13 TCP/IP and Network Programming

Abstract

This chapter covers TCP/IP and network programming. The chapter consists of two parts. The first part covers the TCP/IP protocol and its applications. These include the TCP/IP stack, IP address, hostname, DNS, IP data packets and routers. It describes the UDP and TCP protocols, port number and data flow in TCP/IP networks. It explains the server-client computing model and the socket programming interface. It demonstrates network programming by examples using both UDP and TCP sockets. The first programming project is to implement a pair of TCP server-client to do file operations across the Internet. It allows the user to define additional communication protocols for reliable transfer of file contents.

The second part of the chapter covers Web and CGI programming. It explains the HTTP programming model, Web pages and Web browsers. It shows how to configure the Linux HTTPD server to support user Web pages, PHP and CGI programming. It explains both client side and server side dynamic Web pages. It shows how to create server side dynamic Web pages by both PHP and CGI. The second programming project is for the reader to implement server-side dynamic Web pages by CGI programming on a Linux HTTPD server machine.

13.1 Introduction to Network Programming

Nowadays, accessing the Internet has become a necessity in daily life. While most people may only use the Internet as a tool for information gathering, on-line shopping and social media, etc. computer science students must have some understanding of the Internet technology, as well as some skills in network programming. In this chapter, we shall cover the basics of TCP/IP networks and network programming. These include the TCP/IP protocol, UDP and TCP protocols, server-client computing, HTTP and Web pages, PHP and CGI programming for dynamic Web pages.

13.2 TCP/IP Protocol

TCP/IP (Comer 1988, 2001; RFC1180 1991) is the backbone of Internet. TCP stands for Transmission Control Protocol. IP stands for Internet Protocol. Currently there are 2 versions of IPs, known as IPv4

© Springer International Publishing AG, part of Springer Nature 2018

K. C. Wang, *Systems Programming in Unix/Linux*, https://doi.org/10.1007/978-3-319-92429-8_13

and IPv6. IPv4 uses 32-bit address and IPv6 uses 128-bit address. The discussion here is based on IPv4, which is still the predominant version of IP in use. TCP/IP is organized in several layers, usually referred to as the **TCP/IP stack**. Figure 13.1 shows the TCP/IP layers, representative components in each layer and their functions.

At the top layer are applications which use TCP/IP. Application such as ssh for login to remote hosts, mail for exchanging e-mails, http for Web pages, etc. require reliable data transfer. Such applications typically use TCP in the transport layer. On the other hand, there are applications, e.g. the **ping** command, which is used to query other hosts, do not need reliability. Such applications may use UDP (RFC 768 1980; Comer 1988) in the transport layer for better efficiency. The transport layer is responsible for send/receive application data as packets to/from IP hosts. Data transfer at or above the transport layer between processes and hosts are only logical. Actual data transfer occurs in the **internet** (IP) and Link layers, which divide data packets into data frames for transmission across physical networks. Figure 13.2 shows the data flow paths in TCP/IP networks.

Fig. 13.1 TCP/IP layers

Layer	Components	Functions
Application Layer	ssh ping	Application commands
Transport Layer	TCP UCP	Connection Datagram
Internet Layer	IP	send/receive data frames
Link Layer	Ethernet	send/receive data frames

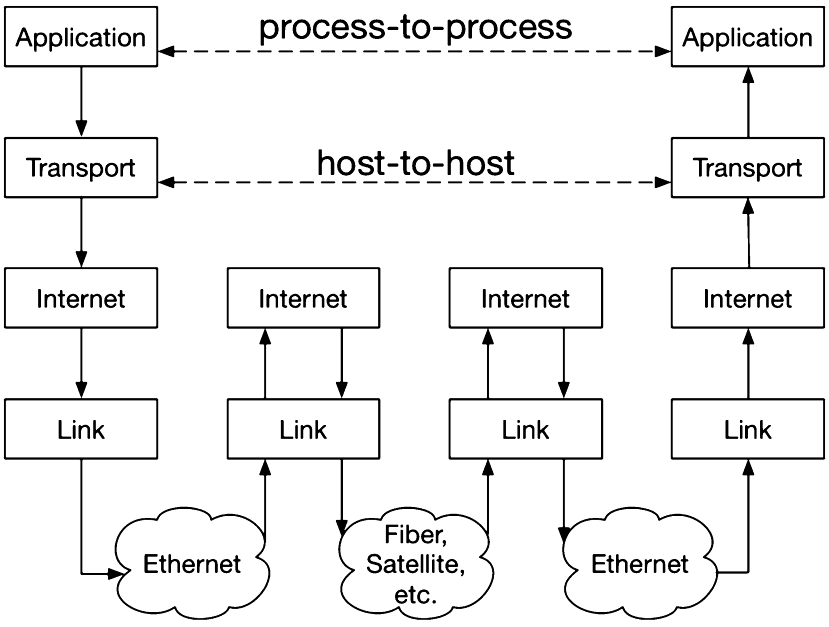


Fig. 13.2 Data flow paths in TCP/IP networks

13.3 IP Host and IP address

A host is a computer or device that supports the TCP/IP protocol. Every host is identified by a 32-bit number called the **IP address**. For convenience, the 32-bit IP address number is usually expressed in a dot notation, e.g.134.121.64.1, in which the individual bytes are separated by dots. A host is also known by a **host name**, e.g. dns1.eecs.wsu.edu. In practice, applications usually use host names rather than IP addresses. Host name and IP address are equivalent in the sense that, given one, we can find the other by **DNS** (Domain Name System) (RFC 134 1987; RFC 1035 1987) servers, which translate IP address into host name and vice versa.

An IP address is divided into two parts: a NetworkID field and a HostID field. Depending on the division, IP addresses are classified into classes A to E. For example, a class B IP address is divided into a 16-bit NetworkID, in which the leading 2 bits are 10, followed by a 16-bit HostID field. Data packets intended for an IP address are first sent to a **router** with the same networkID. The router will forward the packets to a specific host in that network by HostID. Every host has a local host name **localhost** with a default IP address **127.0.0.1.** The Link layer of localhost is a loop-back virtual device, which routes every data packet back to the same localhost. This special feature allows us to run TCP/IP applications on the same computer without actually connecting to the Internet.

13.4 IP Protocol

IP is a protocol for sending/receiving data packets between IP hosts. IP operates in a best-effort manner. An IP host just sends data packets to a receiving host, but it does not guarantee the data packets will be delivered to their destinations, nor in order. This means that IP is not a reliable protocol. If needed, reliability must be implemented above the IP layer.

13.5 IP Packet Format

An IP packet consists of an IP header, sender and receiver IP addresses, followed by data. The maximum size of each IP packet is 64 KB. The IP header contains more information about the data packet, e.g. total length, whether the packet uses TCP or UDP, time-to-live (TTL) count, check-sum for error detection, etc. Figure 13.3 shows the IP header format.

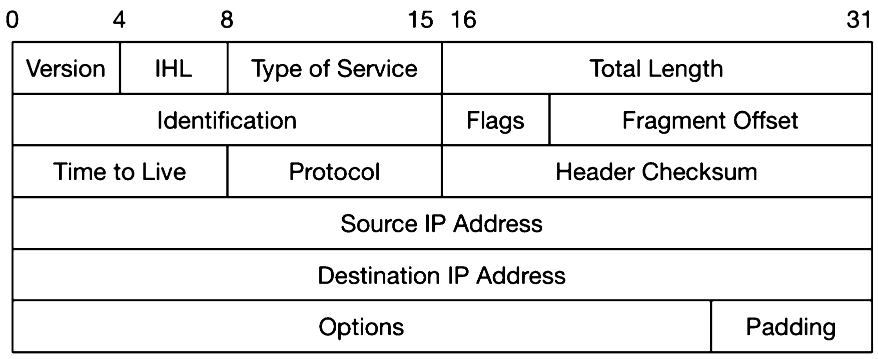


Fig. 13.3 IP header format

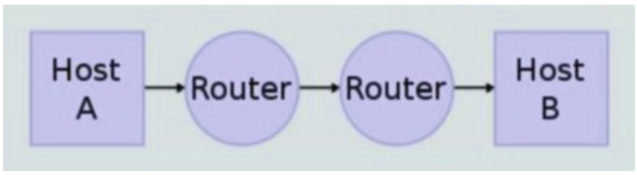


Fig. 13.4 TCP/IP network topology

13.6 Routers

IP hosts may be located far apart. It is usually not possible to send data packets from one host to the other directly. Routers are special IP hosts which receive and forward packets. An IP packet may go through many routers, or hops, before arriving at a destination, if at all. Figure 13.4 shows the topology of a TCP/IP network.

Each IP packet has an 8-bit Time-To-Live (TTL) count in the IP header, which has a maximum value of 255. At each router, the TTL is reduced by 1. If the TTL reduces to 0 and the packet still has not reached the destination, it is simply dropped. This prevents any packets from circulating around indefinitely in the IP network.

13.7 UDP User Datagram Protocol

UDP (RFC 768 1980; Comer 1988) operates on top of IP. It is used to send/receive datagrams. Like IP, UDP does not guarantee reliability but it is fast and efficient. It is used in situations where reliability is not essential. For example, a user may use the ping command to probe a target host, as in

ping hostname OR ping IPaddress

Ping is an application program which sends a UDP packet with a timestamp to a target host. Upon receiving a pinging packet, the target host echoes the UDP packet with a timestamp back to the sender, allowing the sender to compute and display the round-trip time. If the target host does not exist or is down, the pinging UDP packets would be dropped by routers when its TTL reduces to 0. In that case, the user would notice the absence of any response from the target host. The user may either try to ping it again or conclude that the target host is down. In this case, using UDP is preferred since reliability is not essential.

13.8 TCP Transmission Control Protocol

TCP is a connection-oriented protocol for sending/receiving streams of data. TCP also operates on top of IP but it guarantees reliable data transfers. A common analogy is that UDP is similar to USPS of sending mails and TCP is similar to telephone connections.

13.9 Port Number

At each host, many applications (processes) may be using TCP/UDP at the same time. Each application is uniquely identified by a triple

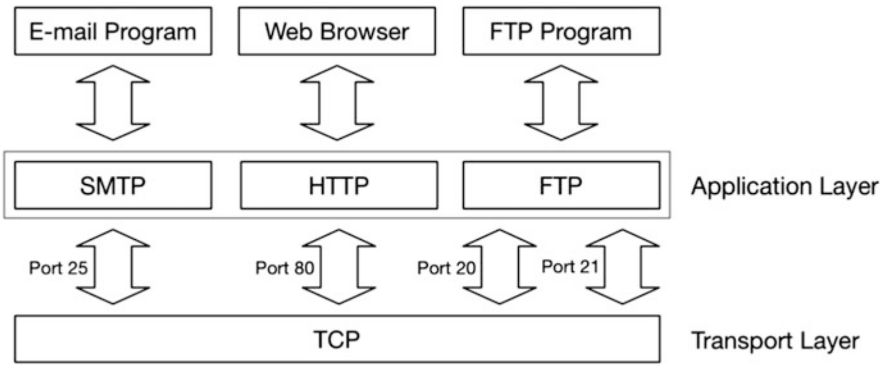


Fig. 13.5 Applications using TCP

Application = (HostIP, Protocol, PortNumber)

where Protocol is either TCP or UDP, PortNumber is a unique unsigned short integer assigned to the application. In order to use UDP or TCP, an application (process) must choose or obtain a PortNumber first. The first 1024 port numbers are reserved. Other port numbers are available for general use. An application may either choose an available port number or let the OS kernel assign a port number. Figure 13.5 shows some of the applications that use TCP in the transport layer and their default port numbers.

13.10 Network and Host Byte Orders

Computers may use either big-endian or little-endian byte order. On the Internet, data are always in network order, which is big-endian. For little-endian machines, such as Intel x86 based PCs, a set of library functions, htons(), htonl(), ntohs(), ntohl(), are available for converting data between host order and network order. For example, a port number 1234 in a PC is an unsigned short integer in host (little-endian) byte order. It must be converted to network order by htons(1234) before using. Conversely, a port number received from the Internet must be converted to host order by ntohs(port).

13.11 Data Flow in TCP/IP Networks

Figure 13.6 shows the data formats of the various layers in a TCP/IP network. It also shows the data flow paths among the different layers.

In Fig. 13.6, data from application layer is passed to the transport layer, which adds a TCP or UDP header to identify the transport protocol used. The combined data is passed on to the IP internet layer, which adds an IP header containing IP addresses to identify the sending and receiving hosts. The combined data is then passed on to the Network Link layer, which divides the data into frames, adding the addresses of the sending and receiving networks for transmission across physical networks. Mapping of IP addresses to network addresses is performed by the Address Resolution Protocol (**ARP**) (ARP 1982). At the receiving end, the data encoding process is reversed. Each layer unpacks the received data by stripping off the headers, reassembles them and delivers the data to a layer above. The original data of an application at a sending host is ultimately delivered to the corresponding application at the receiving host.

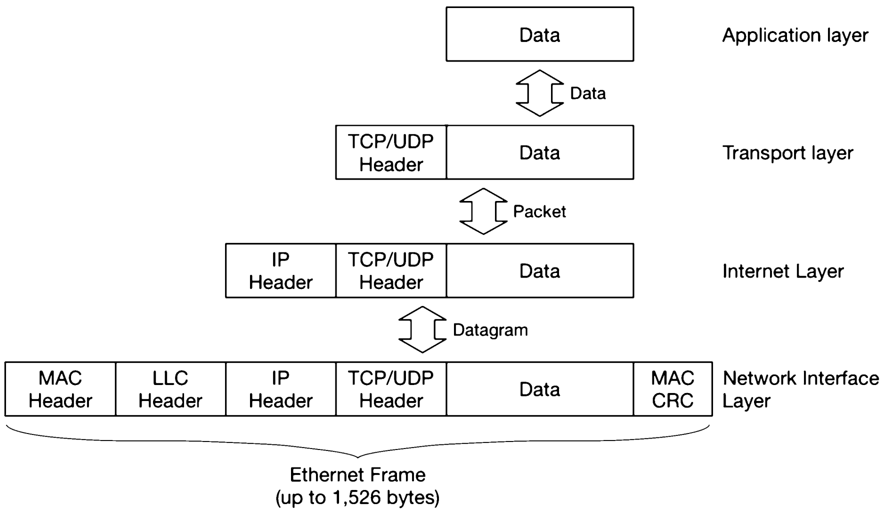


Fig. 13.6 Data format in TCP/IP layers

13.12 Network Programming

All Unix/Linux systems provide TCP/IP support for network programming. In this section, we shall explain the platforms and the server-client computing model for network programming.

13.12.1 Network Programming Platforms

In order to do network programming, the reader must have access to a platform that supports network programming. Such a platform is available in several ways.

(1). User Account on Server Machines: Nowadays, almost all educational institutions provide network access, often in the form of wireless connections, to their faculty, staff and students. Every member of the institution should be able to login to a server machine to access the Internet. Whether the server machine allows general network programming depends on the policy of the local network administration. Here, we describe the setup of a network programming platform at the author's institution, EECS of Washington State University. The author maintains a private server machine

cs360.eecs.wsu.edu

The server machine runs Slackware Linux version 14.2, which comes with a full compliment of support for network programming. The server machine is registered with the DNS server of EECS of WSU. When the server machine boots up, it uses **DHCP** (Dynamic Host Configuration Protocol) to obtain a private IP address from the DHCP server (RFC 2131 1997). Although not a public IP address, it can be accessed on the Internet through **NAT** (Network Address Translation). Then the author creates user accounts for students in the CS360 class for them to login. Students typically connect their laptops to the Internet via the WSU wireless network. Once on the Internet, they can login to the cs360 server machine.

(2). Standalone PCs or Laptops: Even if the reader does not have access to a server machine, it is still possible to do network programming on a standalone computer by using the localhost of the computer. In this case, the reader may have to download and install some of the network components. For example, Ubuntu Linux users may have to install and configure the Apache server for HTTP and CGI programming, which will be described later in Sect. 13.17.

13.12.2 Server-Client Computing Model

Most network programming tasks are based on the **Server-Client** computing model. In the Server-Client computing model, we run a server process at a server host first. Then we run a client from a client host. In UDP, the server waits for datagram from a client, processes the datagram and generates a response to the client. In TCP, the server waits for a client to connect. The client first connects to the server to establish a virtual circuit between the client and the server. After the connection is made, the server and client can exchange continuous streams of data. In the following, we shall show how to do network programming using both UDP and TCP.

13.13 Socket Programming

In network programming, user interface to TCP/IP is through a set of C library functions and system calls, which are collectively known as the **sockets API** (Rago 1993; Stevens et al. 2004). In order to use the socket API, we need the socket address structure, which is used to identify both the server and the client. The socket address structure is defined in netdb.h and sys/socket.h.

13.13.1 Socket Address

```
struct sockaddr_in {
       sa_family_t sin_family;  // AF_INET for TCP/IP
       in_port_t   sin_port;    // port number
       struct in_addr sin_addr; // IP address
};
struct in_addr {                // internet address
       uint32_t    s_addr;      // IP address in network byte order
};
```

In the socket address structure,
sin_family is always set to **AF_INET** for TCP/IP networks.
sin_port contains the port number in network byte order.
sin_addr is the host IP address in network byte order.

13.13.2 The Socket API

A server must create a socket and bind it with a sockaddr containing the server's IP address and port number. It may use either a fixed port number, or let the OS kernel choose a port number if sin_port is 0. In order to communicate with a server, a client must create a socket. For UPD sockets, binding the socket to a server address is optional. If the socket is not bound to any specific server, it must provide a socket address containing the IP and port number of the server in subsequent sendto()/recvfrom() calls. The following shows the socket() system call, which creates a socket and returns a file descriptor

(1). int socket(int domain, int type, int protocol)
Examples:

int udp_sock = socket(AF_INET, SOCK_DGRAM, 0);

This creates a socket for sending/receiving UDP datagrams.

int tcp_sock = socket(AF_INET, SOCK_STREAM, 0);

This creates a connect-oriented TCP socket for sending/receiving data streams.
A newly created socket does not have any associated address. It must be bound with a host address and port number to identify either the receiving host or the sending host. This is done by the bind() system call.

(2). int bind(int sockfd**, struct sockaddr** *addr**, socklen_t** addrlen**);**
The bind() system call assigns the address specified by addr to the socket referred to by the file descriptor sockfd, addrlen specifies the size in bytes of the address structure pointed to by addr. For a UDP socket intended for contacting other UDP server hosts, it must be bound to the address of the client, allowing the server to send reply back. For a TCP socket intended for accepting client connections, it must be bound to the server host address first.

(3). UDP Sockets
UDP sockets use sendto()/recvfrom() to send/receive datagrams.

```
ssize_t sendto(int sockfd, const void *buf, size_t len, int flags,
               const struct sockaddr *dest_addr, socklen_t addrlen);

ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags,
                 struct sockaddr *src_addr, socklen_t *addrlen);
```

sendto() sends len bytes of data in buf to a destination host identified by dest_addr, which contains the destination host IP and port number. Recvfrom() receives data from a client host. In addition to data, it also fills src_addr with client's IP and port number, allowing the server to send reply back to the client.

(4). TCP Sockets
After creating a socket and binding it to the server address, a TCP server uses **listen**() and **accept**() to accept connections from clients

int listen(int sockfd, int backlog);

listen() marks the socket referred to by sockfd as a socket that will be used to accept incoming connection The backlog argument defines the maximum queue length of pending connections.

int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);

The accept() system call is used with connection-based sockets. It extracts the first connection request on the queue of pending connections for the listening socket, sockfd, creates a new connected socket, and returns a new file descriptor referring to that socket., which is connected with the client host. When executing the accept() system call, the TCP server blocks until a client has made a connection through connect().

int connect(int sockfd**, const struct sockaddr** *addr**, socklen_t** addrlen**);**

The connect() system call connects the socket referred to by the file descriptor sockfd to the address specified by addr, and the addrlen argument specifies the size of addr. The format of the address in addr is determined by the address space of the socket sockfd;

If the socket sockfd is of type **SOCK_DGRAM**, i.e. UDP socket, addr is the address to which datagrams are sent by default, and the only address from which datagrams are received. This restricts a UDP socket to communicate with a specific UDP host, but it is rarely used in practice. So for UDP sockets, connection is either optional or unnecessary. If the socket is of type **SOCK_STREAM,** i.e. TCP socket, the connect() call attempts to make a connection to the socket that is bound to the address specified by addr.

(5). send()/read() and recv/write()

After establishing a connection, both TCP hosts may use send() /write() to send data, and recv()/ read() to receive data. Their only difference is the flag parameter in send() and recv(), which can be set to 0 for simple cases.

```
ssize_t send(int sockfd, const void *buf, size_t len, int flags);
ssize_t write(sockfd, void *buf, size_t, len)

ssize_t recv(int sockfd, void *buf, size_t len, int flags);
ssize_t read(sockfd, void *buf, size_t len);
```

13.14 UDP Echo Server-Client Program

In this section, we show a simple echo server/client program using UDP. For ease of reference, the program is denoted by C13.1. The following chart lists the algorithms of the server and client.

```
---------- UDP Server ---------------- UDP Client -------------
1. create a UDP socket        | 1. create a UDP socket
2. set addr = server [IP,port]| 2. set addr = server [IP,port]
3. bind socket to addr        |    while(1){
   while(1){                  | 3.   ask user for an input line
4.   recvfrom() from client   | 4.   sendto() line to server
5.   sendto() reply to client | 5.   recvfrom() reply from server
   }                          |    }
-----------------------------------------------------------------
```

For simplicity, we assume that both the server and client run on the same computer. The server runs on the default localhost with IP = 127.0.0.1, and it uses a fixed port number 1234. This simplifies the program code. It also allows the reader to test run the server and client programs in different xterms on the same computer. The following shows the server and client program code using UDP.

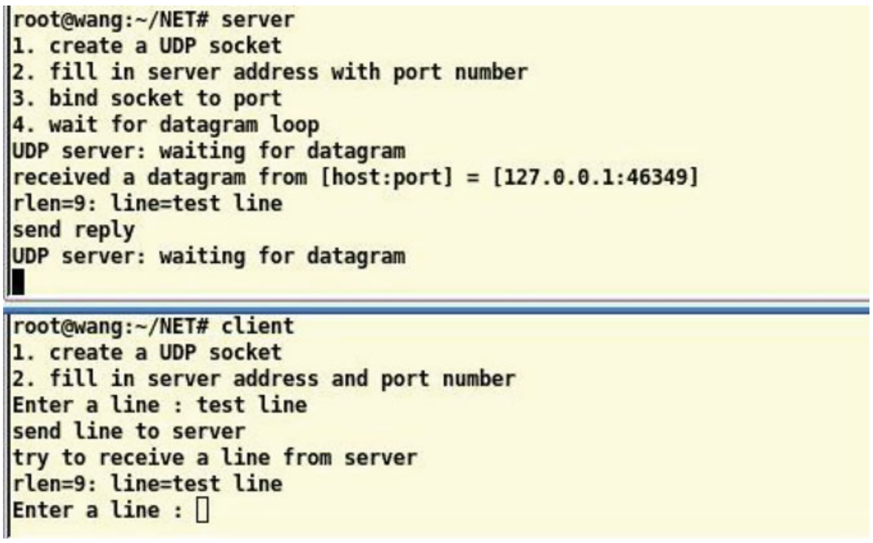
```
/*  C13.1.a: UDP server.c file */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/ip.h>

#define BUFLEN  256  // max length of buffer
#define PORT    1234  // fixed server port number

char line[BUFLEN];
struct sockaddr_in me, client;
int sock, rlen, clen = sizeof(client);

int main()
{
  printf("1. create a UDP socket\n");
  sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);

  printf("2. fill me with server address and port number\n");
  memset((char *)&me, 0, sizeof(me));
  me.sin_family = AF_INET;
  me.sin_port = htons(PORT);
  me.sin_addr.s_addr = htonl(INADDR_ANY); // use localhost

  printf("3. bind socket to server IP and port\n");
  bind(sock, (struct sockaddr*)&me, sizeof(me));

  printf("4. wait for datagram\n");
  while(1){
     memset(line, 0, BUFLEN);
     printf("UDP server: waiting for datagram\n");
     // recvfrom() gets client IP, port in sockaddr_in clinet
     rlen=recvfrom(sock,line,BUFLEN,0,(struct sockaddr *)&client,&clen);
     printf("received a datagram from [host:port] = [%s:%d]\n",
             inet_ntoa(client.sin_addr), ntohs(client.sin_port));
     printf("rlen=%d: line=%s\n", rlen, line);
     printf("send reply\n");
     sendto(sock, line, rlen, 0, (struct sockaddr*)&client, clen);
  }
}
```

```
/***** C13.1.b: UDP client.c file *****/
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<sys/socket.h>
#include <netinet/ip.h>

#define SERVER_HOST "127.0.0.1" // default server IP: localhost
#define SERVER_PORT  1234       // fixed server port number
#define BUFLEN        256       // max length of buffer

char line[BUFLEN];
struct sockaddr_in server;
int sock, rlen, slen=sizeof(server);

int main()
{
  printf("1. create a UDP socket\n");
  sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);

  printf("2. fill in server address and port number\n");
  memset((char *) &server, 0, sizeof(server));
  other.sin_family = AF_INET;
  other.sin_port = htons(SERVER_PORT);
  inet_aton(SERVER_HOST, &server.sin_addr);

  while(1){
     printf("Enter a line : ");
     fgets(line, BUFLEN, stdin);
     line[strlen(line)-1] = 0;
     printf("send line to server\n");
     sendto(sock,line,strlen(line),0,(struct sockaddr *)&server,slen);
     memset(line, 0, BUFLEN);
     printf("try to receive a line from server\n");
     rlen=recvfrom(sock,line,BUFLEN,0,(struct sockaddr*)&server,&slen);
     printf("rlen=%d: line=%s\n", rlen, line);
  }
}
```

Figure 13.7 shows the sample outputs of running the UDP server-client program C13.1.

13.15 TCP Echo Server-Client Program

This section presents a simple echo server-client program using TCP. The program is denoted by C13.2. For simplicity, we assume that both the server and client run on the same computer, and the server port number is hard coded as 1234. The following chart shows the algorithms and sequence of actions of the TCP server and client.



```
root@wang:~/NET# server
1. create a UDP socket
2. fill in server address with port number
3. bind socket to port
4. wait for datagram loop
UDP server: waiting for datagram
received a datagram from [host:port] = [127.0.0.1:46349]
rlen=9: line=test line
send reply
UDP server: waiting for datagram
```

```
root@wang:~/NET# client
1. create a UDP socket
2. fill in server address and port number
Enter a line : test line
send line to server
try to receive a line from server
rlen=9: line=test line
Enter a line :
```

Fig. 13.7 Outputs of UDP server-client program

```
---------- TCP Server ----------- | ---------- TCP Client ----------
1. create a TCP socket            |  1. create a TCP socket sock
2. fill server_addr = [IP, port]  |  2. fill server_addr = [IP, port]
3. bind socket to server_addr     |
4. listen at socket by listen()   |
5. int csock = accept()        <==|= 3. connect() to server via sock
   -------------------------------|---------------------------------
                                     4. while(gets() line){
6. while(read()line from csock){<-|--5.   write() line   to  sock
     write() reply   to  csock  --|--->    read()  reply from sock
   }                              |     }
7. close newsock;                 |  6. exit
8. loop to 5 to accept new client |
----------------------------------------------------------------------
```

```
/******** C13.2.a: TCP server.c file ********/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <netdb.h>

#define MAX          256
#define SERVER_HOST "localhost"
#define SERVER_IP   "127.0.0.1"
#define SERVER_PORT  1234

struct sockaddr_in  server_addr, client_addr;
int  mysock, csock;        // socket descriptors
int  r, len, n;            // help variables
int server_init()
{
  printf("================== server init ======================\n");
  //  create a TCP socket by socket() syscall

  printf("1 : create a TCP STREAM socket\n");
  mysock = socket(AF_INET, SOCK_STREAM, 0);
  if (mysock < 0){
     printf("socket call failed\n"); exit(1);
  }

  printf("2 : fill server_addr with host IP and PORT# info\n");
  // initialize the server_addr structure
  server_addr.sin_family = AF_INET;                // for TCP/IP
  server_addr.sin_addr.s_addr = htonl(INADDR_ANY); // This HOST IP
  server_addr.sin_port = htons(SERVER_PORT);   // port number 1234

  printf("3 : bind socket to server address\n");
  r = bind(mysock,(struct sockaddr*)&server_addr,sizeof(server_addr));
  if (r < 0){
     printf("bind failed\n"); exit(3);
  }
  printf("    hostname = %s port = %d\n", SERVER_HOST, SERVER_PORT);
  printf("4 : server is listening ....\n");
  listen(mysock, 5); // queue length = 5
  printf("=================== init done =======================\n");
}

int main()
{
  char line[MAX];
  server_init();
  while(1){  // Try to accept a client request
    printf("server: accepting new connection ....\n");
    // Try to accept a client connection as descriptor newsock
    len = sizeof(client_addr);
    csock = accept(mysock, (struct sockaddr *)&client_addr, &len);
    if (csock < 0){
       printf("server: accept error\n"); exit(1);
    }
    printf("server: accepted a client connection from\n");
    printf("-----------------------------------------------\n");
    printf("Clinet: IP=%s  port=%d\n",
                inet_ntoa(client_addr.sin_addr.s_addr),
                ntohs(client_addr.sin_port));
    printf("-----------------------------------------------\n");
    // Processing loop: client_sock <== data ==> client
    while(1){
       n = read(csock, line, MAX);
       if (n==0){
           printf("server: client died, server loops\n");
           close(csock);
           break;
       }
       // show the line string
       printf("server: read  n=%d bytes; line=%s\n", n, line);
       // echo line to client
       n = write(csock, line, MAX);
       printf("server: wrote n=%d bytes; ECHO=%s\n", n, line);
       printf("server: ready for next request\n");
    }
  }
}

/******** C13.2.b: TCP client.c file TCP ********/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <netdb.h>

#define MAX          256
#define SERVER_HOST "localhost"
#define SERVER_PORT  1234
struct sockaddr_in  server_addr;
int sock, r;

int client_init()
{
  printf("======= clinet init ==========\n");
  printf("1 : create a TCP socket\n");
  sock = socket(AF_INET, SOCK_STREAM, 0);
  if (sock<0){
     printf("socket call failed\n"); exit(1);
  }

  printf("2 : fill server_addr with server's IP and PORT#\n");
  server_addr.sin_family = AF_INET;
  server_addr.sin_addr.s_addr = htonl(INADDR_ANY); // localhost
  server_addr.sin_port = htons(SERVER_PORT); // server port number

  printf("3 : connecting to server ....\n");
  r = connect(sock,(struct sockaddr*)&server_addr, sizeof(server_addr));
  if (r < 0){
     printf("connect failed\n");  exit(3);
  }
  printf("4 : connected OK to\n");
  printf("---------------------------------------------------------\n");
  printf("Server hostname=%s PORT=%d\n", SERVER_HOST, SERVER_PORT);
  printf("---------------------------------------------------------\n");
  printf("========= init done ==========\n");
}

int main()
{
  int n;
  char line[MAX], ans[MAX];
  client_init();
  printf("********  processing loop  *********\n");
  while (1){
    printf("input a line : ");
    bzero(line, MAX);           // zero out line[ ]
    fgets(line, MAX, stdin);    // get a line from stdin
    line[strlen(line)-1] = 0;   // kill \n at end
    if (line[0]==0)             // exit if NULL line
       exit(0);
    // Send line to server
    n = write(sock, line, MAX);
    printf("client: wrote n=%d bytes; line=%s\n", n, line);
    // Read a line from sock and show it
    n = read(sock, ans, MAX);
    printf("client: read  n=%d bytes; echo=%s\n", n, ans);
  }
}
```

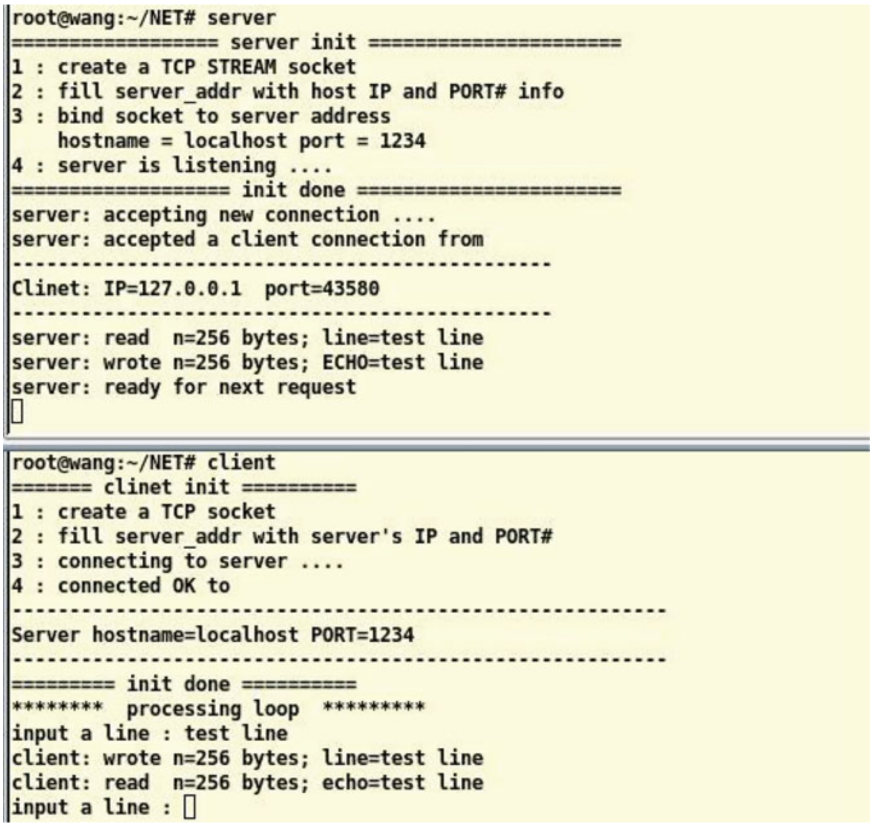
Figure 13.8 shows the sample outputs of running the TCP server-client program C13.2.

13.16 Hostname and IP Address

So for, we have assumed that both the server and client run on the same computer using localhost or IP=127.0.0.1, and the server uses a fixed port number. If the reader intends to run the server and client on different hosts with a server port number assigned by the OS kernel, it is necessary to know server's host name or IP address and its port number. If a computer runs TCP/IP, its hostname is usually recorded in the /etc/hosts file. The library function

gethostname(char *name, sizeof(name))
returns the machine's hostname string in a name array, but it may not be the full official name in dot notation, nor its IP address. The library function

struct hostent *gethostbyname(void *addr, socklen_t len, int typo)
can be used to get the full name of the machine, as well as its IP address. It returns a pointer to a hostent structure in <netdb.h>, which is



```
root@wang:~/NET# server
================== server init ======================
1 : create a TCP STREAM socket
2 : fill server_addr with host IP and PORT# info
3 : bind socket to server address
    hostname = localhost port = 1234
4 : server is listening ....
=================== init done =======================
server: accepting new connection ....
server: accepted a client connection from
-----------------------------------------------
Clinet: IP=127.0.0.1  port=43580
-----------------------------------------------
server: read  n=256 bytes; line=test line
server: wrote n=256 bytes; ECHO=test line
server: ready for next request
```

```
root@wang:~/NET# client
======= clinet init ==========
1 : create a TCP socket
2 : fill server_addr with server's IP and PORT#
3 : connecting to server ....
4 : connected OK to
---------------------------------------------------------
Server hostname=localhost PORT=1234
---------------------------------------------------------
========= init done ==========
********  processing loop  *********
input a line : test line
client: wrote n=256 bytes; line=test line
client: read  n=256 bytes; echo=test line
input a line :
```

Fig. 13.8 Sample outputs of TCP server-client program

```
struct hostent {
   char  *h_name;          // official name of host
   char **h_aliases;       // alias list
   int    h_addrtype;      // host address type
   int    h_length;        // length of address
   char **h_addr_list;     // list of addresses
}
#define h_addr h_addr_list[0] // for backward compatibility
```

Note that h_addr is defined as a char *, but it points at a 4-byte IP address in network byte order. The contents of h_addr can be accessed as

```
u32 NIP = *(u32 *)h_addr is the host IP address in network byte order.
u32 HIP = ntohl(NIP) is NIP in host byte order.
inet_ntoa(NIP) converts NIP to a string in DOT notation.
```

The following code segments show how to use gethostbyname() and getsockname() to get the server IP address and port number if it is dynamically assigned. The server must publish its hostname or IP address and port number for clients to connect.

```
/********* TCP server code *********/
char myname[64];
struct sockaddr_in server_addr, sock_addr;

// 1. gethostname(), gethostbyname()
  gethostname(myname,64);
  struct hostent *hp = gethostbyname(myname);
  if (hp == 0){
     printf("unknown host %s\n", myname); exit(1);
  }

// 2. initialize the server_addr structure
  server_addr.sin_family = AF_INET;    // for TCP/IP
  server_addr.sin_addr.s_addr = *(long *)hp->h_addr;
  server_addr.sin_port = 0; // let kernel assign port number

// 3. create a TCP socket
  int mysock = socket(AF_INET, SOCK_STREAM, 0);

// 4. bind socket with server_addr
  bind(mysock,(struct sockaddr *)&server_addr, sizeof(server_addr));

// 5. get socket addr to show port number assigned by kernel
  getsockname(mysock, (struct sockaddr *)&name_addr, &length);

// 6. show server host name and port number
  printf("hostname=%s IP=%s port=%d\n", hp->h_name,
          inet_ntoa(*(long *)hp->h_addr), ntohs(name_addr.sin_port));

/********* TCP client code *********/
// run as client server_name server_port
  struct sockaddr_in server_addr, sock_addr;
// 1.  get server IP by name
  struct hostent *hp = gethostbyname(argv[1]);
  SERVER_IP   = *(long *)hp->h_addr;
  SERVER_PORT = atoi(argv[2]);

// 2. create TCP socket
  int sock = socket(AF_INET, SOCK_STREAM, 0);

// 3. fill server_addr with server IP and PORT#
  server_addr.sin_family = AF_INET;
  server_addr.sin_addr.s_addr = SERVER_IP;
  server_addr.sin_port = htons(SERVER_PORT);

// 4. connect to server
  connect(sock,(struct sockaddr *)&server_addr, sizeof(server_addr));
```

Incorporating hostname and IP address into the TCP server-client program is left as exercises in the problem section.

13.17 TCP Programming Project: File Server on Internet

The above TCP server-client program can be used as a basis for TCP-based network programming. It can be adapted to different applications by simply changing the data contents and the ways they process the data. For example, instead of numbers, the client may send file operation commands to the server, which processes the commands and send results back to the client. This programming project is to develop a pair of TCP server and client to do file operations across the Internet. The project specification is as follows.

13.17.1 Project Specification

Design and implement a TCP server and a TCP client to do file operations across the Internet. The following charts depict the algorithms of the server and the client.

------------------------------ **Server** ------------------------------	
(1).	Set virtual root to Current Working Directory (CWD)
(2).	Advertise server hostname and port number
(3).	Accept a connection from client
(4).	Get a **command line** = **cmd pathname** from client
(5).	Perform **cmd** on **pathname**
(6).	Send results to client
(7).	Repeat (4) until client disconnects
(8).	Repeat (3) to accept new client connection

------------------------------ **Client** ------------------------------	
(1).	Connect to server at server hostname and port number
(2).	Prompt user for a **command line** = **cmd pathname**
(3).	Send command line to server
(4).	Receive results from server
(5).	Repeat (2) until command line is NULL or quit command

In a command line, depending on the command, pathname may be either a file or a directory. Valid commands are

mkdir:	make a directory with pathname
rmdir:	remove directory named pathname
rm:	remove file named pathname
cd :	change Current Working Directory (CWD) to pathname
pwd:	show absolute pathname of CWD
ls:	list CWD or pathname in the same format as ls –l of Linux
get:	download the pathname file from server
put:	upload the pathname file to server

13.17.2 Helps and Hints

(1). When running on an Internet host, the server must publish its hostname or IP address and port number to allow clients to connect. For security reasons, the server should set the **virtual root** to CWD of the server process to prevent the client from accessing files above the virtual root.

(2). Most commands are simple to implement. For example, each of the first 5 commands requires only a single Linux system call, as shown below.

```
mkdir  pathname:   int r = mkdir(pathname, 0755);  // default permissions
rmdir  pathname:   int r = rmdir(pathname);
rm     pathname:   int r = unlink(pathname);
cd     pathname:   int r = chdir(pathname);
pwd :  char buf[SIZE]; char *getcwd(buf, SIZE);
```

For the ls command, the reader may consult the ls.c program in Section 8.6.7 of Chapter 8. For the get filename command, it is essentially the same as a file copy program split into two parts. The server opens the file for READ, read the file contents and send them to the client. The client opens a file for WRITE, receives data from the server and writes the received data to the file. For the put filename command, simply reverse the roles of the server and client.

(3). When using TCP, data are continuous streams. To ensure the client and server can send/receive commands and simple replies, it is better for both sides to write/read lines of fixed size, e.g. 256 bytes.

(4). The reader must design user-level protocols to ensure correct data transfer between the server and client. The following describes the commands which need user-level data transfer protocols

For the ls command, the server has two options. In the first option, for each entry in a directory, the server generates a line of the form

```
-rw-r--r--  link gid uid size date name
```

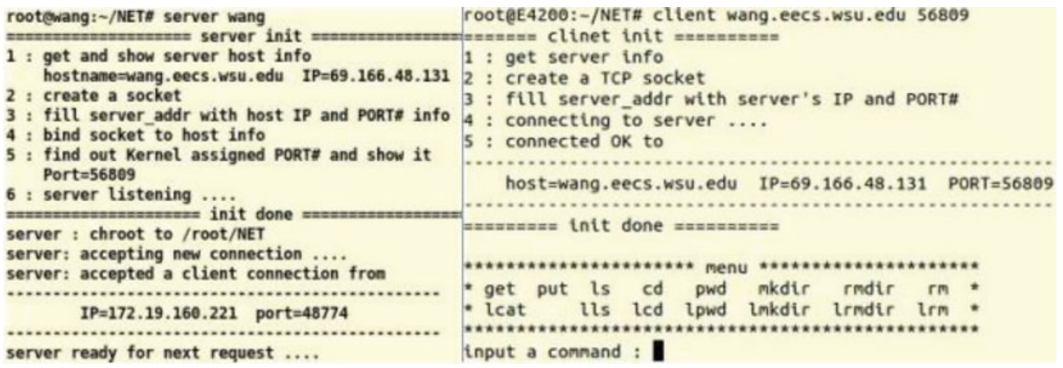
and saves the line into a temporary file. After accumulating all the lines, the server sends the entire temporary file to the client. In this case, the server must be able to tell the client where the file begins and where it ends. Since the ls command generates only text lines, the reader may use special ASCII chars as file begin and end marks.

In the second option, the server may generate a line and send it immediately to the client before generating and sending the next line. In this case, the server must be able to tell the client when it starts to send lines and also when there are no more lines to come.

For the get/put commands, which transfer file contents, it is not possible to use special ASCII chars as file begin and end marks. This is because a binary file may contain any ASCII code. The standard solution to this problem is by bit stuffing [SDLC, IBM, 1979]. In this scheme, while sending data, the sender inserts an extra 0 bit after each sequence of 5 or more consecutive 1 bits, so that the transmitted data never contains any 6 or more consecutive 1's. At the receiving end, the receiver strips off the extra 0 bit after every sequence of 5 consecutive 1's. This allows both sides to use the special flag bits pattern 01111110 as file begin and end marks. Bit-stuffing is inefficient without hardware assistance. The reader must think of other ways to synchronize the server and the client.

Hint: by file size.

(5). The project is suitable for 2-person team work. During development, the team members may discuss how to design user-level protocols and divide the implementation work between them. During testing, one member can run the server on an Internet host while the other runs the client on a different host.



```
root@wang:~/NET# server wang
==================== server init ====================
1 : get and show server host info
    hostname=wang.eecs.wsu.edu  IP=69.166.48.131
2 : create a socket
3 : fill server_addr with host IP and PORT# info
4 : bind socket to host info
5 : find out Kernel assigned PORT# and show it
    Port=56809
6 : server listening ....
==================== init done ====================
server : chroot to /root/NET
server: accepting new connection ....
server: accepted a client connection from
-----------------------------------------------
       IP=172.19.160.221  port=48774
-----------------------------------------------
server ready for next request ....
```

```
root@E4200:~/NET# client wang.eecs.wsu.edu 56809
======= clinet init ==========
1 : get server info
2 : create a TCP socket
3 : fill server_addr with server's IP and PORT#
4 : connecting to server ....
5 : connected OK to
---------------------------------------------------------
    host=wang.eecs.wsu.edu  IP=69.166.48.131  PORT=56809
---------------------------------------------------------
========= init done ==========

********************* menu *********************
* get  put  ls   cd   pwd   mkdir   rmdir   rm  *
* lcat      lls  lcd  lpwd  lmkdir  lrmdir  lrm *
************************************************
input a command : 
```

Fig. 13.9 TCP server-client for file operations

Figure 13.9 shows the sample outputs of running the project program.

As the figure shows, the server runs on the host **wang.eecs.wsu.edu** at port **56809**. The client runs on a different host with **IP=172.19.160.221** at port **48774**. In addition to the specified commands, the client side also implements local file operation commands, which are performed by the client directly.

13.17.3 Multi-threaded TCP Server

In the TCP server-client programming project, the server can only serve one client at a time. In a multi-threaded server, it can accept connections from multiple clients and serve them at the same time or concurrently. This can be done either by fork-exec of new server processes or by multiple threads in the same server process. We leave this extension as another possible programming project.

13.18 Web and CGI Programming

The World Wide Web **(WWW),** or the **Web**, is a combination of resources and users on the Internet that uses the Hypertext Transfer Protocol **(HTTP)** (RFC 2616 1999)**,** for information exchange. Since its debut in the early 90's, coupled with the ever-expanding capability of the Internet, the Web has become an indispensable part of daily lives of people all over the world. It is therefore important for Computer Science students to have some understanding of this technology. In this section, we shall cover the basics of HTTP and Web programming. Web programming in general includes the writing, markup and coding involved in Web development, which includes Web content, Web client and server scripting and network security. In a narrower sense, Web programming refers to creating and maintaining Web pages. The most common languages used in Web programming are HTML, XHTML, JavaScript, Perl 5 and PHP.

13.18.1 HTTP Programming Model

HTTP is a server-client based protocol for applications on the Internet. It runs on top of TCP since it requires reliable transfer of files. Figure 13.10 shows the HTTP programming model.

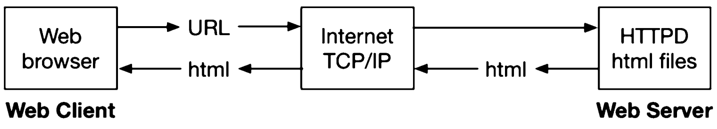


Fig. 13.10 HTTP programming model

In the HTTP programming model, a HTTP server runs on a Web server host. It waits for requests from a HTTP client, which is usually a Web browser. At the HTTP client side, the user enters a **URL** (Uniform Resource Locator) of the form

http://hostname[/filename]
to send a request to a HTTP server, requesting for a file. In the URL, http identifies the HTTP protocol, hostname is the host name of the HTTP server and filename is the requested file. If filename is not specified, the default file is **index.html**. The client first connects to the server to send the request. Upon receiving a request, the server sends the requested file back to the client. Requested files are usually Web page files written in the HTML language for the browser to interpret and display, but it may also be files in other formats, such as video, audio or even binary files.

In HTTP, a client may issue URLs to send requests to different HTTP servers. It is unnecessary, nor desirable, for a client to maintain a permanent connection with a specific server. A client connects to a server only to send a request, after which the connection is closed. Likewise, a server connects to a client only to send a reply, after which the connection is again closed. Each request or reply requires a separate connection. This means that HTTP is a stateless protocol since there are no information maintained across successive requests or replies. Naturally, this would cause a lot of overhead and inefficiency. In order to remedy this lack of state information problem, HTTP server and client may use **cookies**, which are small piece of data imbedded in requests and replies, to provide and maintain some state information between them.

13.18.2 Web Pages

Web pages are files written in the HTML markup language. A Web file specifies the layout of a Web page by a series of HTML elements for a Web browser to interpret and display. Popular Web browsers include Internet Explorer, Firefox, Google Chrome, etc. To create a Web page amounts to creating a text file using HTML elements as building blocks. It is more a clerical type work than programming. For this reason, we shall not discuss how to create Web pages. Instead, we shall only use an example HTML file to illustrate the essence of Web pages. The following shows a simple Web page file in HTML.

```
1.  <html>
2.  <body>
3.  <h1>H1 heading: A Simple Web Page</h1>
4.  <P>This is a paragraph of text</P>
5.  <!---- this is a comment line ---->
6.  <P><img src="firefox.jpg" width=16></P>
7.  <a href="http://www.eecs.wsu.edu/~cs360">link to cs360 web page</a>
    <P>
8.  <font color="red">red</font>
9.  <font color="blue">blue</font>
10. <font color="green">green</font>
```

```
    </P>
    <!--- a table ---->
11. <table>
12.     <tr>
13.        <th>name</th>
14.        <th>ID</th>
15.     </tr>
16.     <tr>
17.         <th>kwang</th>
18.         <th>12345</th>
19.     </tr>
20. </table>
    <!---- a FORM ---->
21. <FORM>
22.   Enter command:  <INPUT NAME="command"><P>
23.   Submit command: <INPUT TYPE="submit" VALUE="Click to Submit">
24. </FORM>
25. </body>
26. </html>
```

Explanations of HTML File Contents

A HTML file comprises HTML elements. Each HTML element is specified by a matched pair of open and close tags.

```
<tag>contents</tag>
```

In fact, a HTML file itself may be regarded as a HTML element specified by a matched pair of <html> tags.

```
<html>HTML file</html>
```

Lines 1 to 26 specify a HTML file. A HTML file includes a body specified by a matched pair of <body> tags

```
<body>body of HTML file</body>
```

Lines 2 to 25 specify the body of the HTML file.

A HTML file may use the tags <H1> to <H7> to display head lines of different font sizes.

Line 3 specifies a <H1> head line.

Each matched pair of <P> tags specifies a paragraph, which is displayed on a new line.

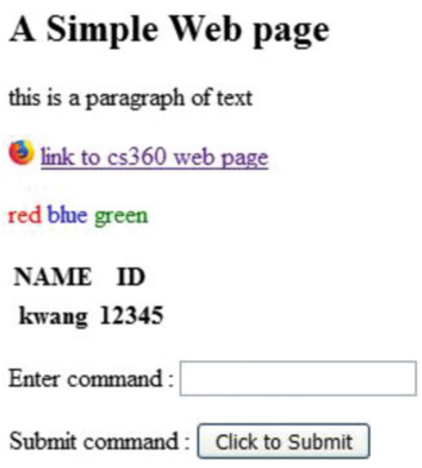
Line 4 specifies a paragraph of text.

Line 5 specifies a comment line, which will be ignored by the browser.

Line 6 specifies an image file, which will be displayed with width pixels per row.

Line 7 specifies a link element

```
<a HREF="link_URL">link</a>
```



A Simple Web page

this is a paragraph of text

link to cs360 web page

red blue green

NAME	ID
kwang	12345

Enter command :

Submit command :

Fig. 13.11 Web page from a HTML file

in which the attribute HREF specifies a link_URL and a text string describing the link. The browser usually displays link texts in dark blue color. If the user clicks on a link, it will direct the request to a Web server identified by the link_URL. This is perhaps the most powerful feature of Web pages. It allows the user to navigate to anywhere in the Web by following the links.

Lines 8 to 10 use <font> elements to display texts in different colors. The <font> element can also specify text in different font sizes and styles.

Lines 11 to 20 specify a table with <tr> as rows, and <th> as columns in each row.

Lines 21 to 24 specify a form for collecting user inputs and submitting them to a Web server for processing. We shall explain and demonstrate HTML forms in more detail in the next section on CGI programming.

Figure 13.11 shows the Web page of the above HTML file.

13.18.3 Hosting Web Pages

Now that we have a HTML file. It must be placed in a Web server. When a Web client requests the HTML file by a URL, the Web server must be able to locate the file and send it back to the client for display. There are several ways to host Web pages.

(1). Sign up with a commercial Web hosting service provider with a monthly fee. For most casual users, this may not be an option at all.

(2). User account on an institutional or departmental server. If the reader has a user account on a server machine running Linux, it's very easy to create a private website in the user's home directory by the following steps

. login to the user account on the server machine.

. in the user's home directory, create a public_html directory with permissions 0755.

. in the public_html directory, create an index.html file and other HTML files.

As an example, from a Web browser on the Internet, entering the URL
http://cs360.eecs.wsu.edu/~kcw
will access the author's website on the server machine **cs360.eecs.wsu.edu**.

(3.) Standalone PC or laptop: The steps described here are for standalone PCs or laptops running standard Linux, but it should be applicable to other Unix platforms as well. For some reason, Ubuntu Linux chooses to do things differently, deviating from the standard setups of Linux. Ubuntu users may consult the HTTPD-Apache2 Web Server Web page of Official Ubuntu Documentation for details.

13.18.4 Configure HTTPD for Web Pages

(3).1. Download and install the Apache Web server. Most Linux distributions, e.g. Slackware Linux 14.2, come with the Apache Web server installed, which is known as HTTPD.
(3).2. Enter ps –x | grep httpd to see whether httpd is running. If not, enter

```
          sudo chmod  +x  /etc/rc.d/rc.httpd
```

to make the rc.httpd file executable. This would start up httpd during next booting. Alternatively, httpd can also be started up manually by entering

```
          sudo /usr/sbin/httpd –kstart.
```

(3).3. Configure httpd.conf file: Operations of the HTTPD server are governed by a httpd.conf file in the /etc/httpd/ directory. To allow individual user websites, edit the httpd.conf file as follows.

```
. Uncomment these lines if they are commented out
    Loadmodule dir_module MODULE_PATH
    Include /etc/httpd/extra/httpd-userdir.conf

. In the first Directory block
       <Directory />
          Require all denied   # deny requests for all files in /
       </Directory>
  Change the line Require all denied to Require all granted.

. All user home directories are in the /home directory. Change the line
       DocumentRoot  /srv/httpd/htdocs to DocumentRoot  /home

. The default directory for all HTML file is htdoc. Change the line
      <Directory /srv/httpd/htdocs> to <Directory /home>

After editing the httpd.conf file, restart the httpd server or enter the
commands
       ps –x | grep httpd    # to see httpd PID
       sudo kill –s 1 httpdPID
```

The kill command sends a number 1 signal to httpd, causing it to read the updated httpd.conf file without restarting the httpd server.

(3).4. Create a user account by **adduser user_name**. login to the user account by

```
ssh user_name@localhost
```

Create public_html directory and HTML files as before.
Then open a Web browser and enter http://localhost/~user_name to access the user's Web pages.

13.18.5 Dynamic Web Pages

Web pages written in standard HTML are all static. When fetched from a server and displayed by a browser, the web page contents do not change. To display a Web page with different contents, a different Web page file must be fetched from the server again. Dynamic Web pages are those whose contents can vary. There are two kinds of dynamic Web pages, known as **client-side** and **server-side** dynamic Web pages, respectively. Client-side dynamic Web page files contain code written in JavaScripts, which are executed by a JavaScripts interpreter on the Client machine. It can respond to user inputs, time events, etc. to modify the Web page locally without any interaction with the server. Server-side dynamic Web pages are truly dynamic in the sense that they are generated dynamically in accordance with user inputs in the URL request. The heart of server-side dynamic Web pages lies in the server's ability to either execute PHP code inside HTML files or CGI programs to generate HTML files by user inputs.

13.18.6 PHP

PHP (Hypertext Preprocessor) (PHP 2017) is a script language for creating server-side dynamic Web pages. PHP files are identified by the .php suffix. They are essentially HTML files containing PHP code for the Web server to execute. When a Web client request a PHP file, the Web server will process the PHP statements first to generate a HTML file, which is sent to the requesting client. All Linux systems running the Apache HTTPD server support PHP, but it may have to be enabled. To enable PHP, only a few modifications to the httpd.conf file are needed, which are shown below.

(1).	DirectoryIndex **index.php**	# default Web page is index.php
(2).	AddType application/x-httpd-php **.php**	# add .php extension type
(3).	Include /etc/httpd/**mod_php.conf**	# load php5 module

After enabling PHP in httpd.conf, restart the httpd server, which will load the PHP module into Linux kernel. When a Web client requests a .php file, the httpd server will fork a child process to execute the PHP statements in the .php file. Since the child process has the PHP module loaded in its image, it can execute the PHP code fast and efficiently. Alternatively, the httpd server can also be configured to execute php as CGI, which is slower since it must use fork-exec to invoke the PHP interpreter. For better efficiency, we assume that .php files are handled by the PHP module. In the following, we shall show basic PHP programming by examples.

(1). PHP statements in HTML files

In a .php file, PHP statements are included inside a pair of PHP tags

```
<?php
  // PHP statements
?>
```

The following shows a simple PHP file, p1.php.

```
<html>
<body>
<?php
   echo "hello world<br>";      // hello world<br>
   print "see you later<br>";  // see you later<br>
?>
</body>
</html>
```

Similar to C programs, each PHP statement must end with a semicolon. It may include comment blocks in matched pairs of /* and */, or use //, # for single comment lines. For outputs, PHP may use either echo or print. In an echo or print statement, multiple items must be separated by the dot (string concatenation) operator, not by white spaces, as in

```
echo "hello world<br> . "see you later<br>";
```

When a Web client requests the p1.php file, the httpd server's PHP preprocessor will execute the PHP statements first to generate HTML lines (shown at the right hand side of PHP lines). It then sends the resulting HTML file to the client.

(2). PHP Variables:

In PHP, variables begins with the $ sign, followed by variable name. PHP variable values may be strings, integers or float point numbers. Unlike C, PHP is a loosely typed language. Users do not need to define variables with types. Like C, PHP allows typecast to change variable types. For most parts, PHP can also convert variables to different types automatically.

```
<?php
  $PID = getmypid();        // return an integer
  echo "pid = $PID <br>";   // pid = php Process PID
  $STR = "hello world!";    // a string
  $A = 123; $B = "456";     // integer 123, string "456"
  $C = $A + $B;             // type conversion by PHP
  echo "$STR Sum=$C<br>";   // hello world! Sum=579<br>
    ?>
```

Like variables in C or sh scripts, PHP variables may be local, global or static.

(3). PHP Operators

In PHP, variables and values may be operated by the following operators.

Arithmetic operators
Assignment operators

Comparison operators
Increment/Decrement operators
Logical operators
String operators
Array operators

Most PHP operators are similar to those in C. We only show some special string and array operators in PHP.

(3).1. String operations: Most string operations, e.g. strlen(), strcmp(), etc. are the same as in C. PHP also supports many other string operations, often in slightly different syntax form. For example, instead of strcat(), PHP uses the dot operator for string concatenation, as in "string1" . "string2"

(3).2. PHP Arrays: PHP arrays are defined by the array() keyword. PHP supports both indexed arrays and multi-dimensional arrays. Indexed arrays can be stepped through by an array index, as in

```
<?php
 $name  = array('name0", "name1", "name2", "name3");
 $value = array(1,2,3,4);  // array of values
 $n = count($name);        // number of array elements
 for ($i=0; $i<n; $i++){   // print arrays by index
     echo $name[$i]; echo " = ";
     echo $value[$i];
 }
?>
```

In addition, PHP arrays can be operated on as sets by operators, such as union (+) and comparisons, or as lists, which can be sorted in different orders, etc.

Associative Arrays: Associative arrays consist of name-value pairs.

```
$A = array('name"=>1, "name1"=>2, "name2"=>3, "name"=>4);
```

An associative array allows accessing element value by name, rather than by index, as in

```
echo "value of name1 = " . $A['name1'];
```

(4). PHP Conditional Statements: PHP supports conditions and test conditions by if, if-else, if-elseif-else and switch-case statements, which are exactly the same as in C, but with slight difference in syntax.

```
<?php
  if (123 < 456){          // test a condition
     echo "true<br>";      // in matched pair of {  }
  } else {
     echo "not true<br>";  // in matched pair of {  }
  }
?>
```

(5). PHP Loop Statements: PHP supports while, do-while, for loop statements, which are the same as in C. The foreach statement may be used to step through an array without an explicit index variable.

```
<?php
  $A = array(1,2,3,4);
  for ($i=0; $i<4; $i++){    // use an index variable
      echo "A[$i] = $A[$i]<br>";
  }
  foreach ($A as $value){    // step through array elements
      echo "$value<br>";
  }
?>
```

(6). PHP Functions: In PHP, functions are defined using the function keyword. Their formats and usage are similar to functions in C.

```
<?php
  function nameValue($name, $value) {
    echo "$name . " has value " . $value <br>";}
  nameValue("abc", 123); // call function with 2 parameters
  nameValue("xyz", 456);
?>
```

(7). PHP Date and Time Functions: PHP has many built-in functions, such as date() and time().

```
<?php
   echo date("y-m-d");  // time in year-month-day format
   echo date("h:i:sa);  // time in hh:mm:ss format
?>
```

(8). File Operations in PHP: One of the great strengths of PHP is its integrated support for file operations. File operations in PHP includes functionalities of both system calls, e.g. mkdir(), link(), unlink(), stat(), etc. and standard library I/O functions of C, e.g. fopen(), fread(), fwrite() and fclose(), etc. The syntax of these functions may differ from that in the I/O library functions in C. Most functions do not need a specific buffer for data since they either take string parameters or return strings directly. As usual, for write operations, the Apache process must have write permissions to the user directory. The reader may consult PHP file operation manuals for details. The following PHP code segments show how to display the contents of a file and copy files by fopen(), fread() and fwrite().

```
<?php
  readfile("filename");          // same as cat filename
  $fp = fopen("src.txt", "r");   // fopen() a file for READ
  $gp = fopen("dest.txt", "w");  // fopen a file for WRITE
  while(!feof($fp){              // feof() same as in C
     $s = fread($fp, 1024);      // read nbytes from $fp
     fwrite($gp, $s, strlen($s));// write string to  $gp
  }
?>
```

(9). Forms in PHP: In PHP, forms and form submission are identical to those in HTML. Form processing is by a PHP file containing PHP code, which is executed by the PHP preprocessor. The PHP

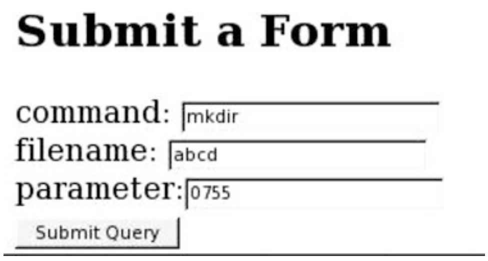


Fig. 13.12 PHP submitting form

```
Submit a Form

command:   mkdir
filename:  abcd
parameter: 0755
Submit Query
```

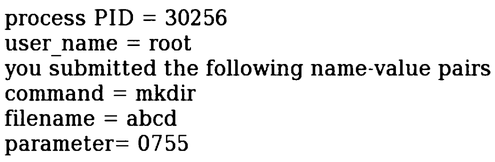


Fig. 13.13 PHP form processing

```
process PID = 30256
user_name = root
you submitted the following name-value pairs
command = mkdir
filename = abcd
parameter= 0755
```

code can get inputs from the submitted form and handle them in the usual way. We illustrate form processing in PHP by an example.

(9).1. A form.php file: This .php file displays a form, collects user inputs and submits the form with METHOD=”post” and ACTION=”action.php” to the httpd server. Figure 13.12 shows the Web page of the form.php file. When the user clicks on Submit, it sends the form inputs to the HPPTD server for processing.

```
<!------- form.php file ------->
<html><body>
<H1>Submit a Form</H1>
<form METHOD="post" ACTION="action.php">
  command:  <input type="text" name="command"><br>
  filename: <input type="text" name="filename"><br>
  parameter:<input type="text" name="parameter"><br>
  <input type="submit">
</form>
</body></html>
```

(9).2. Action.php file: The action.php file contains PHP code to process user inputs. Form inputs are extracted from the global _POST associative array by keywords. For simplicity, we only echo the user submitted input name-value pairs. Figure 13.13 shows the returned Web page of action.php. As the figure shows, it was executed by an Apache process with PID=30256 at the server side.

```
<!------- action.php file ------->
<html><body>
<?php
  echo "process PID = " . getmypid() . "<br>";
  echo "user_name = " . get_current_user() . "<br>";
  $command  = $_POST["command"];
  $filename = $_POST["filename"];
  $parameter= $_POST["parameter"];
  echo "you submitted the following name-value pairs<br>";
  echo "command  = " . $command .   "<br>";
  echo "filename = " . $filename .  "<br>";
  echo "parameter= " . $parameter . " <br>";
?>
</body></html>
```

Summary on PHP

PHP is a versatile script language for developing applications on the Internet. From a technical point view, PHP may have nothing new, but it represents the evolution of several decades of efforts in Web programming. PHP is not a single language but an integration of many other languages. It includes features of many earlier script languages, such as sh and Perl. It includes most of the standard features and functions of C, and it also provides file operations in the standard I/O library of C. In practice, PHP is often used as the front-end of a Web site, which interacts with a database engine at the back-end for storing and retrieving data online through dynamic Web pages. Interface PHP with MySQL databases will be covered the Chap. 14.

13.18.7 CGI Programming

CGI stands for **Common Gateway Interface** (RFC 3875 2004). It is a protocol which allows a Web server to execute programs to generate Web pages dynamically in accordance with user inputs. With CGI, a Web server does not have to maintain millions of static Web page files to satisfy client requests. Instead, it creates Web pages to satisfy client requests by generate them dynamically. Figure 13.14 shows the CGI programming model.

In the CGI programming model, a client sends a request, which is typically a HTML form containing both inputs and the name of a CGI program for the server to execute. Upon receiving the request, the httpd server forks a child process to execute the CGI program. The CGI program may use user inputs to query a database system, such as MySQL, to generate a HTML file based on user inputs. When the child process finishes, the httpd server sends the resulting HTML file back to the client. CGI program can be written in any programming language, such as C, sh scripts and Perl.

13.18.8 Configure HTTPD for CGI

In HTTPD, the default directory of CGI programs is /srv/httpd/cgi-bin. This allows the network administrator to control and monitor which users are allowed to execute CGI programs. In many institutions, user-level CGI programs are usually disabled for security reasons. In order to allow

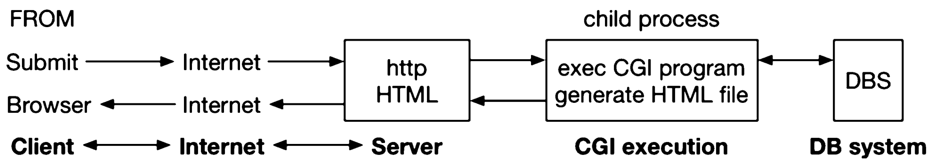


Fig. 13.14 CGI programming model

user-level CGI programming, the httpd server must be configured to enable user-level CGI. Edit the /etc/httpd/httpd.conf file and change the CGI directory settings to

```
 <Directory "/home/*/public_html/cgi-bin">
     Options +ExecCGI
     AddHandler cgi-script .cgi .sh .bin .pl
     Order allow,deny
     Allow from all
</Directory>
```

The modified CGI Directory block sets the CGI directory to public_html/cgi-bin/ in the user home directory. The **cgi-script** setting specifies file with suffix .cgi, .sh, .bin and .pl (for Perl scripts) as executable CGI programs.

13.19 CGI Programming Project: Dynamic Webpage by CGI

This programming project is intended for readers to practice CGI programming. It combines remote file operations, CGI programming and server-side dynamic Web pages into a single package. The organization of the project is as follows.

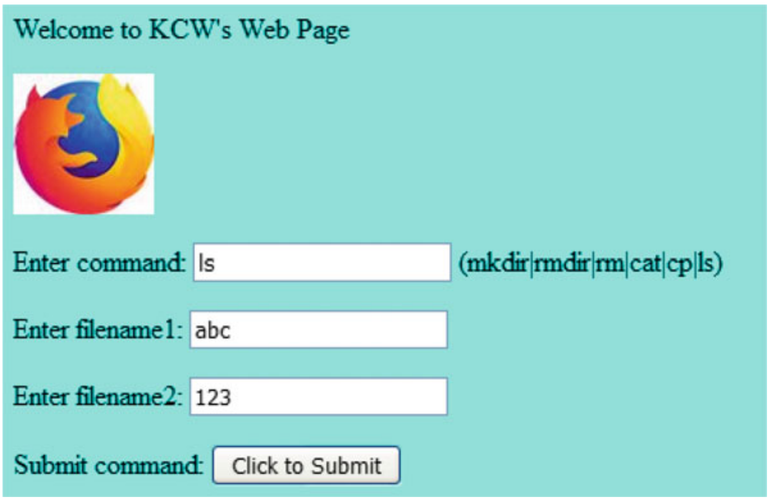
(1). User website: On the cs360.eecs.wsu.edu server machine, each user has an account for login. The user's home directory has a public_html directory containing an index.html file, which can be accessed from a Web browser on the Internet by the URL

http://cs360.eecs.wsu.edu/~username

The following shows the index.html file of the user kcw.

```
<!---------- index.html file ---------->
<html>
<body bgcolor="#00FFFF"
<H1>Welcome to KCW's Web Page</H1><P>
<img src="kcw.jpg" width=100><P>

<FORM METHOD="POST" ACTION=\
  "http://cs360.eecs.wsu.edu/~kcw/cgi-bin/mycgi.bin">
  Enter command: <INPUT NAME="command"> (mkdir|rmdir|rm|cat|cp|ls)<P>
  Enter filename1: <INPUT NAME="filename1"> <P>
  Enter filename2: <INPUT NAME="filename2"> <P>
  Submit command: <INPUT TYPE="submit" VALUE="Click to Submit"> <P>
</FORM>

</body>
</html>
```

Figure 13.15 shows the Web page corresponding to the above index.html file.


Welcome to KCW's Web Page

Enter command: ls (mkdir|rmdir|rm|cat|cp|ls)

Enter filename1: abc

Enter filename2: 123

Submit command: Click to Submit

Fig. 13.15 HTML form page

(2). HTML Form: The index.html file contains a HTML form.

```
<FORM METHOD="POST" ACTION=\
  "http://cs360.eecs.wsu.edu/~kcw/cgi-bin/mycgi.bin">
  Enter command: <INPUT NAME="command"> (mkdir|rmdir|rm|cat|cp|ls)<P>
  Enter filename1: <INPUT NAME="filename1"> <P>
  Enter filename2: <INPUT NAME="filename2"> <P>
  Submit command: <INPUT TYPE="submit" VALUE="Click to Submit"> <P>
</FORM>
```

In a HTML form, METHOD specifies how to submit the form inputs, ACTION specifies the Web server and the CGI program to be executed by the Web server. There are two kinds of form submission methods. In the **GET method**, user inputs are included in the submitted URL, which makes them directly visible and the amount of input data is also limited. For these reasons, GET method is rarely used. In the **POST method**, user inputs are (URL) encoded and transmitted via data streams, which are more secure and the amount of input data is also unlimited. Therefore, most HTML forms use the POST method. A HTML form allows the user to enter inputs in the prompt boxes. When the user clicks on the Submit button, the inputs will be sent to a Web server for processing. In the example HTML file, the form inputs will be sent to the HTTP server at cs360.eecs.wsu.edu, which will execute the CGI program mycgi.bin in the user's cgi-bin directory.

(3). CGI Directory and CGI Programs: The HTTPD server is configured to allow user-level CGI. The following diagram shows the user-lever CGI setup.

```
/home/username:  public_html
                       | ----  index.html
                       | ----  cgi-bin
                                 | ---- mycgi.c
                                 | ---- util.o
                                 | ---- mycgi.bin, sample.bin
```

In the cgi-bin directory, mycgi.c is a C program, which gets and shows user inputs in a submitted HTML form. It echoes user inputs and generates a HTML file containing a FORM, which is sent back to the Web client to display. The following shows the mycgi.c program code.

```
/************** mycgi.c file *************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX 1000
typedef struct{
    char *name;
    char *value;
}ENTRY;
ENTRY entry[MAX];
extern int getinputs(); // in util.o
int main(int argc, char *argv[])
{
  int i, n;
  char cwd[128];
  n = getinputs();     // get user inputs name=value into entry[ ]
  getcwd(cwd, 128);    // get CWD pathname
  // generate a HTML file containing a HTML FORM
  printf("Content-type: text/html\n\n"); // NOTE: 2 new line chars
  printf("<html>");
  printf("<body bgcolor=\"#FFFF00\"); // background color=YELLOW
  printf("<p>pid=%d uid=%d cwd=%s\n", getpid(), getuid(), cwd);
  printf("<H2>Echo Your Inputs</H2>");
  printf("You submitted the following name/value pairs:<p>");
  for(i=0; i<=n; i++)
     printf("%s = %s<P>", entry[i].name, entry[i].value);
  printf("<p>");
  // create a FORM webpage for user to submit again
  printf("---------- Send Back a Form Again ------------<P>");
  printf("<FORM METHOD=\"POST\"
   ACTION=\"http://cs360.eecs.wsu.edu/~kcw/cgi-bin/mycgi.bin\">");
   printf("<font color=\"RED\">");
   printf("Enter command : <INPUT NAME=\"command\"> <P>");
   printf("Enter filename1: <INPUT NAME=\"filename1\"> <P>");
   printf("Enter filename2: <INPUT NAME=\"filename2\"> <P>");
   printf("Submit command: <INPUT TYPE=\"submit\" VALUE=\"Click to \
           Submit\"> <P>");
  printf("</form>");
  printf("</font>");
  printf("-----------------------------------------------<p>");
  printf("</body>");
  printf("</html>");
}
```

Figure 13.16 shows the Web page generated by the above CGI program.

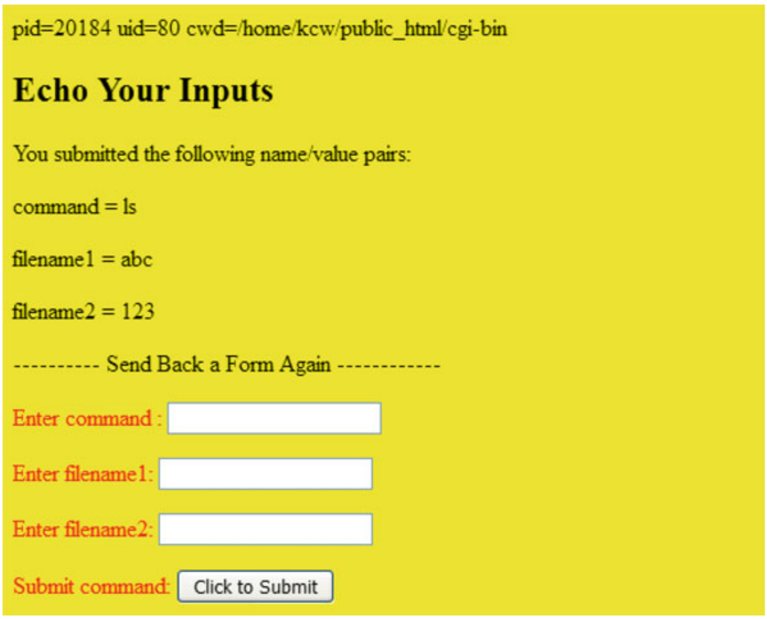


Fig. 13.16 Web page of CGI program

pid=20184 uid=80 cwd=/home/kcw/public_html/cgi-bin

Echo Your Inputs

You submitted the following name/value pairs:

command = ls

filename1 = abc

filename2 = 123

---------- Send Back a Form Again ------------

(4). CGI Programs: When the HTTPD server receives a CGI request**,** it forks a child process with UID 80 to execute the CGI program. The form submission method, inputs encoding and input data length are in the environment variables REQUEST_METHOD, CONETENT_TYPE and CONTENT_LENGTH of the process, and the input data are in stdin. The input data are typically URL-encoded. Decoding the inputs into name-value pairs is straightforward but fairly tedious. For this reason, a pre-compiled util.o file is provided. It contains a function

```
int getinputs()
```

which decodes user inputs into pairs of name-value strings. The CGI program is generated by the Linux commands

```
gcc –o mycgi.bin mycgi.c util.o
```

(5). Dynamic Web Pages: After getting user inputs, the CGI program can process the user inputs to generate output data. In the example program it only echoes the user inputs. Then it generates a HTML file by writing HTML statements as lines to stdout. To qualify the lines as a HTML file, the first line must be

```
printf("Content-type: text/html\n\n");
```

with 2 new line chars. The rest lines can be any HTML statements. In the example program, it generates a FORM identical to the submitted form, which is used to get new user inputs in the next submission.

(6). SETUID Programs: Normally, a CGI program only uses user inputs to read from a database on the server side to generate a HTML file. For security reasons, it may not allow CGI programs to modify the server side database. In the programming project, we shall allow uses requests to do file operations, such as mkdir, rmdir, cp files, etc. which require writing permissions into the user's directory. Since the CGI process has a UID=80, it should not be able to write into the user's directory. There are two options to allow the CGI process to write in the user's directory. In the first option, the user may set the cgi-bin directory permissions to 0777, but this is undesirable because it would allow anyone to be able to write in its directory. The second option is to make the CGI program a SETUID program by

```
chmod u+s mycgi.bin
```

When a process executes a SETUID program, it temporarily assumes the UID of the program owner, allowing it to write in the user's directory.

(7). User Requests for File Operations: Since the objective of the project is CGI programming, we only assume the following simple file operations as user requests:

```
ls        [directory]        // list directory in ls -l form of Linux
mkdir dirname permission     // make a directory
rmdir  dirname               // rm director y
unlink  filename             // rm file
cat       filename           // show file contents
cp        file1 file2        // copy file1 to file2
```

(8) Sample Solution: In the cgi-bin directory, sample.bin is a sample solution of the project. The reader may replace mycgi.bin in the index.html file with sample.bin to test user requests and observe the results.

13.20 Summary

This chapter covers TCP/IP and network programming. The chapter consists of two parts. The first part covers the TCP/IP protocol and its applications. These include the TCP/IP stack, IP address, hostname and DNS, IP data packets and routers. It describes the UDP and TCP protocols, port number, network byte order and data flow in TCP/IP networks. It explains the server-client computing model and the socket programming interface. It demonstrates network programming by examples using both UDP and TCP sockets. The first programming project is to implement a pair of TCP server-client to do file operations across the Internet. It also allows the user to define additional communication protocols for reliable transfer of file contents.

The second part of the chapter covers Web and CGI programming. It explains the HTTP programming model, Web pages and Web browsers. It shows how to configure the Linux HTTPD server to support user Web pages, PHP and CGI programming. It explains both client side and server side dynamic Web pages. It shows how to create server side dynamic Web pages by both PHP and CGI. The second programming project is for the reader to implement server-side dynamic Web pages by CGI programming on a Linux HTTPD server machine.

Problems

1. Modify the UDP client code of Program C13.1 to append a timestamp in usec to the datagram. Modify the UDP server code to append a timestamp, also in usec, to the received datagram before sending it back. In the client code, compute and display the round-trip time of the datagram in msec.
2. Modify the UDP client code of Program 13.1 to add timeout and resend. For each datagram sent, start a REAL mode interval timer with a timeout value. If the client receives a reply from the server

before the timeout expires, cancel the timer. If the timeout expires and the client has not received any reply, resend the datagram.

3. Modify the TCP client code of Program C13.2 to send two integer numbers separated by white spaces, e.g. 1 2, to the server. Modify the TCP server code to send back the sum of the two numbers, e.g. 1 2 sum=3.
4. Modify the TCP server code of Program 13.2 to use gethostbyname() and kernel assigned port number. Modify the TCP client code to find server IP by name.
5. Instead of gethostbyname(), POSIX recommends using the new function getaddrinfo() to get the name and IP address of a host. Read Linux man pages on getaddrinfo() and re-implement the TCP server and client in Program 13.2 using getaddrinfo().
6. Modify the TCP/IP Programming Project of Sect. 13.7 to implement bit-stuffing in both the server and client.
7. Implement a multi-threaded server in the TCP/IP Programming Project of Sect. 13.7 to support multiple clients.

References

ARP, RFC 826, 1982
Apache, HTTP Server Project,https://httpd.apache.org, 2017
Comer, D., “Internetworking with TCP/IP Principles, Protocols, and Architecture”, Prentice Hall, 1988.
Comer, D., “Computer Networks and Internets with Internet Applications”, 3rd edition, Pretence Hall, 2001
IBM Synchronous Data Link Control, IBM, 1979
PHP: History of PHP and Related Projects, www, php.net, 2017
Rago, Stephen A, “Unix System V Network Programming”. Addision Wesley, 1993
RFC 134, Domain Names-Concepts and facilities, 1987
RFC 768, User Datagram Protocol, 1980
RFC 826, An Ethernet address Resolution protocol, 1982
RFC 1035, Domain Names-Implementation and Specification, 1987
RFC1180, A TCP/IP Tutorial, January 1991
RFC 2131, Dynamic Host Configuration Protocol, 1997
RFC 2616, Hypertext Transfer Protocol - HTTP/1.1, June 1999
RFC 3875, The Common Gateway Interface (CGI) Version 1.1, Oct. 2004
Stevens, Richard W., Fenner, Bill, Ruddof Andrew M., “UNIX Network Programming”, Volume 1, 3rd edition, Addison Wesley, 2004