Network Programming IA 2

1. Explain the lack of flow control with UDP.

There is no flow control, and hence no window mechanism. The receiver may overflow with incoming messages. The lack of flow control means that the process using UDP should provide for this service if needed.

- To examine the effect of UDP not having any flow control, we modify our `dg_cli` function to send a fixed number of datagrams.
- Next, we modify the server (`dg_echo`) to receive datagrams and count the number received. This server no longer echoes datagrams back to the client.
- When we terminate the server with our terminal interrupt key, it prints the number of received datagrams and terminates.

`dg_cli` function that writes a fixed number of datagrams to the server:

```
#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_echo` function that counts received datagrams:

```
#include "unp.h"
static void recvfrom_int(int);
static int count;
void dg_echo(int sockfd, SA *pcliaddr, socklen_t clilen)
{
  socklen_t len;
  char mesg[MAXLINE];
  Signal(SIGINT, recvfrom_int);
  for ( ; ; ) {
    len = clilen;
    Recvfrom(sockfd, mesg, MAXLINE, 0, pcliaddr, &len);
```

```
    count++;
  }
}

static void
recvfrom_int(int signo)
{
  printf("\nreceived %d datagrams \n", count);
  exit(0);
}
```

The client sent 2,000 datagrams, but the server application received only 30 of these, for a 98% loss rate. There is no indication whatsoever to the server application or to the client application that these datagrams were lost. This shows that UDP has no flow control and is unreliable.

2. Important functions of the UDP echo server.

Figure 8.2. Simple echo client/server using UDP.

UDP Echo Server `main` Function

```
#include "unp.h"
int main(int argc, char **argv)
{
  int sockfd;
  struct sockaddr_in servaddr, cliaddr;
  sockfd = Socket(AF_INET, SOCK_DGRAM, 0);
  bzero(&servaddr, sizeof(servaddr));
  servaddr.sin_family = AF_INET;
  servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
  servaddr.sin_port = htons(SERV_PORT);
  Bind(sockfd, (SA *) &servaddr, sizeof(servaddr));
  dg_echo(sockfd, (SA *) &cliaddr, sizeof(cliaddr));
}
```

In this UDP socket is created by giving SOCK_DGRAM. The address for the bind is given as INADDR_ANY for the multihomed server. And the const SERV_PORT is a well-known port.

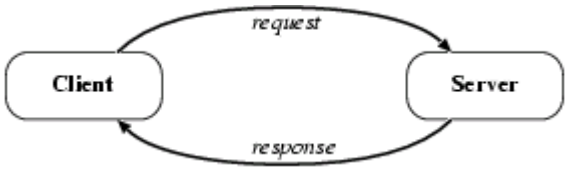
UDP Echo Server `dg_echo` Function

This function is called to perform server processing.

```
#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);
  }
}
```

- The function never terminates (`exit(0)` is not called) as it is a connection-less protocol.
- The main function is an iterative server and not concurrent. There is no call to fork(), so a single server process handles any and all clients.

3. Steps involved in building an echo Client-Server application using UDP.



Client

- In the UDP Echo Client, a socket a created
- Then we bind the socket
- After the binding is successful, we send messages input from the user
- Wait until a response from the server is received
- Process the reply 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 an advertised port number
- Then an infinite loop is started to process the client requests for connections
- The process receives data from the client using the `recvfrom()` function and echoes the same data using the `sendto()` function.
- This type of server is capable of handling multiple clients automatically as UDP is a datagram-based protocol. Hence no exclusive connection is required to a client in this case.

4. The `dg_echo` function in C that is used in the UDP echo server application.

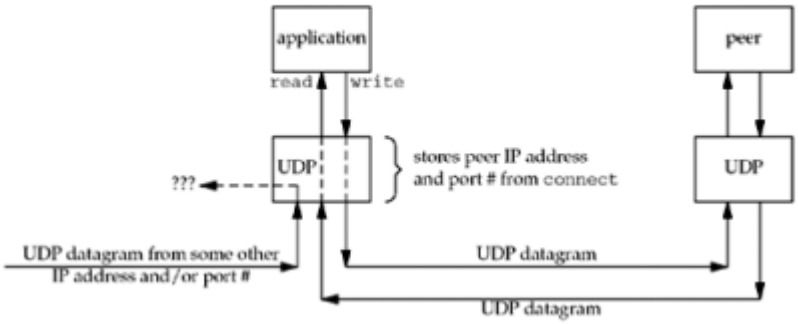
```
#include "unp.h"
void dg_echo(int sockfd, SA *pcliaddr, socklen_t clilen) {
  int n;
  socklen_t len;
  char mesg[MAXLINE];
  for ( ; ; ) {
    len = clilen;
    n = Recvfrom(sockfd, mesg, MAXLINE, 0, pcliaddr, &len);
    Sendto(sockfd, mesg, n, 0, pcliaddr, len);
  }
}
```

- 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:
  - This function never terminates. Since UDP is a connectionless protocol, there is nothing like an EOF as we have with TCP.
  - 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. 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.

5. Unconnected and Connected UDP sockets.

Figure 8.15. Connected UDP socket.



- 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 longer specify the destination IP address and port for an o/p operation that is, we do not use `sendto()`, but use writes 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.
- When an application calls `sendto()` on an unconnected UDP socket, Berkeley-derived kernels temporarily connect the socket, send the datagram, and then unconnect the socket.

6. Identify the resulting changes with a connected UDP socket compared to the

default connected UDP socket?

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

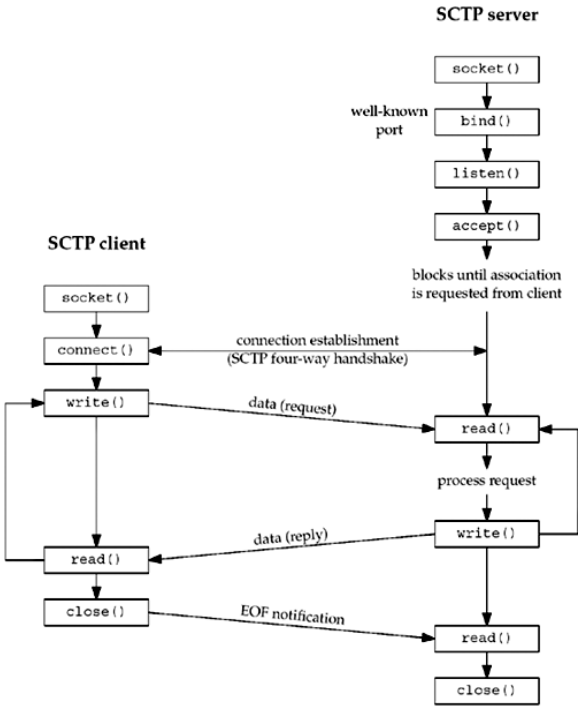
- We can no longer specify the destination IP address and port for an output operation. That is, we do not use `sendto`, but write or send instead. Anything written to a connected UDP socket is automatically sent to the protocol address (e.g., IP address and port) specified by connect.
  Similar to TCP, we can call sendto for a connected UDP socket, but we cannot specify a destination address. The fifth argument to `sendto()` must be a null pointer, and the sixth argument should be 0. The POSIX specification states that when the fifth argument is a null pointer, the sixth argument is ignored.
- 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.
- 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.

7. Different Interface models that are used in SCTP protocol.

There are two types of SCTP sockets:

- one-to-one socket
- one-to-many socket

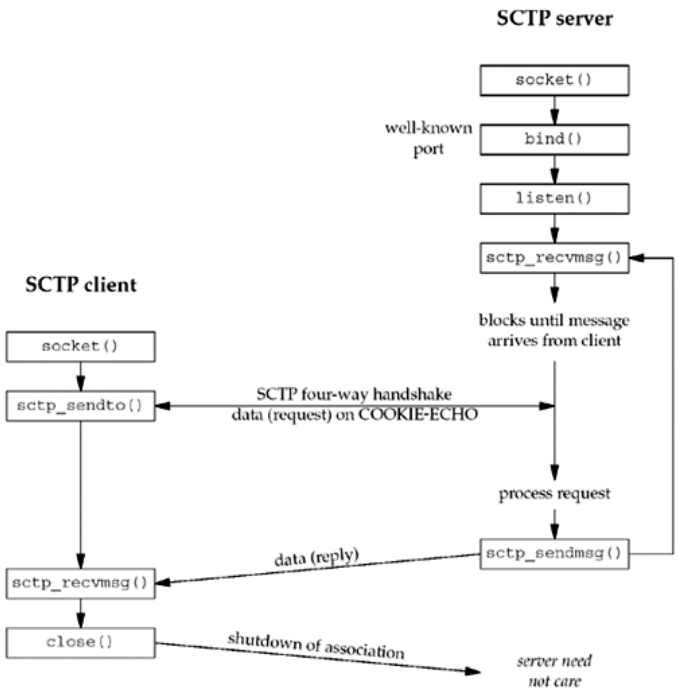
One-to-One Style



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 a normal fashion.

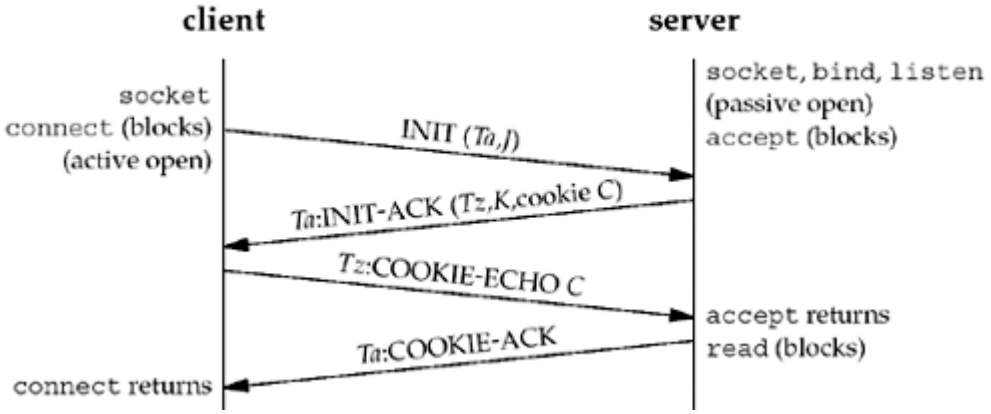
One-to-Many Style



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.

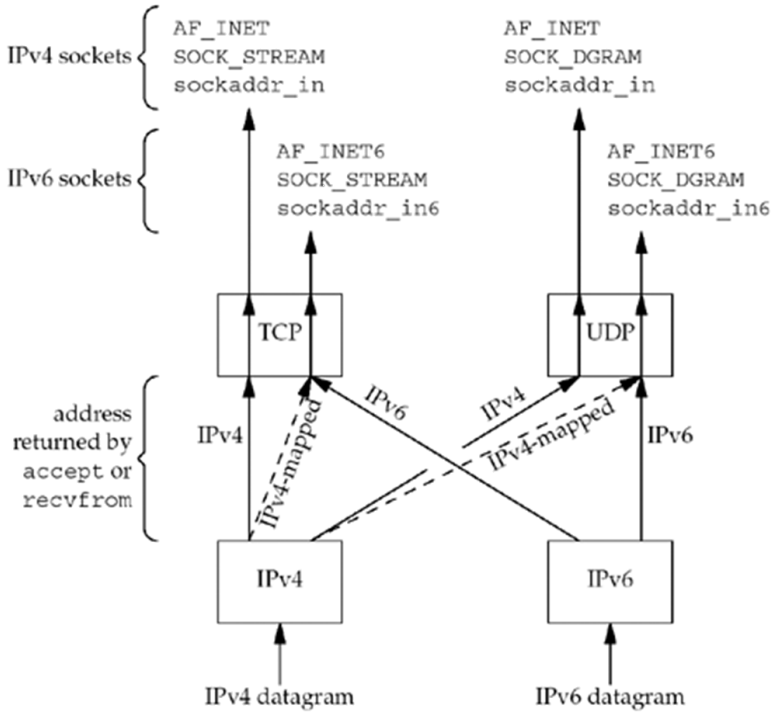
8. Sequence diagram to indicate the functioning of a 4-way handshake for the establishment of an association in SCTP protocol.



Optional Theory, just for understanding concept

- 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, the number of outbound streams the client is requesting, and the 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.

9. Steps that allow an IPv4 TCP client to communicate with an IPv6 server using dual-stack.



The steps are as follows:

- IPV6 Server starts creating 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 the client's host sends an IPV4 SYN to the 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 the IPV4 client

10. `syslogd` daemon. Indicate with a code snippet, how to call the *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:

- 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.
- A Unix domain socket is created and bound to the pathname /var/run/
- A UDP socket is created and bound to port 514 (the syslog service).
- 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 (errno_flag)
    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;
}
```

11. Write a program to echo messages using UDP.

Server

```
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<unistd.h>
#include<netdb.h>
#include<stdio.h>
#include<string.h>
#include<arpa/inet.h>
#define MAXLINE 1024
int main(int argc,char **argv)
{
  int sockfd;
  int n;
  socklen_t len;
  char msg[1024];
  struct sockaddr_in servaddr,cliaddr;
  sockfd=socket(AF_INET,SOCK_DGRAM,0);
  bzero(&servaddr,sizeof(servaddr));
  servaddr.sin_family=AF_INET;
  servaddr.sin_addr.s_addr=INADDR_ANY;
  servaddr.sin_port=htons(5035);
  printf("\n\n Binded");
  bind(sockfd,(struct sockaddr*)&servaddr,sizeof(servaddr));
  printf("\n\n Listening...");
  for(;;)
  {
      printf("\n ");
      len=sizeof(cliaddr);
      n=recvfrom(sockfd,msg,MAXLINE,0,(struct sockaddr*)&cliaddr,&len);
      printf("\n Client's Message : %s\n",msg);
      if(n<6)
        perror("send error");
      sendto(sockfd,msg,n,0,(struct sockaddr*)&cliaddr,len);
  }
  return 0;
}
```

Client

```
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<string.h>
#include<arpa/inet.h>
#include<string.h>
#include<arpa/inet.h>
#include<stdio.h>
#define MAXLINE 1024
int main(int argc,char* argv[])
{
  int sockfd;
  int n;
  socklen_t len;
  char sendline[1024],recvline[1024];
  struct sockaddr_in servaddr;
  strcpy(sendline,"");
  printf("\n Enter the message : ");
  scanf("%s",sendline);
  sockfd=socket(AF_INET,SOCK_DGRAM,0);
  bzero(&servaddr,sizeof(servaddr));
  servaddr.sin_family=AF_INET;
  servaddr.sin_addr.s_addr=inet_addr("127.0.0.1");
  servaddr.sin_port=htons(5035);
  connect(sockfd,(struct sockaddr*)&servaddr,sizeof(servaddr));
  len=sizeof(servaddr);
  sendto(sockfd,sendline,MAXLINE,0,(struct sockaddr*)&servaddr,len);
  n=recvfrom(sockfd,recvline,MAXLINE,0,NULL,NULL);
  recvline[n]=0;
  printf("\n Server's Echo : %s\n\n",recvline);
  return 0;
}
```