Topic 8

Network Programming with Sockets

Objectives

- Have a general understanding of the TCP/IP protocol suite.
- Understand different modes of communication services.
- Know the differences between connection-oriented and connectionless communication services.
- Understand and be able to use the basic socket interface, including commonly used functions.
- Understand and be able to use Unix domain protocol to solve inter-process communication problems.

- Be aware of different byte orders for storing numbers and be able to convert between them.
- Understand and be able to use internet socket addresses.

- Understand and be able to write concurrent servers.
- Understand and be able to handle terminated child server processes.

- Understand the differences between byte streams and messages.
- Be able to handle simple error recovery in UDP programs
- Be able to use TCP to design and implement network applications.
- Be able to use UDP to design and implement network applications.

Readings:

- This lecture notes
- Stevens & Rago: Ch 16
- Skim: Stallings: Chapter 13

Notes to Students

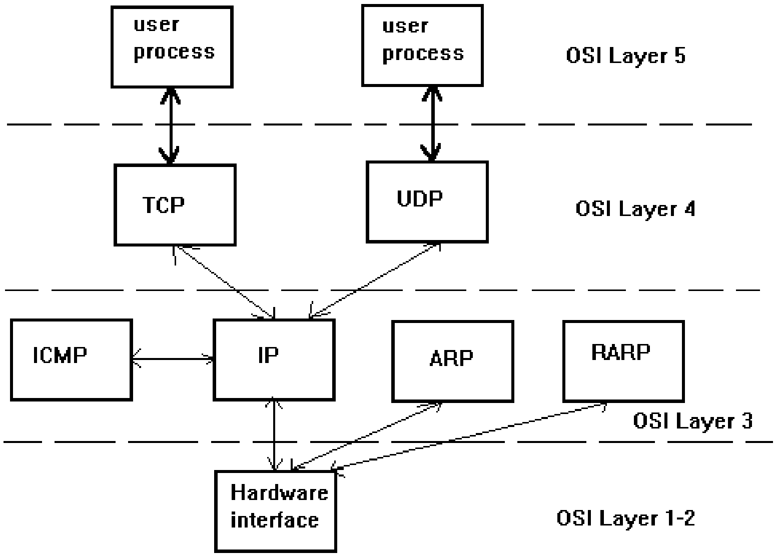
The first part of this lecture covers some essential data communication concepts. If you are already familiar with these concepts, you may skip these sections and start from Section 5 on the socket interface.

The rest of the lecture evolves around 8 examples. The first six examples deal with the same problem – provide a reverse string service. We start from a very crude example (Example 1) containing the bare minimum of features. We then improve each example progressively by identifying a few problems and then solving them in each example.

The first three examples use the Unix domain sockets to communicate between the client and the server. The Unix domain sockets allow a client and its server that are running on the same computer to communicate with each other.

The remaining examples use the TCP or UDP protocol. It is important that you read through each example and understand how each program works. Parts of these examples may be used in your project (Project 2 on simple file transfer protocol, or Project 3 on a simple HTTP client and server).

1. TCP/IP Protocol Suite



TCP: Transmission Control Protocol. This is a *connection-oriented* protocol that provides a reliable (sequencing, flow control, error control), full-duplex, byte-stream for a user process. Most Internet application programs use TCP. Since TCP uses IP, the entire Internet Protocol Suite is often called the *TCP/IP protocol family*.

UDP: User Datagram Protocol. This is a *connectionless* protocol for user processes. Unlike TCP, which is a reliable protocol, there is no guarantee that UDP datagrams will ever reach their intended destinations.

ICMP: Internet Control Message Protocol. The protocol handles error and control information between gateways and hosts. While ICMP messages are transmitted using IP datagrams, these messages are normally guaranteed and processed by the TCP/IP networking software itself, not user processes.

IP: Internet Protocols: IP is the protocol that provides packet delivery services for TCP, UDP, and ICMP. Note that normally a user process does not directly use the IP protocol.

ARP: Address Resolution Protocol. This protocol maps an Internet address into a hardware address, such as an Ethernet address.

RARP: Reverse Address Resolution Protocol. This protocol maps a hardware address (e.g., Ethernet address) into an Internet address.

2. Modes Of Communication Services

(1) Connection-Oriented versus Connectionless

A connection-oriented service requires two processes establishing a logical connection with each other before communication taking place.

- connection establishment
- data transfer
- connection termination

A connectionless service is also called a datagram service. In this type of services, messages are transmitted from one system to the other. Since each message is transmitted independently, it must contain all the information required for its delivery.

Analogy:

- Connection-oriented: - telephone calls
- Connectionless: - sending letters

Internet Protocols:

TCP - connection-oriented
UDP - connectionless

(2) Sequencing

Sequencing describes the property that data are received in the same order with which they were sent.

In a packet-switched network, two consecutive packets can take different routes from the source computer to the destination computer, thus arriving at their destination in a different order from the order in which they were sent.

TCP - providing sequencing
UDP - no sequencing

(3) Error Control

Error control guarantees that error-free data will be delivered to the receiving process. There are two types of lower-level errors:

- Data get corrupted during transmission (may use the checksum to detect).
- Packets get lost during transmission (may wait for acknowledgment to detect)

The two protocols in the transport layer provide the lower-level error control. Once a message is received, you can assume it is error-free. However, UDP does not guarantee that there is no message loss.

TCP - providing error control
UDP - no guarantee that messages will be delivered

(4) Flow Control

Flow control ensures that the sender does not overwhelm the receiver by sending data at a rate faster than the receiver can process the data. If flow control is not provided, the receiver may lose data due to a lack of resources.

TCP - with flow control
UDP - without flow control

(5) Byte-Stream versus Messages

A byte-stream service does not provide any message boundaries to the data stream. The converse of this feature is a message-oriented service that preserves the sender’s message boundaries for the receiver.

TCP - byte-stream protocol
UDP - message-oriented protocol

3. Identifying a Process on the Internet

In a network application, two processes (usually a client and a server) may run on two different hosts (i.e., computers) located in two different networks linked together by an internet. To identify a process on an internet, we need to identify the socket attached to the process. A socket is identified by:

(1) Network ID, to identify the network
(2) Host ID, to identify the host within the network
(3) Port Number, to identify the process running on the host

In TCP/IP (v4), the network ID and the host ID are specified together in a single 32-bit integer. This 32-bit integer is also known as IP number or IP address. The port number is a 16-bit integer. The port spaces for TCP and UPD are separate. For example, TCP port 23 and UDP port 23 are two different ports. IPv6 address consists of 128 bits, of which the first half are used to identify the network and the second half are used to identify the host in the network.

An internet host is often referred to by its domain name, such as ceto.murdoch.edu.au, rather than by its IP number. When the domain name of a host is used, the network program must resolve to its IP number using the domain name resolver `gethostbyname`. The name resolver will first check the file `/etc/hosts` to find out the IP number of the host. Failing that, it will request the DNS to resolve the name.

The servers of most well-known applications (such as ssh and ftp) have a "well-known" port number. E.g.,

ssh	22/TCP
telnet	23/TCP
ftp	21/TCP
smtp	25/TCP

talk	517/UDP
httpd	80/TCP

Port Numbers between 1 and 1023 are reserved for servers of well-known applications. These well-known port numbers are assigned by the Internet Assigned Numbers Authority (IANA). On Unix systems, only processes with superuser privileges may use a port between 1 and1023.

The clients of well-known applications can use any unused port above 1023. All the other applications (servers & clients) should use ports above 1023.

Some of the well-known servers and their well-known ports are listed in file `/etc/services`. A client application may use the server port number directly or use the protocol name defined in file `/etc/services`. The library function `getservbyname` is then used to find out the corresponding port number from that file.

4. Programming Interface

There are two programming interfaces for network programming:

(1) Socket Interface

BSD provides the following system calls for network programming:

```
socket, bind, connect, listen, accept, read,
write, send, recv, sendto, recvfrom, ….
```

(2) TLI (Transport Layer Interface)

System V provides the following system calls for network programming:

```
t_open, t_bind, t_connect, t_listen,
t_accept, t_alloc, t_snd, t_rcv, ….
```

Both programming interfaces are available on Unix systems. These programming interfaces allow the use of not only TCP/IP protocol suite but other protocol families as well, such as:

- XNS (Xerox Network System)
- SNA (IBM System Network Architecture)

Of the two programming interfaces, the socket is more popular (e.g., Windows adopted socket interface). In this unit, we only cover TCP/IP programming using the socket interface.

5. Socket Interface

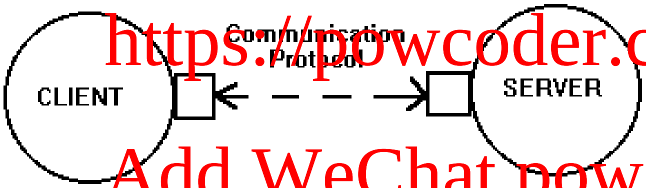
A socket is one end of a communication channel. For two processes to communicate with each other, each process must create a socket and binds an address (e.g., IP number and port number) to that socket. The two sockets must also use the same protocol to communicate with each other. For example:

- Unix Domain Protocol
- TCP
- UDP
- IP

Internet Protocol Family

- SPP (Sequenced Packet Protocol)
- PEX (Packet Exchange Protocol)
- IDP (Internet Datagram Protocol)

XNS – Xerox Network Systems



It is important to understand that the socket interface is designed not just for the TCP/IP protocol family. It was designed for several different protocol families. Since different protocol families use different address formats, different data types are used to represent protocol addresses. For example, TCP/IP (IPv4) uses a 32-bit IP number and a 16-bit port number to identify a socket. While XNS uses a 32-bit network ID, a 48-bit host ID, and a 16-bit port number to identify a socket. The Unix domain protocol simply uses a file name to represent a socket. In the following section, we will introduce the socket system calls without going deep into the protocol addresses.

6. The `socket` System Calls

```
#include <sys/types.h>
#include <sys/socket.h>

int socket(int family, int type,
              int protocol);
```

(a) *family* specifies the Address Family (AF_) or the Protocol Family (PF_) used in communication:

AF_UNIX (or PF_UNIX) - Unix Domain Protocol
AF_INET (or PF_INET) - Internet Protocols (TCP/IP)
AF_NS (or PF_NS) - Xerox NS protocols

(b) *type* specifies the type of communication service, i.e., whether it is stream-like (connection-oriented) or datagram-like (connectionless), or it uses the lower-level protocol. The parameter *type* can be one of the following values:

SOCK_STREAM - stream socket
SOCK_DGRAM - datagram socket
SOCK_RAW - raw socket

Not all combinations of *family* and *type* are valid. For Unix domain protocol and TCP/IP, the following combinations are valid:

	AF_UNIX	AF_INET
SOCK_STREAM	√	√
SOCK_DGRAM	√	√
SOCK_RAW	X	√

(c) The last argument *protocol* is usually implied in the first two arguments. Therefore, we usually use the value 0 (for default) for this argument.

When *protocol* = 0, the implied protocol to be used is given in the following table:

	AF_UNIX	**AF_INET**
SOCK_STREAM	Unix Domain Protocol	TCP
SOCK_DGRAM	Unix Domain Protocol	UDP
SOCK_RAW		IP

Examples: Assume the following declarations:

```
int sd1, sd2, sd3, sd4, sd5;
```

the following statements create various sockets:

```
sd1 = socket(AF_UNIX, SOCK_STREAM, 0);
sd2 = socket(AF_INET, SOCK_STREAM, 0);
sd3 = socket(AF_INET, SOCK_DGRAM, 0);
sd4 = socket(AF_INET, SOCK_RAW, 0);
sd5 = socket(AF_UNIX, SOCK_DGRAM, 0);
```

7. The `bind` System Call

The `bind` system call assigns a name or address to a local socket.

```
#include <sys/types.h>
#include <sys/socket.h>

int bind (int sd,
          struct sockaddr *addr, int addrlen);
```

The first argument, *`sd`*, is the descriptor of a socket (created with the `socket` call), the second argument, *`addr`*, is a pointer to a *protocol-specific* address, and the third argument *`addrlen`* is the size of the address (the number of bytes).

A server must always register a “well-known” address with the operating system using the `bind` system call. This effectively says that “any message destined to this address should be given to this socket.”

A client may choose to assign a specific address to its socket using the `bind` system call. But it is not always necessary.

The type `struct sockaddr` is a generic representation of a protocol address. The actual type and content of a protocol address vary considerably from one protocol family to another. We will consider the data types for the Unix domain protocol as well as TCP/IP later. For the time being, it suffices to think that *`addr`* is just an address for a socket.

8. The `connect` System Call

With a connection-oriented protocol, the client process must use `connect` to establish a connection between the local socket and the remote listening socket in the server process. The `connect` call returns only after the connection between the client and server is established.

With a connectionless protocol, the client may also use the `connect` system call. But the meaning of the `connect` call is different from that of the connection-oriented protocol. It merely attaches a remote address to the local socket, so that the client can use `read` and `recv` calls rather than `recvfrom` call to receive the messages from the remote address, and use `write` and `send` calls rather than `sendto` call to send messages to the remote address.

The system calls `recvfrom` and `sendto` require the socket address of the remote process (either the server or the client).

```
#include <sys/types.h>
#include <sys.socket.h>

int connect (int sd,
    struct sockaddr * remoteSockAddr,
    int addrlen);
```

9. The `listen` System Call

A connection-oriented server must use the system call `listen` to inform the operating system that it is ready to receive connection requests from its clients via the socket.

```
int listen (int sd, int backlog);
```

The argument *sd* is the listening socket. The argument *backlog* specifies how many connection requests can be queued by the system while it waits for the server to execute the `accept` system call. This argument is usually specified as 5.

10. The `accept` System call

After a connection-oriented server called the `listen` system call, an actual connection request from a client process is waited for by having the server executing the `accept` system call:

```
#include <sys/types.h>
#include <sys/socket.h>

int accept (int sd,
     struct sockaddr * peer,
     int * addrlen);
```

On return, this system call creates a new socket for the actual exchange of data between the server and the remote client process that made the request. The client address and its size will also be returned through pointers *peer* and *addrlen*.

11. Sending and Receiving Data

The actual data exchange between the client and the server is done using: `write` or `send`, and `read` or `recv` calls if the destination/source address is not required. Otherwise use the following system calls `sendto` and `recvfrom`.

```
#include  <sys/types.h>
#include  <sys/socket.h>

int send (int sd, char *buf, int nbytes,
      int flag);

int sendto (int sd, char *buf, int nbytes,
      int flag, struct sockaddr *to, int addrlen);

int recv (int sd, char *buf, int nbytes,
      int flag);

int recvfrom (int sd, char *buf, int nbytes,
      int flag, sockaddr *from, int *addrlen);
```

All four system calls return the actual number of bytes written or read, or an error number (negative number).

If no flag is set (i.e., *`flag`*`=0`), `recv` and `send` behave identically as `read` and `write` respectively. The parameter *`flag`* affects the way data are sent or received. For example, if *`flag`*`=MSG_PEEK`, the `recv` call will peek through the incoming data without removing it from the incoming data queue.

12. The Structure of a Client-Server Pair

Connection-Oriented

Server	Client
s = socket(….);	
bind(s, &serv_addr,….);	
listen(s,…..);	
ns = accept(s, ….);	s = socket(….);
	connect(s, &serv addr,….);
	write(s,…..);
read(ns,……);	
process message	
write(ns,…..);	
. . . . . .	read(s,……);
	. . . . . .
close(ns);	close(s);

Connectionless

SERVER	CLIENT
s = socket(…..);	
bind (s, &serv_addr,….);	s = socket(…);
while(continue)	bind(s, &cli_addr,….);
{	. . . . . .
	sendto(s, …, &serv_addr, ….);
recvfrom(s,&cli_addr,….);	
process message;	
sendto(s, &cli_addr,…..);	
}	recvfrom(s, …, &serv_addr,...);

13. The Unix Domain Protocol

The Unix Domain Protocol is just an IPC (Interprocess Communication mechanism), rather than a network communication protocol. It allows two processes on the *same* computer to communicate with each other. The purpose of introducing Unix Domain Protocol is:

(1) Introduce yet another IPC;
(2) Become familiar with socket related system calls;
(3) Understand that the socket interface was designed for multiple protocol families, not just for TCP/IP.

With a Unix domain socket, its address is a file name, just as with FIFOs. Recall that the socket related system calls we have introduced all use the following generic structure for addresses:

```
struct sockaddr
{
  u_short sa_family;   // address family
  char   sa_data[14];    // protocol-specific address
}
```

The actual type to be used depends on the address family (or protocol family). For Unix Domain Protocol, the following type should be used:

```
#include <sys/un.h>

struct sockaddr_un
{
  short sun_family;       // AF_UNIX
  char sun_path[104];  // socket name
}
```

The address is cast back to `struct sockaddr` for the sake of type compatibility.

For example:

```
struct sock_addr_un soc_addr;
char soc_name[] =  "servsoc";

bzero(&soc_addr, sizeof(soc_addr));
                         // initialise to all 0
soc_addr.sun_family = AF_UNIX;
strcpy(soc_addr.sun_path, soc_name);
bind(s, (struct sockaddr *) &soc_addr,
           sizeof(soc_addr));
```

The following is a series of three examples (Example 1 to Example 3). These examples all use Unix domain sockets. The first two are connection-oriented, and the third is connectionless. Each example consists of a client and server pair.

14. Example 1 (`ser1.c` & `cli1.c`).

This example shows the minimum of what is required. Future examples will make incremental improvements over this one. The client and the server use the stream-like Unix domain protocol to communicate with each other. Since Unix domain protocol is not a network protocol, the client and the server must run on the same computer. The server provides a very simple service: echo the message back to the client.

```
/*
 * cli1.c - test for Unix domain stream socket (client part).
 *          This is a very **crude** program.
 */
#include  <unistd.h>
#include  <stdlib.h>
#include  <sys/types.h>
#include  <sys/socket.h>
#include  <sys/un.h>
#include  <string.h>
#include  <stdio.h>
#include  <string.h>

char serversockname[]="serversocket";

int main()
{
     int sd, n;
     char buf[256];
     struct sockaddr_un server_addr;
     char mesg[]="hello from client";

     // create client socket
     sd = socket(PF_UNIX, SOCK_STREAM, 0);

     // construct the server address
     server_addr.sun_family = AF_UNIX;
     strcpy(server_addr.sun_path, serversockname);

     // request connection to serversoc
     connect(sd, (struct sockaddr *)&server_addr, sizeof(server_addr));

     // send a message to server
     write(sd, mesg, strlen(mesg)+1);

     // read the result back from server
     read(sd, buf, sizeof(buf));

     // print out the result
     printf("server echoed '%s'\n", buf);
     exit(0);
}
```

```
/*
 *  ser1.c  test for Unix domain stream socket (server part)
 *          This is a very **crude** program.
 */
#include  <unistd.h>
#include  <stdlib.h>
#include  <string.h>
#include  <sys/types.h>
#include  <sys/socket.h>
#include  <sys/un.h>
#include  <string.h>
#include  <errno.h>

char serversockname[]="serversocket";

int main()
{
     int sd, nsd, n;
     char buf[256];
     struct sockaddr_un server_addr;
     struct sockaddr_un client_addr;
     int  client_addr_len;

     // set up listening socket sd
     sd = socket(PF_UNIX, SOCK_STREAM, 0);

     // construct the server address
     server_addr.sun_family = PF_UNIX;
     strcpy(server_addr.sun_path, serversockname);

     // bind server address to socket sd
     bind(sd, (struct sockaddr *)&server_addr, sizeof(server_addr));

     // set socket sd to a listening socket
     listen(sd, 1);

     // accept connection request
     client_addr_len = sizeof(client_addr);
     nsd = accept(sd, (struct sockaddr *)&client_addr,
                     (socklen_t *)&client_addr_len);

     // data transfer on connected socket ns
     n = read(nsd, buf, sizeof(buf));
     write(nsd, buf, n);

     exit(0);
}
```

15. Example 2 (`ser2.c` & `cli2.c`).

The second example still uses stream-like Unix domain protocol just as Example 1. However, there are several improvements to Example 1:

- The server runs as a daemon.
- The server provides a reverse string service (what a terrific service!). In Example 1, the server merely echoes what it received.
- The server is a *concurrent* server. This means that as soon as the server accepts a client connection request, it forks off a child process to serve that client. The original server (known as the listening server) goes back to wait for the next client request. Therefore, this server can serve multiple clients concurrently (unlike Example 1 which only serves one client). Each child server can process any number of messages.
- The client has also been improved. Now it asks the user to type a message on the keyboard, rather than sending a single fixed message. The client allows the user to send as many messages as he or she wants. To terminate the client (as well as the child server), the user just types "quit".

```
/*
 *  ser2.c  a server program for reversing strings, using
 *          Unix stream socket.  This is a concurrent server.
 */

#include  <unistd.h>
#include  <stdlib.h>
#include  <stdio.h>
#include  <sys/stat.h>
#include  <sys/types.h>
#include  <sys/socket.h>
#include  <sys/un.h>
#include  <string.h>

#define   BUFSIZE   256
char sersockname[]="/tmp/serversoc2"; // must be a full path

void daemon_init(void)
{
        pid_t   pid;

        if ( (pid = fork()) < 0) {
                perror("fork");
                exit(1);
        } else if (pid != 0)
                exit(0);        // parent goes bye-bye

        // child continues
        setsid();               // become session leader
        chdir("/");             // change working directory
        umask(0);               // clear our file mode creation mask
}

void reverse(char *s)
{
    int i, j;
    char c;

    for (i=0, j = strlen(s)-1; i<j; i++, j--) {
        c = s[i]; s[i] = s[j]; s[j] = c;
    }

}

int main()
{
     pid_t pid;
     char buf[BUFSIZE];
     int s, ns, n, cli_addr_len;
     struct sockaddr_un ser_addr, cli_addr;

     daemon_init();      // become a daemon

     // set up listening socket s
     if ((s = socket(PF_UNIX, SOCK_STREAM, 0)) < 0) {
           perror("server:socket"); exit(1);
     }
     // build Unix domain socket address
     ser_addr.sun_family = AF_UNIX;
     strcpy(ser_addr.sun_path, sersockname);

     // bind the server address to s
     unlink(sersockname);                 // just in case
     if (bind(s, (struct sockaddr *) &ser_addr, sizeof(ser_addr) ) < 0){
           perror("server's bind");
           unlink(sersockname);
           exit(1);
     }

     // we are ready to hear connection requests from clients
     listen(s, 5);

     while (1) {
          // acceppt a connection request
          cli_addr_len = sizeof(cli_addr);
          ns = accept(s, (struct sockaddr *) &cli_addr,
                 (socklen_t *)&cli_addr_len);
          if (ns<0) {
              perror("server:accept"); exit(1);
          }

          // create a child process to serve this client
          if ((pid=fork()) <0) {
              perror("fork"); exit(1);
          } else if (pid > 0) {
              close(ns);       // parent to wait for connection
              continue;        // request from next client
          }

          // now in child
          close(s);

          // data transfer on connected socket ns
          while (1){
              // receive a message from that client
              if ((n = read(ns, buf, sizeof(buf))) <= 0)
                  exit(0);   // -1 error, 0 when client dies

              // process the message received
              buf[n] = '\0';     // reverse requires
              reverse(buf);      // a null-terminated string

              // send back the results
              write(ns, buf, n);
          }
     }
}
```

```
/*
 * cli2.c - the client for reversing strings, using Unix stream socket.
 */

#include  <unistd.h>
#include  <stdlib.h>
#include  <sys/types.h>
#include  <sys/socket.h>
#include  <sys/un.h>
#include  <string.h>
#include  <stdio.h>

#define   BUFSIZE   256
char serversockname[]="/tmp/serversoc2";  // the well-known address
                                          // of the server
int main()
{
     int sd, n, i=0;
     char buf[BUFSIZE];
     struct sockaddr_un ser_addr;


     // create a Unix stream socket
     sd = socket(PF_UNIX, SOCK_STREAM, 0);

     // build the server address
     ser_addr.sun_family = AF_UNIX;
     strcpy(ser_addr.sun_path, serversockname);

     // connect client socket to the server
     if (connect(sd,(struct sockaddr *)&ser_addr, sizeof(ser_addr)) < 0){
          perror("connect"); exit(1);
     }

     while (++i) {
          printf("Client Input[%d]: ", i);
          fgets(buf, BUFSIZE, stdin);    // get a message from user
          n = strlen(buf);
          if (buf[n-1] == '\n') {  // get rid of newline in the message
              buf[n-1] = '\0'; --n;
          }
          if (strcmp(buf, "quit")==0) {  // the message is "quit"?
               printf("Bye from client\n"); exit(0);
          }

          if (n > 0) {
               write(sd, buf, n);    // ask server to process the message
               n = read(sd, buf, BUFSIZE); // get back the result
               buf[n] = '\0';
               printf("Sever Output[%d]: %s\n", i, buf);
          }
      }

}
```

16. Example 3 (`ser3.c` & `cli3.c`).

Our third example uses the datagram-like Unix domain protocol. The server is iterative, meaning that it only replies to one client each time (the message may come from different clients though).

```
/*
 *  ser3.c  the server for reversing strings, using Unix datagram
 *          socket.  This is an iterative server - it serves one
 *          client at a time.
 */

#include  <unistd.h>
#include  <stdlib.h>
#include  <stdio.h>
#include  <sys/stat.h>
#include  <sys/types.h>
#include  <sys/socket.h>
#include  <sys/un.h>
#include  <string.h>

char sersockname[]="/tmp/serversoc3"; // must be a full path
char clisockname[]="/tmp/clientsoc3"; // since in daemon

void daemon_init(void)
{
        pid_t   pid;

        if ( (pid = fork()) < 0) {
                perror("fork"); exit(1);
        } else if (pid != 0)
                exit(0);        // parent goes bye-bye

        // child continues

        setsid();               / become session leader

        chdir("/");             // change working directory

        umask(0);               // clear our file mode creation mask
}

void reverse(char *s)
{
    int i, j;
    char c;

    for (i=0, j = strlen(s)-1; i<j; i++, j--) {
        c = s[i]; s[i] = s[j]; s[j] = c;
    }

}
```

```
int main()
{
     char buf[256];
     int sd, nr, nw, cli_addr_len;
     struct sockaddr_un ser_addr, cli_addr;

     // turn the process into a daemon
     daemon_init();

     // create a Unix datagram socket sd
     if ((sd = socket(PF_UNIX, SOCK_DGRAM, 0)) < 0) {
           perror("server:socket"); exit(1);
     }

     // build server socket address
     ser_addr.sun_family = AF_UNIX;
     strcpy(ser_addr.sun_path, sersockname);

     // bind server address to socket sd
     if (bind(sd, (struct sockaddr *)&ser_addr, sizeof(ser_addr)) < 0) {
           perror("server's bind"); unlink(sersockname); exit(1);
     }

     while (1){
          // get a message from a client */
          cli_addr_len = sizeof(cli_addr);
          nr= recvfrom(sd, buf, sizeof(buf), 0,
              (struct sockaddr *) &cli_addr, (socklen_t *) &cli_addr_len);
          if (nr < 0) {                      // error
               perror("revfrom");
               unlink(sersockname); exit(1);
          }

          // process the message
          buf[nr] = '\0'; reverse(buf);

          // send the result to whoever sending us this message
          nw = sendto(sd, buf, nr, 0, (struct sockaddr *) &cli_addr,
                         (socklen_t) cli_addr_len);
          if (nw < nr){                     // error
               unlink(sersockname); exit(1);
          }
     }
}
```

```
/*
 * cli3.c   client to reverse strings, using Unix datagram socket.
 */

#include  <sys/types.h>
#include  <sys/socket.h>
#include  <sys/un.h>
#include  <string.h>
#include  <stdio.h>
#define   BUFSIZE   256
char clientsockname[]="/tmp/clientsoc3";
char serversockname[]="/tmp/serversoc3";

int main()
{
     char buf[256];
     int sd, n, nr, nw, i=0, cli_addr_len;
     struct sockaddr_un ser_addr, cli_addr;

     // create a Unix datagram socket sd
     sd = socket(PF_UNIX, SOCK_DGRAM, 0);

     // build a client socket address, and bind it to sd
     cli_addr.sun_family = AF_UNIX;
     strcpy(cli_addr.sun_path, clientsockname);
     if (bind(sd, (struct sockaddr *) &cli_addr, sizeof(cli_addr)) < 0){
          perror("bind"); exit(1);
     }

     // build a server socket address
     ser_addr.sun_family = AF_UNIX;
     strcpy(ser_addr.sun_path, serversockname);

     while (++i) {
          printf("Client Input[%d]: ", i);
          fgets(buf, BUFSIZE, stdin);   // get a message from user
          n = strlen(buf);
          if (buf[n-1] == '\n') {     // get rid of newline
               buf[n-1] = '\0'; --n;
          }
          if (strcmp(buf, "quit")==0) {  // is the message "quit"?
               printf("Bye from client\n");
               unlink(clientsockname); exit(0);
          }
          if (n > 0) {
               // send the message to the server
               nw = sendto(sd, buf, n, 0,
                      (struct sockaddr *) &ser_addr, sizeof(ser_addr));
               if (nw < n)
                   printf("client: sendto error\n");
               // receive the response from server
               nr = recv(sd, buf, sizeof(buf), 0);
               buf[nr] = '\0';
               printf("Sever Output[%d]: %s\n", i, buf);
          }
      }
}
```

17. Internet Socket Address

```
#include <netinet/in.h>

struct in_addr
{
    in_addr_t  s_addr;
};

struct sockaddr_in
{
    sa_family_t  sin_family;  // AF_INET
    in_port_t  sin_port;      //16-bit port
    struct in_addr sin_addr;  //32-bit IP number
    char sin_zero[8];         //unused
};
```

Note:

(1) The types for `sin_family` and `sin_port` above are new, POSIX.1g compliant. The previous types were `short` and `u_long`.

(2) The field `sin_addr` is a structure containing a single 32-bit number (`in_addr_t`), rather than a straightforward `in_addr_t` type due to historical reasons.

(3) Both the port number (`sin_port`) and the IP number (`sin_addr`) must be in *network byte order.*

18. Network Byte Order

As we know, different processor architectures use different number representations, especially in the byte order of the 2-byte quantity (`short`), and 4-byte quantity (`int`). There are two types of byte-orderings:

Little-Endian: in this scheme, the least significant byte in a number is placed at the starting address.

```
short   x = 0xA038
int     y = 0xB25A6078;
char s[6] = “hello”;
```

100	102	106
38 A0	78 60 5A B2	h e l l o \0

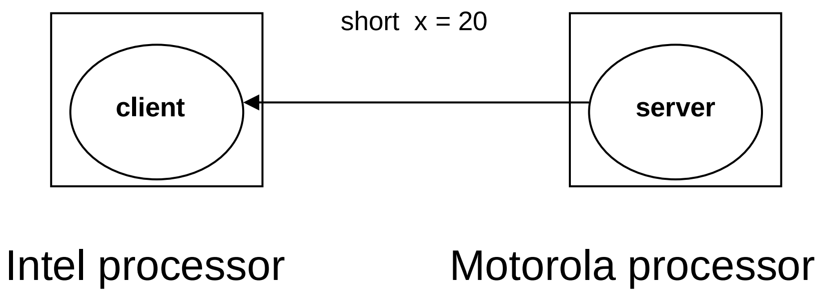
Architectures: Intel 80x86, DEC VAX, DEC PDP-11, DEC ALPHA

Big-Endian: in this scheme, the most significant byte is placed at the starting address.

100	102	106
A0 38	B2 5A 60 78	h e l l o \0

Architectures: IBM 370, Motorola 69000, MIPS, Sun SPARC.

Since the two end processes of communication may run on two computers with different processor architectures, one with little-endian byte order, and the other with big-endian byte order, care must be taken to ensure that the receiving process can correctly interpret what it receives.



Example:
A server running on a Motorola processor sends a short integer `x = 20` to its client running on an Intel processor.

Motorola processor sends two bytes: `00 14`

When the Intel processor receives the two bytes, it puts them on a short variable, such as:

```
    short y: 00 14
```

which is considered to be `0x14 x 256 + 0 = 20 x 256 = 5120` by the Intel processor, rather than `20`.

To avoid this kind of errors, we must convert any 2-byte and 4-byte integers into the common byte order (network byte order) before sending them over the network:

```
    u_long htonl(u_long hostlong);
    u_short htons(u_short hostshort);
```

On the receiving end, the process should convert any multiple-byte quantity from the network byte order back to the order used by the underlying processor architecture:

```
    u_long  ntohl (u_long  netlong);
    u_short ntohs (u_short netshort);
```

There are also functions for 64-bit numbers such as `htobe64` and `be64toh` for conversion between host byte order and big-endian, and `htole64`, `le64toh` for conversion between host byte order and little-endian. However, these functions are not yet standardised.

Example:

```
struct sockaddr_in  server_addr;

server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(4000);
server_addr.sin_addr.s_addr = htonl (2255700040);
```

19. Example 4 (`ser4.c` & `cli4.c`)

The following example is the modified version of Example 2. We use TCP sockets instead of Unix domain sockets to provide transport. Unlike Example 1 to 3, the server and client can run on different machines as long as there is a TCP/IP connection.

```
/*
 *  ser4.c  a server for reversing strings, using TCP stream socket.
 *          Server machine's address (134.115.64.72) is hard
 *          coded in the server & client.
 */

#include  <unistd.h>
#include  <stdlib.h>
#include  <stdio.h>
#include  <sys/stat.h>
#include  <sys/types.h>
#include  <sys/socket.h>
#include  <netinet/in.h>      // struct sockaddr_in, htons, htonl
#include  <string.h>

#define   BUFSIZE         256
#define   SERV_INET_NO    2255700040     // IP number 134.115.64.72
#define   SERV_TCP_PORT   40000          // server port number

void daemon_init(void)
{       pid_t  pid;

        if ( (pid = fork()) < 0) {
             perror("fork"); exit(1);
        } else if (pid > 0)
             exit(0);        // parent goes bye-bye
        // child continues
        setsid();                  // become session leader
        chdir("/");                // change working directory
        umask(0);                  // clear our file mode creation mask
}

void reverse(char *s)
{   char c;  int i, j;

    for (i=0, j = strlen(s)-1; i<j; i++, j--) {
        c = s[i]; s[i] = s[j]; s[j] = c;
    }
}

void serve_a_client(int sd)
{
    int nr, nw;
    char buf[BUFSIZE];
    while (1){
         // read a message from new socket sd
         if ((nr = read(sd, buf, sizeof(buf))) <= 0)
             exit(0);   // connection down

         // process the message
         buf[nr] = '\0'; reverse(buf);

         // send processed message to client
         nw = write(sd, buf, nr);
    }
}

int main()
{
     int sd, nsd, n, cli_addrlen;  pid_t pid;
     struct sockaddr_in ser_addr, cli_addr;

     // turn the program into a daemon
     daemon_init();

     // set up listening socket sd
     if ((sd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
           perror("server:socket"); exit(1);
     }

     // build server listening socket address
     bzero((char *)&ser_addr, sizeof(ser_addr));
     ser_addr.sin_family = AF_INET;
     ser_addr.sin_port = htons(SERV_TCP_PORT);
     ser_addr.sin_addr.s_addr = htonl(SERV_INET_NO);

     // bind server address to the listening socket sd
     if (bind(sd, (struct sockaddr *) &ser_addr, sizeof(ser_addr))<0){
           perror("server bind"); exit(1);
     }

     // become a listening socket
     listen(sd, 5);

     while (1) {
          // wait to accept a client request for connection
          cli_addrlen = sizeof(cli_addr);
          nsd = accept(sd, (struct sockaddr *) &cli_addr,
                   (socklen_t *)&cli_addrlen);
          if (nsd < 0) {
               perror("server:accept"); exit(1);
          }

          // create a child process to serve this client
          if ((pid=fork()) <0) {
              perror("fork"); exit(1);
          } else if (pid > 0) {
              close(nsd);
              continue;    // parent to wait for next client
          }
          // now in child, serve the current client
          close(sd);
          serve_a_client(nsd);
     }
}
```

```
/*
 * cli4.c   a client for reversing strings, using TCP socket.
 *          The server machine address (134.115.64.72) is hardcoded.
 */

#include  <unistd.h>
#include  <stdlib.h>
#include  <sys/types.h>
#include  <sys/socket.h>
#include  <netinet/in.h>       //  struct sockaddr_in, htons, htonl
#include  <string.h>
#include  <stdio.h>

#define   BUFSIZE      256
#define SERV_TCP_PORT  40000       // "well-known" server port number
#define SERV_INET_NO   2255700040  // "134.115.64.72"

int main()
{
     int sd, n, nr, nw, i=0;
     char buf[BUFSIZE];
     struct sockaddr_in ser_addr;

     // build a server socket address
     bzero((char *) &ser_addr, sizeof(ser_addr));
     ser_addr.sin_family = AF_INET;
     ser_addr.sin_port = htons(SERV_TCP_PORT);
     ser_addr.sin_addr.s_addr = htonl(SERV_INET_NO);

     // create TCP socket & connect socket to server address
     sd = socket(AF_INET, SOCK_STREAM, 0);
     if (connect(sd, (struct sockaddr *) &ser_addr, sizeof(ser_addr))<0) {
          perror("client connect"); exit(1);
     }

     while (++i) {
          printf("Client Input[%d]: ", i);
          fgets(buf, BUFSIZE, stdin);    // get a message from user
          nr = strlen(buf);
          if (buf[nr-1] == '\n') { buf[nr-1] = '\0'; --nr; }

          if (strcmp(buf, "quit")==0) {  // is the message "quit"?
               printf("Bye from client\n"); exit(0);
          }

          if (nr > 0) {
               nw = write(sd, buf, nr);
               nr = read(sd, buf, BUFSIZE); buf[nr] = '\0';
               printf("Sever Output[%d]: %s\n", i, buf);
          }
     }
}
```

20. Problems in Example 4

(1) The server IP address was hardcoded in the program. Consequently, the server can only run on that specific host with IP number `134.115.64.72`;

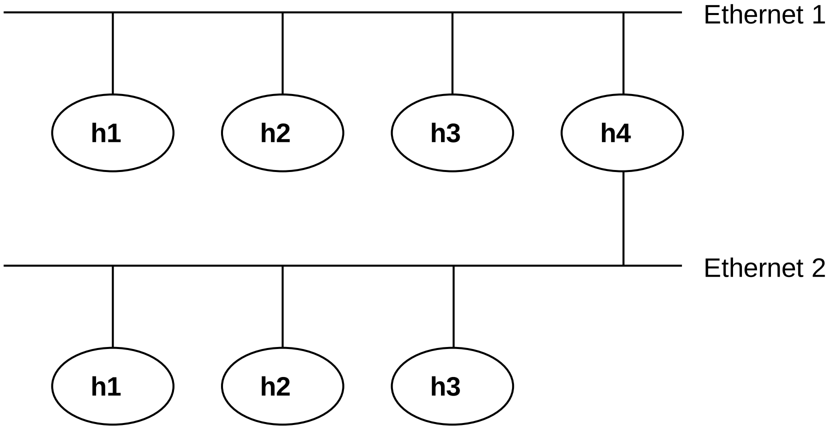
***Solution**:* instead of using a fixed IP address, we use a generic number to represent the localhost:

```
INADDR_ANY
```

as in the following example:

```
bzero ((char *)& ser-addr, sizeof(ser_addr));
ser_addr.sin_family = AF_INET;
ser_addr.sin_port = htons(SERV_TCP_PORT);
ser_addr.sin_addr.s_addr = htonl(INADDR_ANY);
```

This means the server will receive data sent to any one of the network interface devices on the localhost. For example, in the following diagram, host h4 has two Ethernet network interface cards. Using `INADDR_ANY` on h4, the server process can receive messages sent to it through either network interface.



(2) The client has the server's IP number hardcoded in the program. Consequently, it can only communicate with the server running on the host with IP number `134.115.64.72`.

***Solution**:* The user provides the name of the host on which the server is running. The client calls the function:

`gethostbyname (`*`hostname`*`)`

to obtain the IP number of the *hostname*. The function `gethostbyname` first looks through the file `/etc/hosts` on the local host to find the IP number (in dotted decimal format) of the host. If the host is not listed in the file, the function will then ask a "Name Server" to find the IP number for that host.

The function `gethostbyname` returns a pointer to the following structure which is defined in `<netdb.h>`

```
struct hostent
{
   char * h_name;        //official name of host
   char **h_aliases;     //aliase list
   int h_addrtype;       //AF_INET
   int h_length;         //length of address
   char **h_addr_list;   //addresses from name server
}
//first address
#define h_addr  h_addr_list[0]
```

Here `h_addr_list` is an array of pointers, each points to one address. Since IP address (or IP number) is a 32-bit integer (IPv4), each pointer is pointing to one value of `u_long type` (`unsigned long`), rather than one character. Therefore, we must cast that pointer to a `u_long` pointer.

Example:

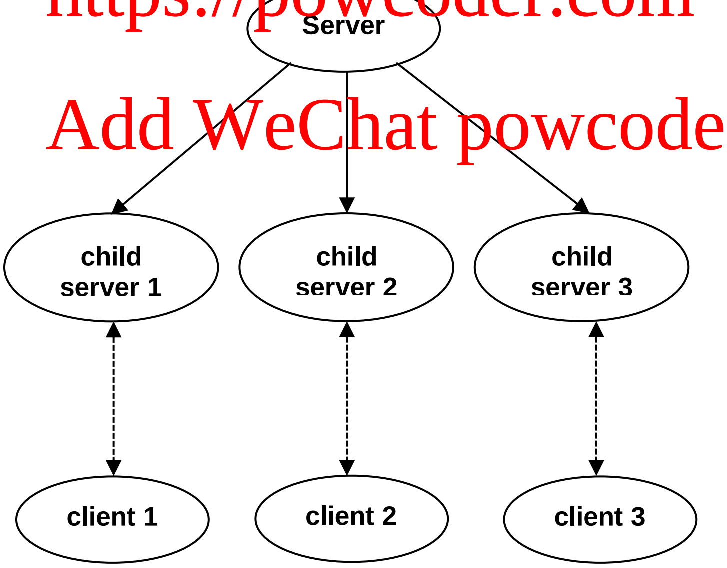
```
struct hostent *hp;
hp = gethostbyname("ceto.murdoch.edu.au");

ser_addr.sin_addr.s_addr = *(u_long *)hp->h_addr;
```

or

```
ser_addr.sin_addr.s_addr
             = *(u_long *)hp->h_addr_list[0];
```

Since this address is already in the network byte order, no conversion is needed.

(3) The server is a concurrent server, i.e., it forks off a child process to handle each client.



Since a child server will terminate when its client terminates, and when a child server terminates, its parent (i.e., the original listening server) is still alive and has not attempted to claim the child, the child server will become *zombie process*. For a zombie process, most resources it took have been released but it still occupies an entry in the process table. Since each zombie process takes up one slot in the process table, zombies may accumulate quickly and exhaust the process table.

To prevent zombies from filling up the process table, the server must claim its child whenever one of its child processes terminates. When a child dies, it sends the signal SIGCHLD to its parent process. We can catch this signal in the parent process (the listening server) and use waitpid to claim the child.

```
void claim_children()
{   pid_t  pid=1;
    while (pid > 0)
        pid = waitpid(0, (int *)0, WNOHANG);
}
```

Of course, we need to set up the signal handler for SIGCHLD first:

```
struct sigaction act;
act.sa_handler = claim_children;
sigemptyset(act.sa_mask);
act.sa_flags = SA_NOCLDSTOP;
sigaction(SIGCHLD, (struct sigaction *) &act,
                   (struct sigaction *) 0);
```

Usually, SIGCHLD is generated when a child is terminated or *stopped* (job control). If we do not want SIGCHLD generated when the child is *stopped*, we can set the option SA_NOCLDSTOP when installing the signal handler for SIGCHLD. In the above example, the handler is only called when a child terminates. It will not be called when a child is stopped.

(4) Since a signal may interrupt a "slow" system call, i.e., causing it to return prematurely, care must be taken to prevent malfunction of your programs. System calls such as `accept` and `read` are slow system calls. They could be interrupted by an incoming signal.

```
while(1)
{
    nsd=accept(...);
    if (nsd < 0)
        if (errno == EINTR)
           continue;

}
```

We have incorporated the above changes in Example 5 in the next section.

21. Example 5

```
/*
 *  ser5.c  a much improved (still not perfect, though) version
 *          of "ser4.c".
 */
#include  <unistd.h>
#include  <stdlib.h>
#include  <stdio.h>
#include  <sys/stat.h>
#include  <string.h>     // strlen(), strcmp() etc
#include  <errno.h>      // extern int errno, EINTR, perror()
#include  <signal.h>     // SIGCHLD, sigaction()
#include  <sys/types.h>  // pid_t, u_long, u_short
#include  <sys/socket.h> // struct sockaddr, socket(), etc
#include  <sys/wait.h>   // waitpid(), WNOHAND
#include  <netinet/in.h> // struct sockaddr_in, htons(), htonl(),
                         // and INADDR_ANY

#define   BUFSIZE         256
#define   SERV_TCP_PORT   40001           // server port no

void claim_children()
{
     pid_t pid=1;

     while (pid>0) {       // claim as many zombies as we can
         pid = waitpid(0, (int *)0, WNOHANG);
     }
}

void daemon_init(void)
{
     pid_t   pid;
     struct sigaction act;

     if ( (pid = fork()) < 0) {
          perror("fork"); exit(1);
     } else if (pid > 0)
          exit(0);                  // parent goes bye-bye

     // child continues
     setsid();                      // become session leader
     chdir("/");                    // change working directory to root
     umask(0);                      // clear file mode creation mask

     // catch SIGCHLD to remove zombies from system
     act.sa_handler = claim_children; // use reliable signal
     sigemptyset(&act.sa_mask);       // not to block other signals
     act.sa_flags   = SA_NOCLDSTOP;   // not catch stopped children
     sigaction(SIGCHLD,(struct sigaction *)&act,(struct sigaction *)0);
     /* note: a less than perfect method is to use
             signal(SIGCHLD, claim_children);
     */
}
```

```
void reverse(char *s)
{
    char c;
    int i, j;

    for (i=0, j = strlen(s)-1; i<j; i++, j--) {
        c = s[i]; s[i] = s[j]; s[j] = c;
    }
}

void serve_a_client(int sd)
{   int nr, nw;
    char buf[BUFSIZE];

    while (1){
         // read a message from client
         if ((nr = read(sd, buf, sizeof(buf))) <= 0)
             exit(0);   // connection broken down

         // process the message
         buf[nr] = '\0'; reverse(buf);

         // send the processed message to client
         nw = write(sd, buf, nr);
    }
}

int main()
{
     int sd, nsd, n, cli_addrlen;  pid_t pid;
     struct sockaddr_in ser_addr, cli_addr;

     // turn the program into a daemon
     daemon_init();

     // set up listening socket sd
     if ((sd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
           perror("server:socket"); exit(1);
     }

     // build server listening socket address
     bzero((char *)&ser_addr, sizeof(ser_addr));
     ser_addr.sin_family = AF_INET;
     ser_addr.sin_port = htons(SERV_TCP_PORT);
     ser_addr.sin_addr.s_addr = htonl(INADDR_ANY);
     /* note: accept client request sent to any one of the
        network interface(s) on this host.
     */

     // bind server address to socket sd
     if (bind(sd, (struct sockaddr *) &ser_addr, sizeof(ser_addr))<0){
           perror("server bind"); exit(1);
     }

     // become a listening socket
     listen(sd, 5);
     while (1) {

          // wait to accept a client request for connection
          cli_addrlen = sizeof(cli_addr);
          nsd = accept(sd, (struct sockaddr *) &cli_addr,
                 (socklen_t *)&cli_addrlen);
          if (nsd < 0) {
               if (errno == EINTR)   // if interrupted by SIGCHLD
                    continue;
               perror("server:accept"); exit(1);
          }

          // create a child process to handle this client
          if ((pid=fork()) <0) {
              perror("fork"); exit(1);
          } else if (pid > 0) {
              close(nsd);
              continue;    // parent to wait for next client
          }

          // now in child, serve the current client
          close(sd);
          serve_a_client(nsd);
     }
}
```

```
/* cli5.c   a much improved (still imperfect) version of "cli4.c".*/
#include  <unistd.h>
#include  <stdlib.h>
#include  <sys/types.h>
#include  <sys/socket.h>
#include  <netinet/in.h>       // struct sockaddr_in, htons, htonl
#include  <netdb.h>            // struct hostent, gethostbyname()
#include  <string.h>
#include  <stdio.h>

#define   BUFSIZE        256
#define   SERV_TCP_PORT  40001 // server's "well-known" port number

int main(int argc, char *argv[])
{
     int sd, n, nr, nw, i=0;
     char buf[BUFSIZE], host[BUFSIZE];
     struct sockaddr_in ser_addr; struct hostent *hp;

     // get server host name from command line
     if (argc==1)                      // if no host name, assume the server
         strcpy(host, "localhost");    // runs on the localhost
     else if (argc == 2)               // use the given host name
         strcpy(host, argv[1]);
     else {
         printf("Usage: %s [<server_host_name>]\n", argv[0]); exit(1);
     }

     // get host address, & build a server socket address
     bzero((char *) &ser_addr, sizeof(ser_addr));
     ser_addr.sin_family = AF_INET;
     ser_addr.sin_port = htons(SERV_TCP_PORT);
     if ((hp = gethostbyname(host)) == NULL){
           printf("host %s not found\n", host); exit(1);
     }
     ser_addr.sin_addr.s_addr = * (u_long *) hp->h_addr;

     // create TCP socket & connect socket to server address
     sd = socket(AF_INET, SOCK_STREAM, 0);
     if (connect(sd, (struct sockaddr *) &ser_addr, sizeof(ser_addr))<0) {
          perror("client connect"); exit(1);
     }

     while (++i) {
          printf("Client Input[%d]: ", i);
          fgets(buf, BUFSIZE, stdin); nr = strlen(buf);
          if (buf[nr-1] == '\n') { buf[nr-1] = '\0'; --nr; }
          if (strcmp(buf, "quit")==0) {
               printf("Bye from client\n"); exit(0);
          }
          if (nr > 0) {
               nw = write(sd, buf, nr);
               nr = read(sd, buf, BUFSIZE); buf[nr] = '\0';
               printf("Sever Output[%d]: %s\n", i, buf);
          }
     }
}
```

22. Some Useful Functions

(1) Address Conversion

An Internet address is often written in the dotted-decimal format for the sake of human readability. For example:

```
134.115.64.72
```

However, it is stored internally as a 32-bit binary unsigned long integer. Written in decimal notation, this number is `2255700040`.

To convert an internet address between the two formats, we use the following routines:

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

unsigned long inet_addr (char *ptr);
char *inet_ntoa (struct in_addr inaddr);
```

Example: The function call

```
inet_addr ("134.115.64.72")
```

returns the unsigned long value `2255700040`, while the function call:

```
inet_ntoa((struct in_addr)2255700040)
```

returns a pointer to string “`134.115.64.72`”.

(2) Socket Address

The following routine is used to obtain the address associated with a socket descriptor:

```
#include <sys/types.h>
#include <sys/socket.h>

int getsockname (int sd, struct sockaddr * addr,
                      int *addrlen);
```

In a connection-oriented application, we may want to know the remote address of a connection. The following routine can be used:

```
#include <sys/types.h>
#include <sys/socket.h>

int getpeername (int sd, struct sockaddr
                  *peeraddr, int *peeraddrlen);
```

Note: getsockname is used to obtain the address bound to the local socket, while getpeername is used to get the address of the remote socket at the other end of the connection.

(3) Port Numbers of Some Standard Applications

The file `/etc/services` contains the port numbers and the protocols used to implement the server part of some standard network applications such as ssh, ftp, telnet, rlogin, finger, talk, etc. A typical `/etc/services` file may look like this:

```
#name  port#/protocol  aliases   # comments

ftp     21/tcp
ftp     21/udp
sftp    115/tcp
sftp    115/udp
telnet  23/tcp
telnet  23/udp
ssh     22/tcp
ssh     22/udp
smtp    25/tcp      # Simple Mail Transfer
smtp    25/udp      # Simple Mail Transfer
time    37/tcp      # timeserver
time    37/udp      # timeserver
name    42/udp      # nameserver
finger  79/tcp
login   513/tcp
printer 515/tcp
who     513/udp
talk    517/udp
nfsd    2049/udp
xerver0 6000/tcp
```

These port numbers are “well known”. This is why it is possible to ftp any machine on the Internet. When choosing a port number for our own server, we must avoid using these port numbers that have already been assigned to a protocol. Given the name of a well-known protocol, we may use the following function to search file `/etc/services` to obtain its port number:

```
#include  <netdb.h>
struct  servent  *
getservbyname (char * servname, char * protname);
```

This function returns a pointer to the following structure:

```
struct servent
{
    char *s_name;      //official service name
    char **s_aliases;  //alias list
    int s_port;        //port number, already in
                       //network byte order
    char *s_proto;     //protocol to use
}
```

Example: Server port number for FTP

```
struct servent *sp;

sp = getservbyname ("ftp", "tcp");
printf("ftp port = %d\n", ntohs(sp->s_port);
```

(4) Byte Operations

The following three routines are used to manipulate non-string data (binary data, which may contain the null character `'\0'` in the data):

```
bzero(char * dest, int nbytes);
bcopy(char *source, char *dest, int nbytes);
int bcmp(char *ptr1, char *ptr2, int nbytes);
```

For example, we may copy the following structure:

```
    struct person
    {
        char  name[40];
        short  age;
    }
```

to a buffer:

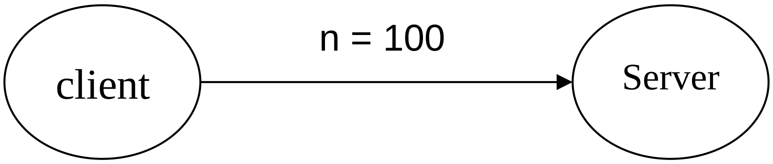
```
    char buf[BUFIZE]; //BUFSIZE > sizeof person
    struct peson p;

    strcpy (p.name, “Kathy Freeman”);
    p.age = 20;
    bcopy ((char *) &p, buf, sizeof(p));
```

Note: We cannot use `strcpy` to do the job (why?). Compare

`bcopy` with `strcpy`
`bcmp` with `strcmp`

23. How to Send Numbers

Suppose we want to send an integer to the server, how should we do it?



We must convert the integer into the network byte order before sending it, and convert it back to the host byte order after receiving it.

Sender:

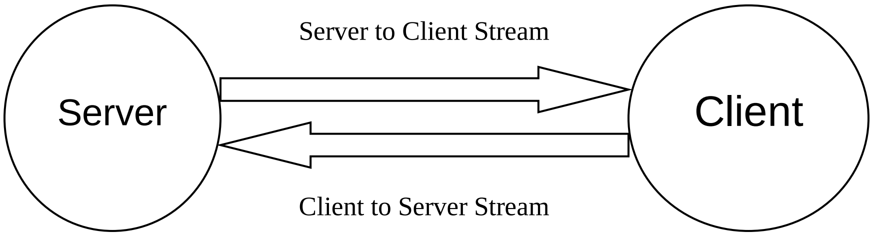
```
int n, nn;

n =100;
if (sizeof (n) == 2)
    nn = htons(n);
else
    nn = htonl(n);
    write (sd, (char *) &nn, sizeof (n));
```

Receiver:

```
int nn, n;

read (sd, (char *) &nn, sizeof (nn));
if (sizeof (n) == 2)
    n = ntohs(nn);
else
    n = ntohl(nn);
```

24. Byte Streams vs Messages

As a stream-oriented protocol, TCP keeps two streams of bytes in a connection, one for each direction.



However, the message boundaries are not preserved. For example, if the client sends:

```
Success is walking from failure to failure with no loss of enthusiasm.
```

The server may only receive "`Success is walking from failure to failure`" with the first read. Subsequent reads may obtain the rest of the message.

This is because when the first read begins, only the first part of the message has arrived. In this case, a `read` call will return whatever is available to the server at that time without waiting for the complete message to arrive.

On the contrary, the message-oriented protocol UDP preserves the message boundaries. The receiver will get the whole message with a single read, i.e., the `read` call will not return until the complete message has arrived. However, if the server only intend to read the first 10 bytes, it will only get the first 10 bytes in the message. The rest of the message is lost. The subsequent read will read in the next message, not the rest of the first message.

The same principle also applies to the stream write (TCP) and message write (UDP).

Example 7 demonstrates the effects of the stream I/O: the connection-oriented protocol such as TCP does not preserve the message boundary. Note that Example 7 is independent of the previous examples. It is not an improvement to Example 5 or Example 6.

Example 6 is the improved version of Example 5. This is achieved by incorporating the stream read and stream write in the programs. Both the client (`cli6.c`) and the server (`ser6.c`) include the head file "`stream.h`". The programs can be compiled as follows:

```
% gcc  -o ser6  ser6.c  stream.c
% gcc  -o cli6  cli6.c  stream.c
```

We will look at Example 7 first. The source code and protocol specification for Example 6 follows Example 7.

25. Example 7

The client sends a sequence of messages of different sizes (e.g., 10 bytes, 100 bytes, 1000 bytes, etc) using TCP. The server is expected to receive each message, process it, and then send the outcome back to the client.

```
/*
 *   ser7.c  Test stream I/O  (server part)
 */
#include  <unistd.h>
#include  <stdlib.h>
#include  <stdio.h>
#include  <string.h>     /* strlen(), strcmp() etc */
#include  <errno.h>      /* extern int errno, EINTR, perror() */
#include  <sys/types.h>  /* pid_t, u_long, u_short */
#include  <sys/socket.h> /* struct sockaddr, socket(), etc */
#include  <netinet/in.h> /* struct sockaddr_in, htons(), htonl(), */
                         /* and INADDR_ANY */

#define   SERV_TCP_PORT   40004  /* server port no */
#define   BUFSIZE         (1024*256)

void reverse(char *s, int len)
{
    char c;
    int i, j;

    for (i=0, j = len-1; i<j; i++, j--) {
        c = s[i]; s[i] = s[j]; s[j] = c;
    }
}

void serve_a_client(int sd)
{   int nr, nw, i=0;
    char buf[BUFSIZE];

    while (++i){

         // read a message from client
         if ((nr = read(sd, buf, sizeof(buf))) <= 0)
             exit(0);   // connection broken down
         printf("server[%d]: %d bytes received\n", i, nr);

         // process the message we have received
         reverse(buf, nr);
         printf("server[%d]: %d bytes processed\n", i, nr);

         // send the processed message to client
         nw = write(sd, buf, nr);
         printf("server[%d]: %d bytes sent out\n", i, nw);
    }
}
```

```
int main()
{
     int sd, nsd, n, cli_addrlen;  pid_t pid;
     struct sockaddr_in ser_addr, cli_addr;

     // set up listening socket sd
     if ((sd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
           perror("server:socket"); exit(1);
     }

     // build server Internet socket address
     bzero((char *)&ser_addr, sizeof(ser_addr));
     ser_addr.sin_family = AF_INET;
     ser_addr.sin_port = htons(SERV_TCP_PORT);
     ser_addr.sin_addr.s_addr = htonl(INADDR_ANY);
     /* note: accept client request sent to any one of the
        network interface(s) on this host.
     */

     // bind server address to socket sd
     if (bind(sd, (struct sockaddr *) &ser_addr, sizeof(ser_addr))<0){
           perror("server bind"); exit(1);
     }

     // become a listening socket
     listen(sd, 5);

     while (1) {

          // wait to accept a client request for connection
          cli_addrlen = sizeof(cli_addr);
          nsd = accept(sd, (struct sockaddr *) &cli_addr,
                    (socklen_t *)&cli_addrlen);

          // create a child process to handle this client
          if ((pid=fork()) <0) {
              perror("fork"); exit(1);
          } else if (pid > 0) {
              close(nsd);
              continue; // parent to wait for next client
          }

          // now in child, serve the current client
          close(sd);
          serve_a_client(nsd);
     }
}
```

```
/* cli7.c   Test stream I/O (client part)
 */
#include  <unistd.h>
#include  <stdlib.h>
#include  <sys/types.h>
#include  <sys/socket.h>
#include  <netinet/in.h>       // struct sockaddr_in, htons, htonl
#include  <netdb.h>            // struct hostent, gethostbyname()
#include  <string.h>
#include  <stdio.h>

#define   SERV_TCP_PORT  40004 /* server's "well-known" port number */
#define   BUFSIZE        (1024*256)

int main(int argc, char *argv[])
{
     int sd, n, nr, nw, i=0, j;
     char buf1[BUFSIZE], buf2[BUFSIZE], host[60];
     struct sockaddr_in ser_addr; struct hostent *hp;

     // get server host name
     if (argc==1)           // assume server running on the local host
         strcpy(host, "localhost");
     else if (argc == 2)    // use the given host name
         strcpy(host, argv[1]);
     else {
         printf("Usage: %s [<server_host_name>]\n", argv[0]); exit(1);
     }

     // get host address, & build a server socket address
     bzero((char *) &ser_addr, sizeof(ser_addr));
     ser_addr.sin_family = AF_INET;
     ser_addr.sin_port = htons(SERV_TCP_PORT);
     if ((hp = gethostbyname(host)) == NULL){
           printf("host %s not found\n", host); exit(1);
     }
     ser_addr.sin_addr.s_addr = * (u_long *) hp->h_addr;

     // create TCP socket & connect socket to server address
     sd = socket(AF_INET, SOCK_STREAM, 0);
     if (connect(sd, (struct sockaddr *) &ser_addr, sizeof(ser_addr))<0) {
          perror("client connect"); exit(1);
     }

     // fills buf1 and buf2
     for (i=0; i < BUFSIZE; ++i) {
          buf1[i]=' '; buf2[i] = ' ';
     }

     i=0;
     while (++i) {

          printf("client[%d]: how many bytes to send? ", i);
          scanf("%d", &n);

          if (n==0) {
               printf("Bye from client\n"); exit(0);
          }

          printf("client[%d]: sending %d bytes\n", i, n);
          nw=write(sd, buf1, n);
          printf("client[%d]: %d bytes sent\n", i, nw);

          printf("client[%d]: expecting %d bytes\n", i, n);
          nr=read(sd, buf2, sizeof(buf2));
          printf("client[%d]: %d bytes received\n", i, nr);
     }
}
```

```
newton[1]cli7 boole
client[1]: how many bytes to send? 10
client[1]: sending 10 bytes
client[1]: 10 bytes sent
client[1]: expecting 10 bytes
client[1]: 10 bytes received
client[2]: how many bytes to send? 100
client[2]: sending 100 bytes
client[2]: 100 bytes sent
client[2]: expecting 100 bytes
client[2]: 100 bytes received
client[3]: how many bytes to send? 1000
client[3]: sending 1000 bytes
client[3]: 1000 bytes sent
client[3]: expecting 1000 bytes
client[3]: 1000 bytes received
client[4]: how many bytes to send? 10000
client[4]: sending 10000 bytes
client[4]: 10000 bytes sent
client[4]: expecting 10000 bytes
client[4]: 1460 bytes received
client[5]: how many bytes to send? 0
Bye from client
newton[2]exit
newton[3]
```

```
boole[1]ser7
server[1]: receive 10 bytes
server[1]: 10 bytes processed
server[1]: send out 10 bytes
server[2]: receive 100 bytes
server[2]: 100 bytes processed
server[2]: send out 100 bytes
server[3]: receive 1000 bytes
server[3]: 1000 bytes processed
server[3]: send out 1000 bytes
server[4]: receive 1460 bytes
server[4]: 1460 bytes processed
server[4]: send out 1460 bytes
server[5]: receive 1460 bytes
server[5]: 1460 bytes processed
server[5]: send out 1460 bytes
server[6]: receive 1460 bytes
server[6]: 1460 bytes processed
server[6]: send out 1460 bytes
server[7]: receive 1460 bytes
server[7]: 1460 bytes processed
server[7]: send out 1460 bytes
server[8]: receive 2920 bytes
server[8]: 2920 bytes processed
server[8]: send out 2920 bytes
server[9]: receive 1240 bytes
server[9]: 1240 bytes processed
server[9]: send out 1240 bytes
boole[2]exit
boole[3]
```

26. Example 6

Reverse String Protocol

This protocol provides reverse string service. The client sends a null-terminated ASCII string to the server, and the server reverses the string and sends the reversed string back to the client. This continues until the client closes the connection. The client and server use TCP to transport messages. Each message contains a header and a null-terminated ASCII string. The message header contains a 2-byte integer value in two’s complement and in network byte order which is the total length of the string including the null-terminator at the end. The client always initiates a message, and the server always responds to the client's message with a message that contains the reversed string. A message must not exceed 1024*5 bytes in size. The server uses TCP port 40003.

```
/*
 *  stream.h      head file for stream read and stream write.
 */
```

```
#define MAX_BLOCK_SIZE (1024*5)    // maximum size of any piece of
                                   // data that can be sent by client
```

```
/*
 * purpose:  read a message from "fd" and place the string in the
 *           message to "buf".
 * pre:      1) size of buf bufsize >= MAX_BLOCK_SIZE,
 * post:     1) buf contains the byte stream;
 *           2) return value > 0   : number of bytes read
 *                           = 0   : connection closed
 *                           = -1  : read error
 *                           = -2  : protocol error
 *                           = -3  : buffer too small
 */
int readn(int fd, char *buf, int bufsize);


/*
 * purpose:  constructing a message containing "nbytes" bytes from "buf".
 *           Send the message to "fd".
 * pre:      1) nbytes <= MAX_BLOCK_SIZE,
 * post:     1) nbytes bytes from buf written to fd;
 *           2) return value = nbytes : number ofbytes written
 *                           = -3     : too many bytes to send
 *                           otherwise: write error
 */
int writen(int fd, char *buf, int nbytes);
```

```
/*
 *   stream.c
 *          routines for stream read and write.
 */

#include  <unistd.h>
#include  <sys/types.h>
#include  <netinet/in.h> // struct sockaddr_in, htons(), htonl()
#include  "stream.h"

int readn(int fd, char *buf, int bufsize)
{
    short data_size;    // sizeof (short) must be 2
    int n, nr, len;

    // check buffer size len
    if (bufsize < MAX_BLOCK_SIZE)
         return (-3);     // buffer too small

    // get the size of data sent to me
    if (read(fd, (char *) &data_size, 1) != 1) return (-1);
    if (read(fd, (char *) (&data_size)+1, 1) != 1) return (-1);
    len = (int) ntohs(data_size);  // convert to host byte order

    // read len number of bytes to buf
    for (n=0; n < len; n += nr) {
        if ((nr = read(fd, buf+n, len-n)) <= 0)
            return (nr);       // error in reading
    }
    return (len);
}

int writen(int fd, char *buf, int nbytes)
{
    short data_size = nbytes;     // short must be two bytes long
    int n, nw;

    if (nbytes > MAX_BLOCK_SIZE)
         return (-3);    // too many bytes to send in one go

    // send the data size
    data_size = htons(data_size);
    if (write(fd, (char *) &data_size, 1) != 1) return (-1);
    if (write(fd, (char *) (&data_size)+1, 1) != 1) return (-1);

    // send nbytes
    for (n=0; n<nbytes; n += nw) {
        if ((nw = write(fd, buf+n, nbytes-n)) <= 0)
            return (nw);    // write error
    }
    return (n);
}
```

```
/*
 *  ser6.c        An improved version of "ser5.c". This version
 *                handles the message boundaries which are not
 *                preserved by the TCP. Each message transmitted
 *                between the client and the server is preceeded
 *                by a two byte value which is the length of the
 *                message. The handling of the message length is
 *                done in routines readn and writen.
*/
#include  <unistd.h>
#include  <sys/stat.h>
#include  <string.h>     // strlen(), strcmp() etc
#include  <errno.h>      // extern int errno, EINTR, perror()
#include  <signal.h>     // SIGCHLD, sigaction()
#include  <syslog.h>
#include  <sys/types.h>  // pid_t, u_long, u_short
#include  <sys/socket.h> // struct sockaddr, socket(), etc
#include  <sys/wait.h>   // waitpid(), WNOHAND
#include  <netinet/in.h> // struct sockaddr_in, htons(), htonl()
                         // and INADDR_ANY
#include  "stream.h"     // MAX_BLOCK_SIZE, readn(), writen()

#define   SERV_TCP_PORT   40003           // server port no

void claim_children()
{
     pid_t pid=1;

     while (pid>0) { // claim as many zombies as we can
         pid = waitpid(0, (int *)0, WNOHANG);
     }
}

void daemon_init(void)
{    pid_t   pid;
     struct sigaction act;

     if ( (pid = fork()) < 0) {
          perror("fork"); exit(1);
     } else if (pid > 0)
          exit(0);                    // parent goes bye-bye

     // child continues
     setsid();                        // become session leader
     chdir("/");                      // change working directory
     umask(0);                        // clear file mode creation mask

     // catch SIGCHLD to remove zombies from system
     act.sa_handler = claim_children; // use reliable signal
     sigemptyset(&act.sa_mask);       // not to block other signals
     act.sa_flags   = SA_NOCLDSTOP;   // not catch stopped children
     sigaction(SIGCHLD,(struct sigaction *)&act,(struct sigaction *)0);
     /* note: a less than perfect method is to use
              signal(SIGCHLD, claim_children);
     */
}
```

```
void reverse(char *s)
{
    char c;
    int i, j;

    for (i=0, j = strlen(s)-1; i<j; i++, j--) {
        c = s[i]; s[i] = s[j]; s[j] = c;
    }
}

void serve_a_client(int sd)
{   int nr, nw;
    char buf[MAX_BLOCK_SIZE];

    while (1){
         // read a message from client
         if ((nr = readn(sd, buf, sizeof(buf))) <= 0)
             return;   // connection broken down

         // process the message
         buf[nr] = '\0'; reverse(buf);

         // send the processed message to client
         nw = writen(sd, buf, nr);
    }
}

int main()
{
     int sd, nsd, n, cli_addrlen;  pid_t pid;
     struct sockaddr_in ser_addr, cli_addr;

     // turn the process into a daemon
     daemon_init();

     // set up listening socket sd
     if ((sd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
           perror("server:socket"); exit(1);
     }

     // build server listening socket address
     bzero((char *)&ser_addr, sizeof(ser_addr));
     ser_addr.sin_family = AF_INET;
     ser_addr.sin_port = htons(SERV_TCP_PORT);
     ser_addr.sin_addr.s_addr = htonl(INADDR_ANY);
     /* note: accept client request sent to any one of the
        network interface(s) on this host.
     */

     // bind server address to socket sd
     if (bind(sd, (struct sockaddr *) &ser_addr, sizeof(ser_addr))<0){
           perror("server bind"); exit(1);
     }

     // become a listening socket
     listen(sd, 5);
     while (1) {

          // wait to accept a client request for connection
          cli_addrlen = sizeof(cli_addr);
          nsd = accept(sd, (struct sockaddr *) &cli_addr,
                 (socklen_t *) &cli_addrlen);
          if (nsd < 0) {
               if (errno == EINTR)   // if interrupted by SIGCHLD
                    continue;
               perror("server:accept"); exit(1);
          }

          // create a child process to serve this client
          if ((pid=fork()) <0) {
              perror("fork"); exit(1);
          } else if (pid > 0) {
              close(nsd);
              continue;    // parent to wait for next client
          }

          // now in child, serve the current client
          close(sd);
          serve_a_client(nsd);
          exit(0);
     }
}
```

```
/* cli6.c         An improved version of "cli5.c". Since TCP does
 *                not preserve the message boundaries, each message
 *                is preceeded by a two byte value which is the length
 *                of the message.
 */
#include  <stdlib.h>
#include  <stdio.h>
#include  <sys/types.h>
#include  <sys/socket.h>
#include  <netinet/in.h>       // struct sockaddr_in, htons, htonl
#include  <netdb.h>            // struct hostent, gethostbyname()
#include  <string.h>
#include  <stdio.h>
#include  "stream.h"           // MAX_BLOCK_SIZE, readn(), writen()

#define   SERV_TCP_PORT  40003 // server's "well-known" port number

int main(int argc, char *argv[])
{
     int sd, n, nr, nw, i=0;
     char buf[MAX_BLOCK_SIZE], host[60];
     struct sockaddr_in ser_addr; struct hostent *hp;

     // get server host name
     if (argc==1)  // assume server running on the local host
         gethostname(host, sizeof(host));
     else if (argc == 2) // use the given host name
         strcpy(host, argv[1]);
     else {
         printf("Usage: %s [<server_host_name>]\n", argv[0]); exit(1);
     }

     // get host address, & build a server socket address
     bzero((char *) &ser_addr, sizeof(ser_addr));
     ser_addr.sin_family = AF_INET;
     ser_addr.sin_port = htons(SERV_TCP_PORT);
     if ((hp = gethostbyname(host)) == NULL){
           printf("host %s not found\n", host); exit(1);
     }
     ser_addr.sin_addr.s_addr = * (u_long *) hp->h_addr;

     // create TCP socket & connect socket to server address
     sd = socket(AF_INET, SOCK_STREAM, 0);
     if (connect(sd, (struct sockaddr *) &ser_addr, sizeof(ser_addr))<0) {
          perror("client connect"); exit(1);
     }

     while (++i) {
          printf("Client Input[%d]: ", i);
          fgets(buf, sizeof(buf), stdin); nr = strlen(buf);
          if (buf[nr-1] == '\n') { buf[nr-1] = '\0'; --nr; }

          if (strcmp(buf, "quit")==0) {
               printf("Bye from client\n"); exit(0);
          }

          if (nr > 0) {
               if ((nw=writen(sd, buf, nr)) < nr) {
                    printf("client: send error\n"); exit(1);
               }
               if ((nr=readn(sd, buf, sizeof(buf))) <=0) {
                    printf("client: receive error\n"); exit(1);
               }
               buf[nr] = '\0';
               printf("Sever Output[%d]: %s\n", i, buf);
          }
     }
}
```

27. Error Recovery in UDP

Since UDP is not reliable, messages can get lost during the transmission. Programs using UDP to deliver messages may need to handle this type of errors.

To send a message and receive its reply, a sender usually follows these two steps:

```
    sendto(sd, message, ......);
    recv(sd, reply, ......);
```

The `recv` call may be blocked forever if the reply is not received, either because the sender failed to deliver the message to its recipient, or because the recipient failed to deliver its reply to the sender.

There are many methods to handle this type of errors. One such method is to set waiting time for each `recv` operation and re-sends the message that *may* have lost.

The waiting time is doubled each time the message is re-sent until the number of retrials reaches a pre-determined up limit.

If the message was not lost, the same reply could be received more than once. Therefore, the sender process should handle potential duplications of the same reply.

The algorithm below gives the main thrust of the solution:

```
interval = 1;
alarm_expired = 0;
signal(SIGALRM, time_expired); // the signal handler would set
                               // alarm_expired to 1

......

for (retry=0; retry<MAX_RETRIES; retry++) {
    sendto(sd, message, ...);

    alarm(interval);
    nr=recv(sd, reply, ...);

    if (nr<0)
        if (alarm_expired) {
            alarm_expired=0;
            interval *= 2;   // double the waiting time
        } else {
            perror("recv");
            exit(1);
        }
    else
        if (reply is the right reply)
            break;
        else
            discard the reply
}
```

Example 8 implements the above algorithm. In this example, the client sends a message to the server and then waits for the processed message (reversed message) from the server. If the client does not receive the processed message within a pre-set time, it assumes that the message has lost, so it re-sends the same message to the server. When the client receives a reply from the server, it checks to see if the reply has already been received. If the answer is yes, it simply discards the reply.

To allow the client to determine whether a message is a duplicate, each message carries a message-id. When the server replies to the client, it includes the message-id in the reply. If the client finds a reply from the server with the same message-id as a previous one, it knows the current message has already been received.

```
/*
 *  ser8.c  An UDP version of reverse string service (server part).
 */
#include  <unistd.h>
#include  <stdio.h>
#include  <stdlib.h>
#include  <sys/stat.h>
#include  <strings.h>
#include  <errno.h>      // extern int errno, EINTR, perror()
#include  <sys/types.h>  // pid_t, u_long, u_short
#include  <sys/socket.h> // struct sockaddr, socket(), etc
#include  <netinet/in.h> // struct sockaddr_in, htons(), htonl()
                         // and INADDR_ANY

#define   BUFSIZE         256
#define   SERV_UDP_PORT   40001           // server port no

void daemon_init(void)
{
     pid_t   pid;

     if ( (pid = fork()) < 0) {
          perror("fork"); exit(1);
     } else if (pid > 0)
          exit(0);                  // parent goes bye-bye

     // child continues
     setsid();                      // become session leader
     chdir("/");                    // change working directory
     umask(0);                      // clear file mode creation mask
}


void reverse(char *s, int len)
{
    char c;
    int i, j;

    for (i=0, j = len-1; i<j; i++, j--) {
        c = s[i]; s[i] = s[j]; s[j] = c;
    }
}

int main()
{
     int sd, cli_addr_len, n;  char buf[BUFSIZE];
     struct sockaddr_in ser_addr, cli_addr;

     // turn the program into a daemon
     daemon_init();

     // set up server socket sd
     if ((sd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
           perror("server:socket"); exit(1);
     }

     // build server Internet socket address
     bzero((char *)&ser_addr, sizeof(ser_addr));
     ser_addr.sin_family = AF_INET;
     ser_addr.sin_port = htons(SERV_UDP_PORT);
     ser_addr.sin_addr.s_addr = htonl(INADDR_ANY);
     /* note: accept client request sent to any one of the
        network interface(s) on this host.
     */

     // bind server address to socket sd
     if (bind(sd, (struct sockaddr *) &ser_addr, sizeof(ser_addr))<0){
           perror("server bind"); exit(1);
     }

     while (1) {

         // wait for a client request
         cli_addr_len = sizeof(cli_addr);
         n = recvfrom(sd, buf, sizeof(buf), 0,
                    (struct sockaddr *)&cli_addr,
                  (socklen_t *)&cli_addr_len);
         if (n <= 0) exit(1);   // receive error

         // process the client message
         reverse(buf+4, n-4);   // do not change the message id */

         // send the reversed message to that client
         n = sendto(sd, buf, n, 0, (struct sockaddr *) &cli_addr,
                                     (socklen_t) cli_addr_len);
         if (n <= 0) exit(1);    // send error
    }
}
```

When the client sends a message to the server, the message consists of a message header and a message body. The message header is 4-byte long, including a 2-byte message number and a 2-byte retry number, as shown in the following diagram:

0	2	4
message_no	retry_no	message body

The server does not change the message header. It includes the message header in its reply.

```
/*
 * cli8.c   An UDP version of string reverse (client part).
 */
#include  <stdlib.h>
#include  <string.h>
#include  <stdio.h>
#include  <signal.h>
#include  <unistd.h>
#include  <netdb.h>            // struct hostent, gethostbyname()
#include  <sys/types.h>
#include  <sys/socket.h>
#include  <netinet/in.h>       // struct sockaddr_in, htons, htonl

#define   BUFSIZE        256
#define   SERV_UDP_PORT  40001 // server's "well-known" port number
#define   MAX_RETRY      10
#define   INIT_INTERVAL  1
int  alarm_expired = 0;

void alarm_expire()
{
     alarm_expired = 1;
}

int main(int argc, char *argv[])
{
     unsigned short message_no, retry_no, retry, i=0;
     int sd, n, nr, nw, interval, ser_addr_len;
     char buf1[BUFSIZE], buf2[BUFSIZE], host[100];
     char *mesg = buf1+4;
     struct sockaddr_in ser_addr; struct hostent *hp;

     // get server host name
     if (argc==1)  // assume server running on the local host
         gethostname(host, sizeof(host));
     else if (argc == 2) // use the given host name
         strcpy(host, argv[1]);
     else {
         printf("Usage: %s [<server_host_name>]\n", argv[0]); exit(1);
     }

     // get host address, & build a server socket address
     bzero((char *) &ser_addr, sizeof(ser_addr));
     ser_addr.sin_family = AF_INET;
     ser_addr.sin_port = htons(SERV_UDP_PORT);
     if ((hp = gethostbyname(host)) == NULL){
           printf("host %s not found\n", host); exit(1);
     }
     ser_addr.sin_addr.s_addr = * (u_long *) hp->h_addr;
     ser_addr_len = sizeof(ser_addr);

     // create UDP socket
     sd = socket(AF_INET, SOCK_DGRAM, 0);

     // set alarm clock handler
     signal(SIGALRM, alarm_expire);
```

```
    while (++i) {
        // get a message from user, place it in buf1+4
        printf("\nClient Input[%d]             :", i);
        fgets(mesg, BUFSIZE-5, stdin); nr = strlen(mesg);
        if (mesg[nr-1] == '\n') { mesg[nr-1] = '\0'; --nr; }
        if (nr <=  0) continue;

        if (strcmp(mesg, "quit")==0) {
            printf("Bye from client\n"); exit(0);
        }

        interval = INIT_INTERVAL;    // initial waiting time
        message_no = htons(i);       // add the header
        bcopy((char *) &message_no, buf1, 2);
        n = nr+4;    // total message length, including the header

        // try to send the message i in buf1 and get its reply
        for (retry=0; retry < MAX_RETRY; retry++) {

            if (retry) printf("Client Retry No. %d ...\n", retry);

            // add retry no to header
            retry_no = htons(retry);
            bcopy((char *)&retry_no, buf1+2, 2);

            // send the message id and message body to server
            nw = sendto(sd, buf1, n, 0,
                 (struct sockaddr *) &ser_addr, ser_addr_len);

            alarm (interval);   // set alarm clock

            nr = recv(sd, buf2, BUFSIZE, 0);
            if (nr <=0) {
                 if (alarm_expired) {
                      alarm_expired = 0; interval *= 2;
                 } else {
                      perror("recv"); close(sd); exit(1);
                 }
            } else {  // a reply is received
                 buf2[nr] = '\0'; alarm(0);
                 message_no = ntohs( * (short *) buf2 );
                 retry_no = ntohs( * (short *) (buf2+2) );
                 if (message_no == i) break; // got its reply
                 printf("Client: discard message[%d]: %s\n",
                         message_no, buf2+4);
            }
        }

        if (retry < MAX_RETRY)
            printf("Sever Output[%d] (retry_no=%d): %s\n",
                    i, retry_no,  buf2+4);
        else {
            printf("client: message[%d] lost\n", i);
            close(sd); exit(1);
        }
    }
}
```

Test output:

```
newton[1]cli8.osf client40.comlab.ox.ac.uk

Client Input[1]              :hello!
Sever Output[1] (retry_no=0): !olleH

Client Input[2]              :How dod you do today?
Client Retry No. 1 ...
Client Retry No. 2 ...
Sever Output[2] (retry_no=2): ?yadot od uoy dod woH

Client Input[3]              :How is the weather tomorrow?
Sever Output[3] (retry_no=0): ?worromot rehtaew eht si woH

Client Input[4]              :Will you to go to the football?
Client Retry No. 1 ...
Client Retry No. 2 ...
Client Retry No. 3 ...
Sever Output[4] (retry_no=3): ?llabtoof eht ot og uoy lliW

Client Input[5]              :How is the traffic in the freeway?
Sever Output[5] (retry_no=0): ?yaweerf eht ni ciffart eht si woH

Client Input[6]              :let's go home.
Sever Output[6] (retry_no=0): .emoh og s'tel

Client Input[7]              :quit
Bye from client
newton[2]exit
newton[3]
```