket buffers” indicates how many datagram were received by UDP but were discarded because the receiving socket’s receiving queue was full. The number of datagrams received by the server in this example is non deterministic. It depends on many factors, such as network load, processing load on client host, and in server host.

Solution: fast server, slow client; increase size of socket receive buffer

1. Important functions of udp echo server

Table

Description automatically generated with low confidence

Main:

1 #include "unp.h"

2 int

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

5 int sockfd;

6 struct sockaddr\_in servaddr, cliaddr;

7 sockfd = Socket(AF\_INET, SOCK\_DGRAM, 0);

8 bzero(&servaddr, sizeof(servaddr));

9 servaddr.sin\_family = AF\_INET;

10 servaddr.sin\_addr.s\_addr = htonl(INADDR\_ANY);

11 servaddr.sin\_port = htons(SERV\_PORT);

12 Bind(sockfd, (SA \*) &servaddr, sizeof(servaddr));

13 dg\_echo(sockfd, (SA \*) &cliaddr, sizeof(cliaddr));

14 }

We create a UDP socket by specifying the second argument to socket as SOCK\_DGRAM (a datagram socket in the IPv4 protocol).

Dg\_echo:

1 #include "unp.h"

2 void

3 dg\_echo(int sockfd, SA \*pcliaddr, socklen\_t clilen){

5 int n;

6 socklen\_t len;

7 char mesg[MAXLINE];

8 for ( ; ; ) {

9 len = clilen;

10 n = Recvfrom(sockfd, mesg, MAXLINE, 0, pcliaddr, &len);

11 Sendto(sockfd, mesg, n, 0, pcliaddr, len);

12 }}

This function is a simple loop that reads the next datagram arriving at the server's port using recvfrom and sends it back using sendto. Despite the simplicity of this function, there are numerous details to consider. First, this function never terminates. Since UDP is a connectionless protocol, there is nothing like an EOF as we have with TCP.

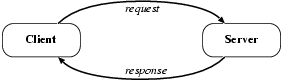
Next, this function provides an iterative server, not a concurrent server as we had with TCP. There is no call to fork, so a single server process handles any and all clients. In general, most TCP servers are concurrent and most UDP servers are iterative.

1. Steps involved in building udp echo client server application

Client:

* In the UDP Echo client a socket is created.
* Then we bind the socket.
* After the binding is succesful , we send messages input from the user and display the data received from the server using sendto() and recvfrom() functions.

Server:



* In the UDP Echo server , we create a socket and bind to a advertized port number.
* Then an infinite loop is started to process the client requests for connections.
* The process receives data from the client using recvfrom () function and echoes the same data using the sendto() function.
* Please note that this server is capable of handles multiple clients automatically as UDP is a datagram based protocol hence no exclusive connection is required to a client in this case.

1. Dg\_echo function of udp echo server

#include "unp.h"

Void dg\_echo(int sockfd, SA \*pcliaddr, socklen\_t clilen){

int n;

socklen\_t len;

char mesg[MAXLINE];

for ( ; ; ) {

len = clilen;

n = Recvfrom(sockfd, mesg, MAXLINE, 0, pcliaddr, &len);

Sendto(sockfd, mesg, n, 0, pcliaddr, len);

}}

This function is a simple loop that reads the next datagram arriving at the server's port using recvfrom and sends it back using sendto.

Despite the simplicity of this function, there are numerous details to consider. First, this function never terminates. Since UDP is a connectionless protocol, there is nothing like an EOF as we have with TCP.

Next, this function provides an iterative server, not a concurrent server as we had with TCP. There is no call to fork, so a single server process handles any and all clients. In general, most TCP servers are concurrent and most UDP servers are iterative.

There is implied queuing taking place in the UDP layer for this socket. Indeed, each UDP socket has a receive buffer and each datagram that arrives for this socket is placed in that socket receive buffer. When the process calls recvfrom, the next datagram from the buffer is returned to the process in a first-in, first-out (FIFO) order. This way, if multiple datagrams arrive for the socket before the process can read what's already queued for the socket, the arriving datagrams are just added to the socket receive buffer. But, this buffer has a limited size.

1. Unconnected and connected udp sockets

This does not result in anything like a TCP connection: there is no three-way handshake. Instead, the kernel just records the IP address and port number of the peer.

With a connect UDP socket three things change:

* We can no long specify the destination IP address and port for an o/p operation that is, we do not use sendto(), but use write or send instead.
* We do not use recvfrom(), but read or receive instead
* Asynchronous errors are returned to the process for a connected udp socket

Diagram

Description automatically generated

Datagrams arriving from any other IP address or port are not passed to the connected socket because either the source IP address or source UDP port does not match the protocol address to which the socket is connected.



1. Identify resulting changes with connected udp socket compared with other

With a connected UDP socket, three things change, compared to the default unconnected UDP socket:

1. We can no longer specify the destination IP address and port for an output

operation. That is, we do not use sendto, but write or send instead. Anything

written to a connected UDP socket is automatically sent to the protocol address

(e.g., IP address and port) specified by connect.

Similar to TCP, we can call sendto for a connected UDP socket, but we cannot

specify a destination address. The fifth argument to sendto (the pointer to the

socket address structure) must be a null pointer, and the sixth argument (the size

of the socket address structure) should be 0. The POSIX specification states that

when the fifth argument is a null pointer, the sixth argument is ignored.

2. We do not need to use recvfrom to learn the sender of a datagram, but read, recv,

or recvmsg instead. The only datagrams returned by the kernel for an input

operation on a connected UDP socket are those arriving from the protocol address

specified in connect. Datagrams destined to the connected UDP socket's local

protocol address (e.g., IP address and port) but arriving from a protocol address

other than the one to which the socket was connected are not passed to the

connected socket. This limits a connected UDP socket to exchanging datagrams with

one and only one peer.

Technically, a connected UDP socket exchanges datagrams with only one IP address,

because it is possible to connect to a multicast or broadcast address.

3. Asynchronous errors are returned to the process for connected UDP sockets.

The corollary, as we previously described, is that unconnected UDP sockets do not

receive asynchronous errors.

1. Different interface models in sctp protocol

The one to one style:

The one-to-one style was developed to ease the porting of existing TCP applications to SCTP. There are some differences one should be aware of, especially when porting existing TCP applications to SCTP using this style:

* Any socket options must be converted to the SCTP equivalent. Two commonly found options are TCP\_NODELAY and TCP\_MAXSEG.
* SCTP preserves message boundaries; thus, application-layer message boundaries are not required. For example, an application protocol based on TCP might do a write() system call to write a two-byte message length field, x, followed by a write() system call that writes x bytes of data. However, if this is done with SCTP, the receiving SCTP will receive two separate messages
* Some TCP applications use a half-close to signal the end of input to the other side. To port such applications to SCTP, the application-layer protocol will need to be rewritten so that the application signals the end of input in the application data stream.
* The send function can be used in the normal fashion.

Diagram

Description automatically generated

The one to many style:

The one-to-many style provides an application writer the ability to write a server without managing a large number of socket descriptors.

* When the client closes the association, the server side will automatically close as well, thus removing any state for the association inside the kernel.
* Using the one-to-many style is the only method that can be used to cause data to be piggybacked on the third or fourth packet of the four-way handshake
* Any sendto, sendmsg, or sctp\_sendmsg to an address for which an association does not yet exist will cause an active open to be attempted, thus creating a new association with that address.
* The user must use the sendto, sendmsg, or sctp\_sendmsg functions, and may not use the send or write function.
* Anytime one of the send functions is called, the primary destination address that was chosen by the system at association initiation time will be used unless the MSG\_ADDR\_OVER flag is set by the caller in a supplied sctp\_sndrcvinfo structure.
* Association events may be enabled, so if an application doesn’t wish to receive these events, it should disable them explicitly using the SCTP\_EVENTS socket option.

Diagram

Description automatically generated

1. SCTP association establishment

The server must be prepared to accept an incoming association. This preparation is normally done by calling socket, bind, and listen.

The client issues an active open by calling connect or by sending a message. This causes the client SCTP to send an INIT message to tell the server the client's list of IP addresses, number of outbound streams the client is requesting, and number of inbound streams the client can support.

The server acknowledges the client's INIT message with an INIT-ACK message, which contains the server's list of IP addresses, number of outbound streams the server is requesting, number of inbound streams the server can support, and a state cookie.

The client echos the server's state cookie with a COOKIE-ECHO message.

The server acknowledges that the cookie was correct and that the association was established with a COOKIE-ACK message.

The minimum number of packets required for this exchange is four; hence, this process is called SCTP's four-way handshake.

Diagram

Description automatically generated

1. Steps that allow ipv4 tcp client and ipv6 server

The steps are as follows:

* IPV6 Server starts, creates a listening IPV6 socket and it binds wildcard address to the socket
* IPV4 client calls gethostbyname and finds an A record
* The Client calls connect and client’s host sends an IPV4 SYN to server
* The Server host receives IPV4 SYN directed to IPV6 socket, sets a flag indicating that this connection is using IPV4 mapped IPV6 address and responds with IPV4 SYN/ACK
* When the server host sends IPV4-mapped-IPV6 address, IP Stack generates IPV4 datagram to IPV4 address
* Dual Stack handles all the finer details and the Server is unaware that it is communicating with IPV4 client



1. Syslogd daemon and syslog function

Syslogd daemon:

Unix systems normally start a daemon named syslogd from one of the system

initializations scripts, and it runs as long as the system is up. Berkeley-derived

implementations of syslogd perform the following actions on startup:

1. The configuration file is read, specifying what to do with each type of log message that the daemon can receive. These messages can be appended to a file, written to a specific user or forwarded to the syslogd daemon on another host.

2. A Unix domain socket is created and bound to the pathname /var/run/

3. A UDP socket is created and bound to port 514 (the syslog service).

4. The pathname /dev/klog is opened. Any error messages from within the kernel

appear as input on this device.

Syslog function:

Syslog stands for System Logging Protocol and is a standard protocol used to send system log or event messages to a specific server, called a syslogserver.

It is primarily used to collect various device logs from several different machines in a central location for monitoring and review.  
  
void syslog(int priorityLevel, char \*msg);

Static void

Err\_doit(int errnoflag, int level, const char\*fmt, va\_list ap)

{

Int errno\_save, n;

Char buf[MAXLINE];

Errno\_save = errno;

If(errnoflag)

Snprintf(buf+n, sizeof(buf)-n, “:%s”, strerror(errno\_save));

Strcat(buf, “\n”);

If(daemon\_proc) {

Syslog(level, buf);

} else {

Fflush(stdout);

Fputs(buf, stdout);

Fflush(stdout);

}

Return;

}