Computer Networks 1: Fundamentals

Agenda

- Topic 5–Transport Protocols
  - TCP
    - Response to Congestion
    - Quick questions

- Topic 6-Socket programming and project review

➢ Response to Congestion

- When cwnd<ssthresh,
  - If in each RTT, ack arrives, cwnd is doubled;
  - If timer expires, it implies that congestion occurs. The congestion avoidance algorithm will be launched.
- If cwnd>ssthresh, for each RTT elapsing, cwnd++
- cwnd=ssthresh, either of the above.

➢ Response to Congestion

- Congestion avoidance–slow-start
  - If timer expires, ssthresh will be halved, but not less than 2.
  - cwnd is reset to 1.
  - Conduct the algorithm on page 3.

EStablishing a TCP Connection

➔ Tasks:

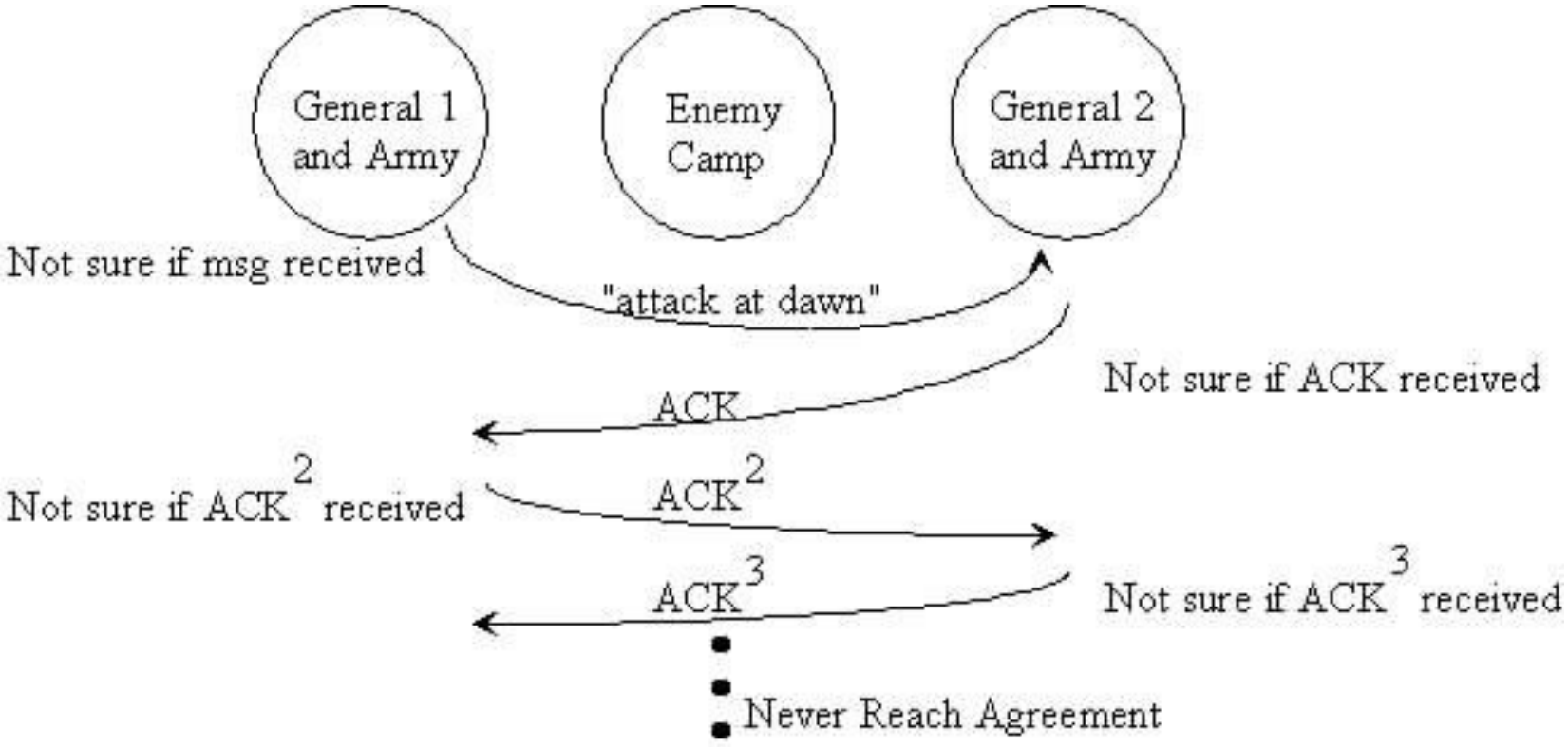
•Be aware the existence of the other party of the connection;

•Perimeter negotiation (mss,window size,QoS, et al);

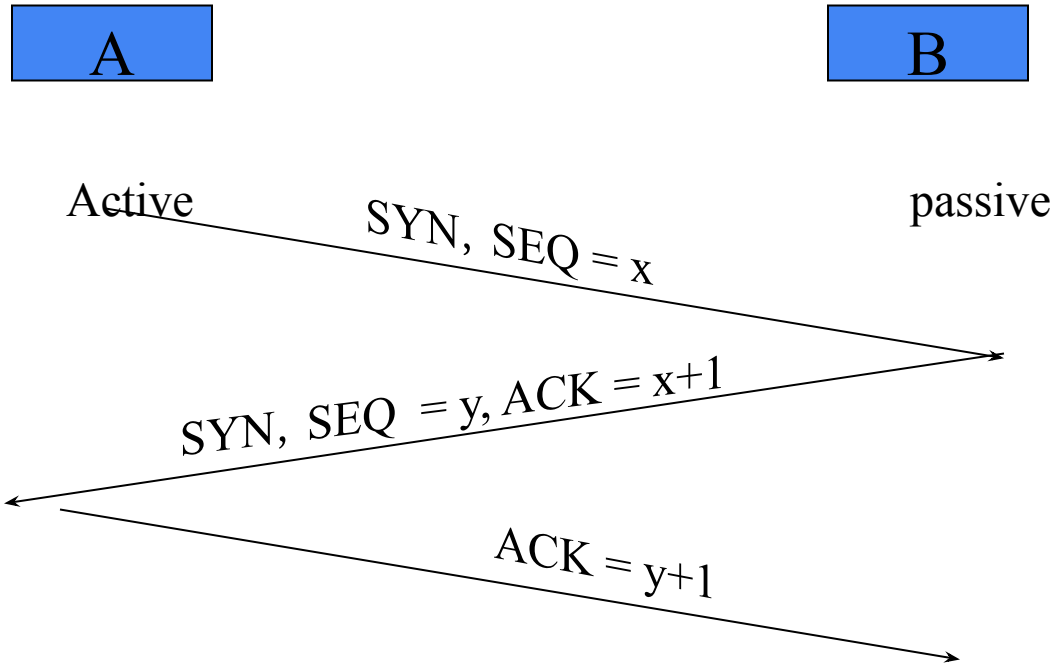
•Resource allocation: buffer, connection table space, et al.

➔ Method: 3-way handshake

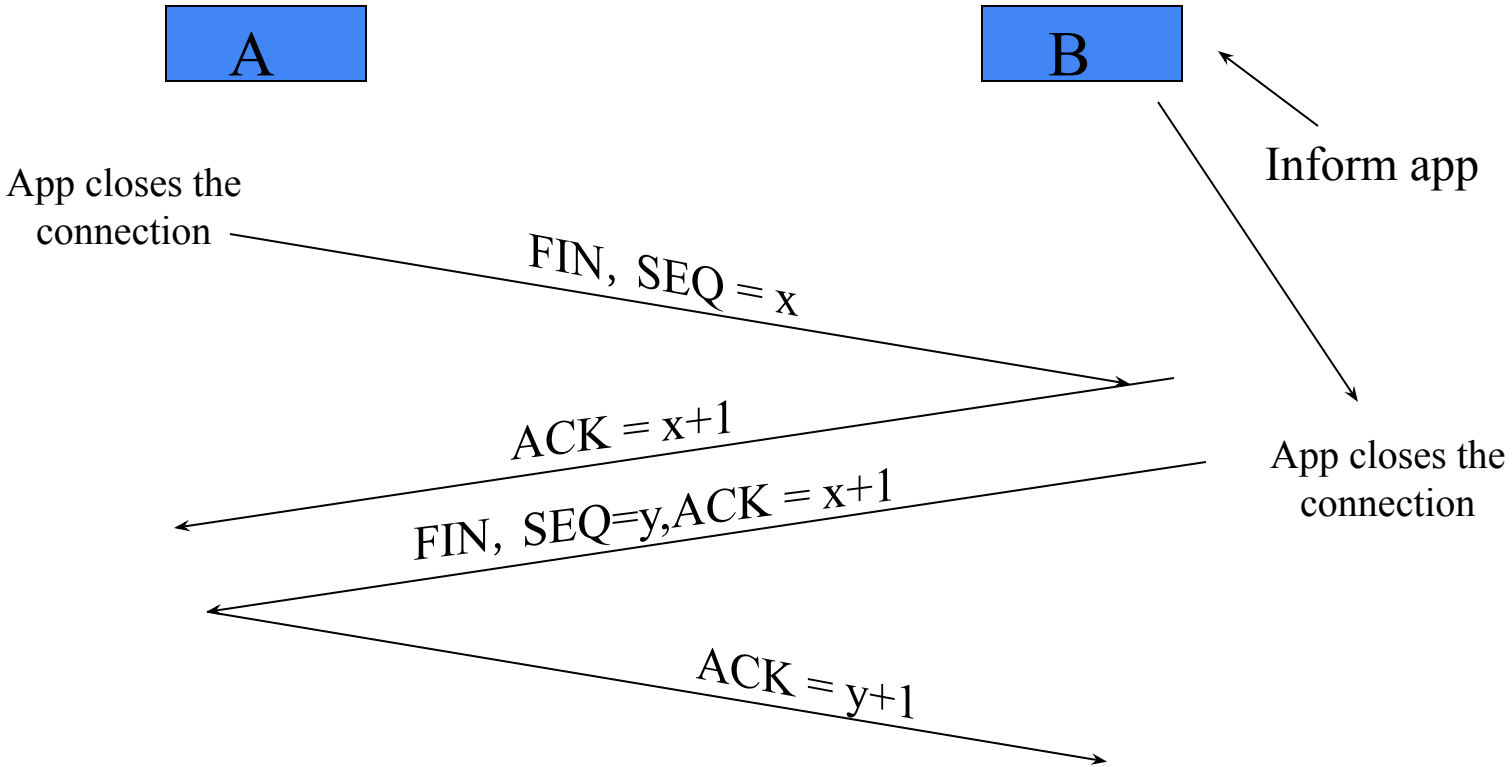
Two-army Problem



EStablishing a TCP Connection



Closing a TCP Connection

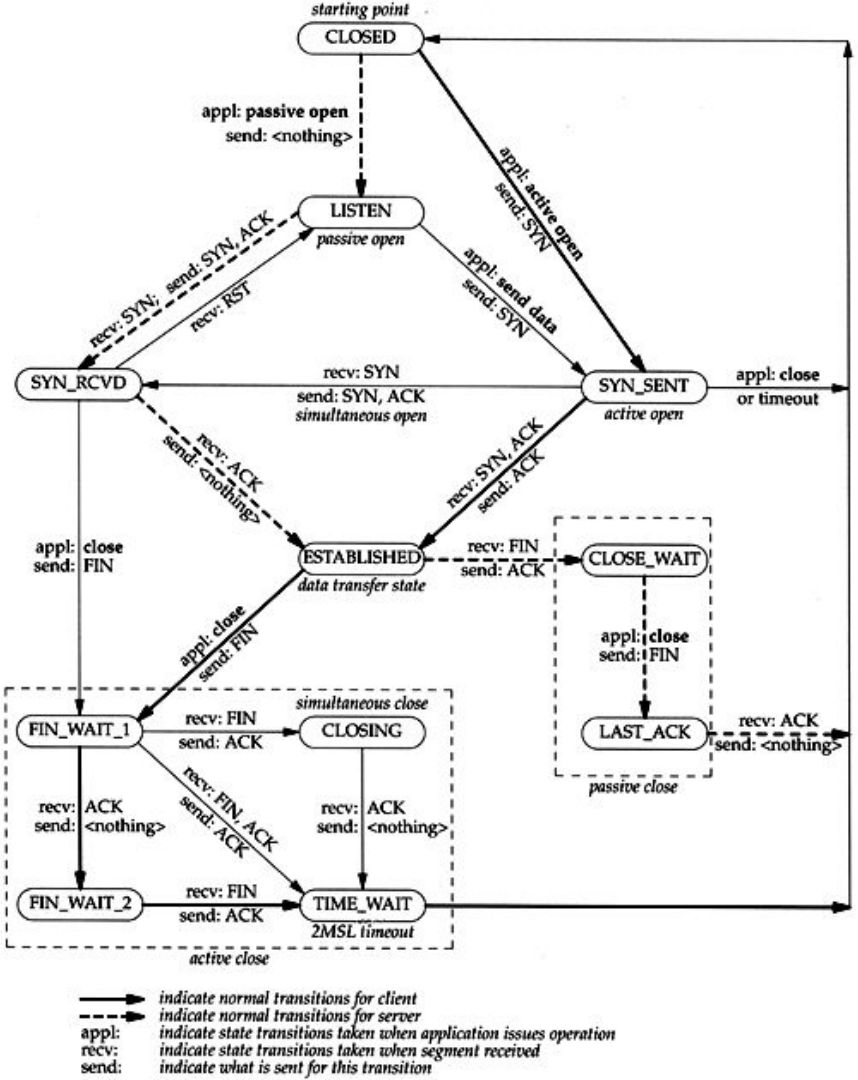


TCP State Machine

TCP Connection Table in a Host

	Status	local IP	local port	dest IP	dest port
connection 1					
connection 2					
……					
connection n					

TCP State Machine



Quick Questions

1. In a long time (>1s), the sender's application only generates a data of 1 octet. If the data is carried by TCP, What is the throughput? If it is carried by UDP, what is the throughput?

2. Argue that the scenario below will not happen to TCP: A and B are two parties of a TCP connection. When A is done with sending data to B, it sends out a segment with FIN and releases the connection. But at the moment, B has some data in the output buffer. The release by A causes B's data is the output buffer voided.

Quick Questions（cnt’d）

3. A, B are two hosts in the Internet. A actively sends a connection request to B (SYN=1). But B does not agree to the request due to resources shortage and sends back a reply (SYN=1, ACK=0). In a short time, B’s resources are cleared and it requests for a TCP connection with A by sending SYN segment. Notice that two segments from B may arrive at A disordered (because they are carried by IP). How could A tell these two TCP segments: which one is the disagreement and which one is the request given that they both set SYN as 1 and ACK as 0?

Topic 6–socket programming

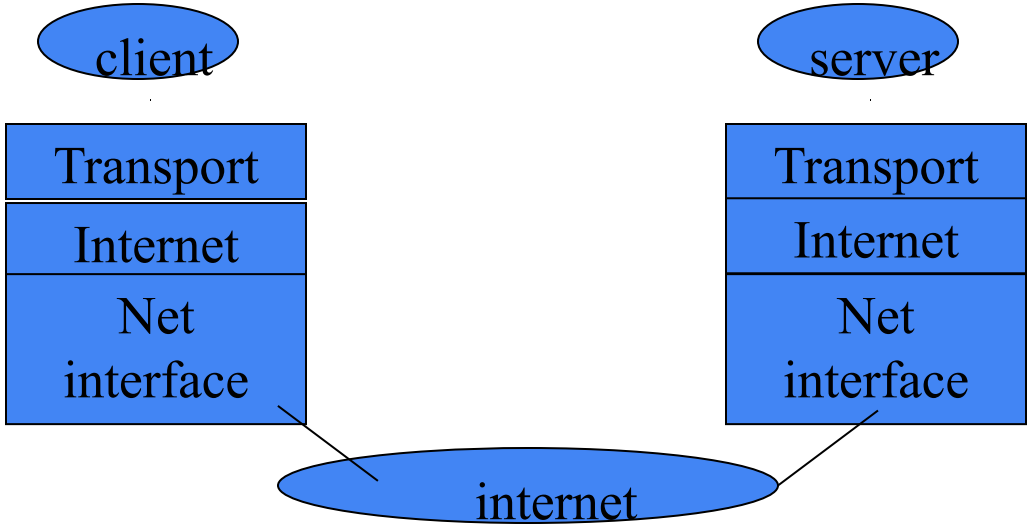
Most OSs provide a set of APIs to for TCP and UDP service invoking.

API: Application Program Interface, it is a channel that applications or programs and OS communicate through.

Socket API: OS could define and implement functions of protocols with API functions. Most OSs provide Socket API (Unix, Windows 9x, Windows NT/2000/XP/10, et al.)

Socket API was proposed by BSD unix and designed as a part of unix. Some other OSs provide socket API through Library functions.

➢ Client/Server Structure



Client: active party ;

Server: passive party.

(1) The *socket* Procedure

The *socket* procedure creates a socket and returns an integer descriptor:

descriptor = socket(*protofamily, type, protocol*)

protofamily specifies the protocol family to be used with the socket. For example, the value PF_INET is used to specify the TCP/IP protocol suite, and PF_APPLETALK is used to specify AppleTalk protocols.

(1) The *socket* Procedure

type specifies the type of communication the socket will use. The two most common types are a connection-oriented stream transfer (specified with the value SOCK_STREAM), and a connectionless message-oriented transfer (specified with the value SOCK_DGRAM).

protocol specifies a particular transport protocol used with the socket. Having a ***protocol*** argument in addition to a type argument, permits a single protocol suite to include two or more protocols that provide the same service. Of course, the values that can be used with the protocol argument depend on the protocol family. For examples although the TCP/IP protocol suite includes the protocol TCP, the AppleTalk suite does not.

(2) The *close* Procedure

The ***close*** procedure tells the system to terminate use of a socket. It has the form:

close(*socket*)

where ***socket*** is the descriptor for a socket being closed. If the socket is using a connection-oriented transport protocol, ***close*** terminates the connection before closing the socket. Closing a socket immediately terminates use-the descriptor is released, preventing the application from sending more data, and the transport protocol stops accepting incoming messages directed to the socket, preventing the application from receiving more data.

(3) The *bind* Procedure

When created a socket has neither a local address nor a remote address. A server uses the ***bind*** procedure to supply a protocol port number at which the server will wait for contact. ***bind*** takes three arguments:

bind(*socket, localaddr, addrlen*)

Argument *socket* is the descriptor of a socket that has been created but not previously bound; the call is a request that the socket be assigned a particular protocol port number.

Argument *localaddr* is a structure that specifies the local address to be assigned to the socket.

Argument *addrlen* is an integer that specifies the length of the address.

(3) The *bind* Procedure

Because sockets can be used with arbitrary protocols, the format of an address depends on the protocol being used. The socket API defines a generic form used to represent addresses, and then requires each protocol family to specify how their protocol addresses use the generic form. The generic format for representing an address is defined to be a *sockaddr* structure. Although several versions have been released, the latest Berkeley code defines a *sockaddr* structure to have three fields:

```
struct sockaddr {
    u_char sa_len;        /*total length of the address */
    u_char sa_family;     /* family of the address */
    char   sa_data[14];   /* the address itself */
    };
```

(3) The *bind* Procedure

Each protocol family defines the exact format of addresses used with the sa_data field of a sockaddr structure. For example, TCP/IP protocols use structure *sockaddr_in* to define an address:

```
struct sockaddr_in {
u_char sin_len;              /*total length of the address */
u_char sin_family;           /* family of the address */
u_short sin_port;            /*protocol port number */
struct in_addr_ sin_addr     /* IP address of computer */
char sin_zero[8] ;           /* not used(set zero) */
};
```

(4) The *listen* Procedure

After specifying a protocol port, a server must instruct the operating system to place a socket in passive mode so it can be used to wait for contact from clients. To do so, a server calls the ***listen*** procedure, which takes two arguments:

listen(*socket, queuesize*)

Argument ***socket*** is the descriptor of a socket that has been created and bound to a local address.

Argument *queuesize* specifies a length for the socket's request queue.

(5) The *accept* Procedure

All servers begin by calling socket to create a socket and bind to specify a protocol port number. After executing the two calls, a server that uses a connectionless transport protocol is ready to accept message. However, a server that uses a connection-oriented transport protocol requires additional steps before it can receive messages: the server must call **listen** to place the socket in passive mode, and must then accept a connection request. Once a connection has been accepted, the server can use the connection to communicate with a client. After it finishes communication, the server closes the connection.

(5) The *accept* Procedure

A server using connection-oriented transport calls procedure ***accept*** to accept the next connection request. If a request is present in the queue, accept returns immediately; if no requests have arrived, the system blocks the server until a client forms a connection.

newsock = accept(*socket, caddress, caddresslen*)

Argument ***socket*** is the descriptor of a socket the server has created and bound to a specific protocol port.

Argument ***caddress*** is the address of a structure of type sockaddr, and ***caddresslen*** is a pointer to an integer.

(6) The *connect* Procedure

Clients use procedure ***connect*** to establish connection with a specific server. The form is:

connect(*socket, saddress, saddresslen*)

Argument *socket* is the descriptor of a socket on the client's computer to use for the connection.

Argument *saddress* is a sockaddr structure that specifies the server's address and protocol port number.

Argument *saddresslen* specifies the length of the server's address measured in octets.

(6) The *connect* Procedure

When used with a connection-oriented transport protocol such as TCP, ***connect*** initiates a transport-level connection to the specified server. In essence, ***connect*** is the procedure a client uses to contact a server that has called ***accept***. Interestingly, a client that uses a connectionless transport protocol can also call connect. However, doing so does not initiate a connection or cause packets to cross the internet. Instead, connect merely marks the socket connected, and records the address of the server.

(7) The *send, sendto*, And *sendmsg* Procedures

Both clients and servers need to send information. Usually, a client sends a request, and a server sends a response. If the socket is connected, procedure ***send*** can be used to transfer data. Send has four arguments:

send(*socket, data, length,flags*)

Argument ***socket*** is the descriptor of a socket to use.

Argument ***data*** is the address in memory of the data to send.

Argument ***length*** is an integer that specifies the number of bytes of data.

Argument ***flags*** contains bits that request special options.

(7) The *send, sendto*, And *sendmsg* Procedures

Procedures ***sendto*** and ***sendmsg*** allow a client or server to send a message using an unconnected socket; both require the caller to specify a destination. ***Sendto***, takes the destination address as an argument. It has the form:

sendto(*socket, data, length, flags, destaddress, addresslen*)

The first four arguments correspond to the four argument of the ***send*** procedure. The final two arguments specify the address of a destination and the length of that address.The form of the address in argument ***destaddress*** is the sockaddr strcture (specifically, structure *sockaddr_in* when used with TCP/IP).

(7) The *send, sendto*, And *sendmsg* Procedures

The ***sendnmsg*** procedure performs the same operation as sendto, but abbreviates the arguments by defining a structure. The shorter argument list can make programs that use ***sendmsg*** easier to read:

sendmsg(*socket, msgstruct, flags*)

Argument *msgstruct* is a structure that contains information about the destination address, the length of the address, the message to be sent, and the length of the message.

(8) The recv, recvfrom, And recvmsg Procedures

A client and a server each need to receive data sent by the other. The socket API provides several procedures that can be used. For example, an application can call ***recv*** to receive data from a connected socket. The procedure has the form:

recv(*socket, buffer, length, flags*)

Argument ***socket*** is the descriptor of a socket from which data is to be received.

Argument ***buffer*** specifies the address in memory in which the incoming message should be placed.

Argument length specifies the size of the buffer.

Argument ***flags*** allows the caller to control details.

(8) The recv, recvfrom, And recvmsg Procedures

If a socket is not connected, it can be used to receive messages from an arbitrary set of clients. In such cases, the system returns the address of the sender along with each incoming message. Applications use procedure *recvfrom* to receive both a message and the address of the sender:

recvfrom(socket, buffer, length, flags, sndraddr, saddrlen)

The first four arguments comespond to the arguments of recv; the two additional arguments, ***sndraddr*** and ***saddrlen,*** are used to record the sender's address. Argument ***sndraddr*** is a pointer to a sockaddr structure into which the system writes the sender's address, and argument ***saddrlen*** is a pointer to an integer that the system uses to record the length of the address.

(8) The *recv,recvfrom*, And *recvmsg* Procedures

recvfrom records the sender's address in exactly the same form that sendto expects. Thus, if an application uses ***recvfrom*** to receive an incoming message, sending a reply is easy - the application simply uses the recorded address as a destination for the reply.

The socket API includes an input procedure analogous to the ***sendnmsg*** output procedure. Procedure ***recvmsg*** operates like ***recvfrom***, but requires fewer arguments. It has the form:

recvmsg(*socket, msgstruct, flags*)

where argument ***msgstruct*** gives the address of a structure that holds the address for an incoming message as well as locations for the sender's address. The ***msgstruct*** recorded by r***ecvrmsg*** uses exactly the same format as the structure required by ***sendmsg***. Thus, the two procedures work well for receiving a message and sending a reply.

(9) *read* And *write* With Sockets

We said the socket API was originally designed to be part of UNIX, which uses ***read*** and ***write*** for I/O. Consequently, sockets also allow applications to use ***read*** and ***write*** to transfer data. Like ***send*** and ***recv***, ***read*** and ***write*** do not have arguments that permit the caller to specify a destination. Instead, ***read*** and ***write*** each have three arguments: a socket descriptor, the location of a buffer in memory used to store the data,and the length of the memory buffer. Thus, ***read*** and ***write*** must be used with connected sockets.

The chief advantage of using read and write is generality - an application program can be created that transfers data to or from a descriptor without knowing whether the descriptor corresponds to a file or a socket. Thus, a programmer can use a file on a local disk to test a client or server before attempting to communicate across a network. The chief disadvantage of using ***read*** and ***write*** is that a socket library implementation may introduce additional overhead in the file I/O of any application that also uses sockets.

(10) Other Socket Procedures

The socket API contains other useful procedures. For example, ***getpeername, gethostname, setsockopt, getsockopt*** **.**

Getpeername: to obtain the complete address of the remote client that initiated the connection. after a server calls procedure *accept* to accept an incoming connection request, the server can call procedure ***getpeername*** to obtain the complete address of the remote client that initiated the connection.

Gethostname: to obtain information about the computer on which it is running. A client or server can also call ***gethostname*** to obtain information about the computer on which it is running.

(10) Other Socket Procedures

We said that a socket has many parameters and options. Two general-purpose procedures are used to set socket options or obtain a list of current values.

setsockopt: to store values in socket options,

getsockopt: to obtain current option values.

Options are used mainly to handle special cases (e.g., to increase performance by changing the internal buffer size the protocol software uses).

(10) Other Socket Procedures

Gethostbyname: name–IP.

gethostbyaddr : IP–name.

Getprotobyname: TCP–ID.

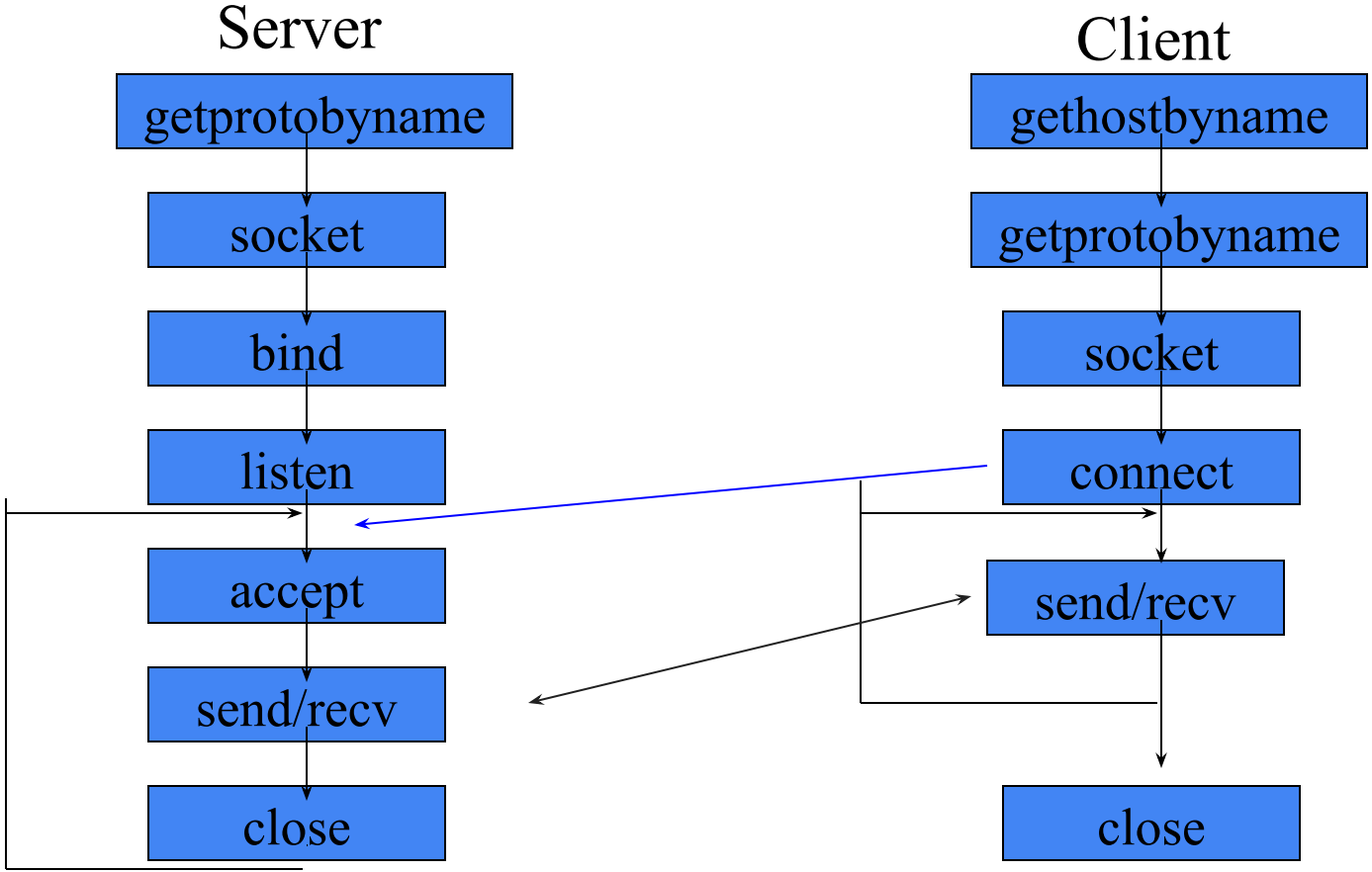
htonl: byte order in host —-byte order in network (long int).

htons:byte order in host —-byte order in network (short int).

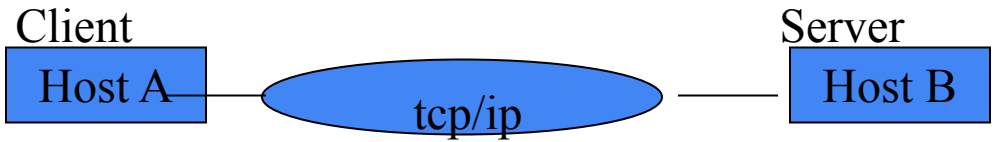
ntohl:byte order in network —-byte order in host (long int).

ntohs:byte order in network —-byte order in host (short int).

(11) APIs for TCP service



Socket programming –example1



client host port server port

output:This server has been contacted *n* time.

client.c

```
/* client.c - code for example client program that uses TCP */
#ifndef unix
#define WIN32
#include <windows.h>
#include <winsock.h>
#else
#define closesocket close
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#endif
#include <stdio.h>
#include <string.h>
#define PROTOPORT       5193            /* default protocol port number */
extern  int     errno;
char    localhost[] =   "localhost";       /* default host name  */
```

client.c

```
/*--------------------------------------------------------------------
 * Program:   client
 *
 * Purpose:   allocate a socket, connect to a server, and print all output
 *
 * Syntax:    client [ host [port] ]
 *
 *               host  - name of a computer on which server is executing
 *               port  - protocol port number server is using
 *
 * Note:  Both arguments are optional.  If no host name is specified,
 *            the client uses "localhost"; if no protocol port is
 *            specified, the client uses the default given by PROTOPORT.
 *
 *-------------------------------------------------------------------- */
```

client.c

```
main(argc, argv)
int     argc;
char    *argv[];
{
        struct  hostent  *ptrh;    /* pointer to a host table entry       */
        struct  protoent *ptrp;     /* pointer to a protocol table entry   */
        struct  sockaddr_in sad;  /* structure to hold an IP address     */
        int     sd;                      /* socket descriptor                */
        int     port;                    /* protocol port number              */
        char  *host;                /* pointer to host name              */
        int      n;                      /* number of characters read          */
        char  buf[1000];          /* buffer for data from the server     */
#ifdef WIN32
          WSADATA wsaData;
          WSAStartup(0x0101, &wsaData);
#endif
```

client.c

```
memset((char *)&sad,0,sizeof(sad)); /* clear sockaddr structure */
sad.sin_family = AF_INET;              /* set family to Internet     */

/* Check command-line argument for protocol port and extract    */
/* port number if one is specified.  Otherwise, use the default */
/* port value given by constant PROTOPORT                       */

if (argc > 2) {                /* if protocol port specified   */
        port = atoi(argv[2]);   /* convert to binary           */
} else
{
        port = PROTOPORT;       /* use default port number      */
}
if (port > 0)                  /* test for legal value         */
        sad.sin_port = htons((u_short)port);
else {                         /* print error message and exit */
        fprintf(stderr,"bad port number %s\n",argv[2]);
        exit(1);
}
```

client.c

```
/* Check host argument and assign host name. */

        if (argc > 1) {
                host = argv[1];         /* if host argument specified   */
        } else {
                host = localhost;
        }

        /* Convert host name to equivalent IP address and copy to sad. */

        ptrh = gethostbyname(host);
        if ( ((char *)ptrh) == NULL ) {
                fprintf(stderr,"invalid host: %s\n", host);
                exit(1);
        }
        memcpy(&sad.sin_addr, ptrh->h_addr, ptrh->h_length);
```

client.c

```
/* Map TCP transport protocol name to protocol number. */
if ( ((int)(ptrp = getprotobyname("tcp"))) == 0) {
        fprintf(stderr, "cannot map \"tcp\" to protocol number");
        exit(1);
}

/* Create a socket. */
sd = socket(PF_INET, SOCK_STREAM, ptrp->p_proto);
if (sd < 0) {
        fprintf(stderr, "socket creation failed\n");
        exit(1);
}
/* Connect the socket to the specified server. */
if (connect(sd, (struct sockaddr *)&sad, sizeof(sad)) < 0) {
        fprintf(stderr,"connect failed\n");
        exit(1);
}
```

client.c

```
	/* Repeatedly read data from socket and write to user's screen. */

	n = recv(sd, buf, sizeof(buf), 0);
	while (n > 0) {
		write(1,buf,n);
		n = recv(sd, buf, sizeof(buf), 0);
	}

	/* Close the socket. */

	closesocket(sd);

	/* Terminate the client program gracefully. */

	exit(0);
}
```

server.c

```
/* server.c - code for example server program that uses TCP */
#ifndef unix
#define WIN32
#include <windows.h>
#include <winsock.h>
#else
#define closesocket close
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#endif

#include <stdio.h>
#include <string.h>
#define PROTOPORT       5193            /* default protocol port number */
#define QLEN            6               /* size of request queue        */

int     visits      =   0;              /* counts client connections    */
```

server.c

```
/*------------------------------------------------------------------------
* Program:   server
*
* Purpose: allocate a socket and then repeatedly execute the following:
*              (1) wait for the next connection from a client
*              (2) send a short message to the client
*              (3) close the connection
*              (4) go back to step (1)
*
* Syntax:    server [ port ]
*
*              port  - protocol port number to use
*
* Note:      The port argument is optional.  If no port is specified,
*              the server uses the default given by PROTOPORT.
*
*------------------------------------------------------------------------
*/
```

server.c

```
main(argc, argv)
int     argc;
char    *argv[];
{
        struct  hostent  *ptrh;      /* pointer to a host table entry          */
        struct  protoent *ptrp;      /* pointer to a protocol table entry      */
        struct  sockaddr_in sad;  /* structure to hold server's address  */
        struct  sockaddr_in cad;  /* structure to hold client's address   */
        int     sd, sd2;                 /* socket descriptors                     */
        int     port;                    /* protocol port number                   */
        int     alen;                    /* length of address                      */
        char    buf[1000];          /* buffer for string the server sends  */
#ifdef WIN32
        WSADATA wsaData;
        WSAStartup(0x0101, &wsaData);
#endif
        memset((char *)&sad,0,sizeof(sad));       /* clear sockaddr structure */
        sad.sin_family = AF_INET;                 /* set family to Internet     */
        sad.sin_addr.s_addr = INADDR_ANY;  /* set the local IP address   */
```

server.c

```
/* Check command-line argument for protocol port and extract    */
/* port number if one is specified.  Otherwise, use the default */
/* port value given by constant PROTOPORT                       */

if (argc > 1) {                 /* if argument specified        */
        port = atoi(argv[1]);   /* convert argument to binary   */
} else {
        port = PROTOPORT;       /* use default port number      */
}
if (port > 0)                   /* test for illegal value       */
        sad.sin_port = htons((u_short)port);
else {                          /* print error message and exit */
        fprintf(stderr,"bad port number %s\n",argv[1]);
        exit(1);
}
  /* Map TCP transport protocol name to protocol number */
  if ( ((int)(ptrp = getprotobyname("tcp"))) == 0) {
        fprintf(stderr, "cannot map \"tcp\" to protocol number");
        exit(1);
}
```

server.c

```
 /* Create a socket */
sd = socket(PF_INET, SOCK_STREAM, ptrp->p_proto);
 if (sd < 0) {
        fprintf(stderr, "socket creation failed\n");
        exit(1);
 }

 /* Bind a local address to the socket */
  if (bind(sd, (struct sockaddr *)&sad, sizeof(sad)) < 0) {
        fprintf(stderr,"bind failed\n");
        exit(1);
 }

 /* Specify size of request queue */
 if (listen(sd, QLEN) < 0) {
        fprintf(stderr,"listen failed\n");
        exit(1);
 }
```

server.c

```
/* Main server loop - accept and handle requests */

        while (1) {
                alen = sizeof(cad);
                if ( (sd2=accept(sd, (struct sockaddr *)&cad, &alen)) < 0)
               {
                        fprintf(stderr, "accept failed\n");
                        exit(1);
                }
                visits++;
                sprintf(buf,"This server has been contacted %d time%s\n",
                        visits);
                send(sd2,buf,strlen(buf),0);
                closesocket(sd2);
        }
}
```

Socket programming -example2-Raw Sockets (Ping)

Raw Sockets provide some functions that UDP Sockets or TCP Sockets can not fullfill:

- Access IGMP(Internet Group Management Protocol) and ICMP messages;
- read/write IP packets that the kernel does not process;
- Program by calling IP directly.

ping.c

```
/*********************************************************\
* ping.c - Simple ping utility using SOCK_RAW
\*********************************************************/

#define WIN32_LEAN_AND_MEAN
#include <winsock2.h>
#include <stdio.h>
#include <stdlib.h>

#define ICMP_ECHO 8          // type 9-echo request
#define ICMP_ECHOREPLY 0     // type0-echo reply
#define ICMP_MIN 12          // minimum 12 byte icmp packet (just header)
```

ping.c

```
/* The IP header  20bytes*/
typedef struct iphdr {
        unsigned char h_len:4;          // length of the header
        unsigned char version:4;        // Version of IP
        unsigned char tos;              // Type of service
        unsigned short total_len;       // total length of the packet
        unsigned short ident;           // unique identifier
        unsigned short frag_and_offset;    // flags and offset
        unsigned char  ttl;
        unsigned char proto;               // protocol (TCP, UDP etc)
        unsigned short checksum;        // IP checksum
        unsigned int sourceIP;
        unsigned int destIP;

}IpHeader;
```

ping.c

```
 /* ICMP header  12 bytes */
typedef  struct _ihdr {
    BYTE  i_type;
    BYTE  i_code;              /* type sub code */
    USHORT  i_cksum;
    USHORT  i_id;
    USHORT  i_seq;
    /* This is not the std header,  but we  reserve space for future*/
    ULONG  timestamp;
}IcmpHeader;
;
#define  STATUS_FAILED 0xFFFF
#define  MAX_PACKET 1024
#define  xmalloc(s) HeapAlloc(GetProcessHeap(),HEAP_ZERO_MEMORY,(s))
#define  xfree(p)   HeapFree(GetProcessHeap(),0,(p))
```

ping.c

```
USHORT checksum(USHORT *, int);
void fill_icmp_head(char *);
void decode_resp(char *,int ,struct sockaddr_in *);

void Usage(char *progname)
{
fprintf(stderr,"Usage:\n");
fprintf(stderr,"%s <host>\n",progname);
ExitProcess(STATUS_FAILED);
}
```

ping.c

```
int main(int argc, char **argv)
{
  WSADATA wsaData;
  SOCKET sockRaw;
  struct sockaddr_in dest,src;
  struct hostent *hp;
  int bread,datasize;
  int fromlen = sizeof(src);
  char *dest_ip;
  char *icmp_data;
  char *recvbuf;
  unsigned int addr=0;
  USHORT seq_no = 0;
```

ping.c

```
  if (WSAStartup(0x0101,&wsaData) != 0){
      fprintf(stderr,"WSAStartup failed: %d\n",GetLastError());
      ExitProcess(STATUS_FAILED);
  }
  if (argc <2 ) {
      Usage(argv[0]);
  }
 if((sockRaw=socket(AF_INET,SOCK_RAW,IPPROTO_ICMP))==INVALID_SOCKET)
{
      fprintf(stderr,"WSAStartup failed: %d\n",GetLastError());
      ExitProcess(STATUS_FAILED);
  }
  memset(&dest,0,sizeof(dest));
```

ping.c

```
hp = gethostbyname(argv[1]);

if (hp!=NULL){
        memcpy(&(dest.sin_addr),hp->h_addr,hp->h_length);
        dest.sin_family = AF_INET;
        dest_ip = inet_ntoa(dest.sin_addr);
}
else{
        fprintf(stderr,"Unable to resolve %s\n",argv[1]);
        ExitProcess(STATUS_FAILED);
}
```

ping.c

```
    datasize=sizeof(IcmpHeader);                    // 12
    icmp_data = xmalloc(MAX_PACKET);        // 1024
    recvbuf = xmalloc(MAX_PACKET);                  // 1024

if (!icmp_data) {
    fprintf(stderr,"HeapAlloc failed %d\n",GetLastError());
    ExitProcess(STATUS_FAILED);
}

memset(icmp_data,0,MAX_PACKET);
fill_icmp_head(icmp_data);
```

ping.c

```
while(1) {
int bwrote;
((IcmpHeader*)icmp_data)->i_cksum = 0;
((IcmpHeader*)icmp_data)->timestamp = GetTickCount();
((IcmpHeader*)icmp_data)->i_seq = seq_no++;
((IcmpHeader*)icmp_data)->i_cksum=checksum((USHORT*)icmp_data,  sizeof(IcmpHeader));
bwrote = sendto(sockRaw,icmp_data,datasize,0,(struct sockaddr*)&dest,sizeof(dest));
if (bwrote == SOCKET_ERROR){
  fprintf(stderr,"sendto failed: %d\n",WSAGetLastError());
  ExitProcess(STATUS_FAILED);
 }
 if (bwrote < datasize ) {
  fprintf(stdout,"Wrote %d bytes\n",bwrote);
 }
```

ping.c

```
bread = recvfrom(sockRaw,recvbuf,MAX_PACKET,0,(struct sockaddr*)&from,&fromlen);
 if (bread == SOCKET_ERROR){
   if (WSAGetLastError() == WSAETIMEDOUT)
 { printf("timed out\n");
     continue; }
fprintf(stderr,"recvfrom failed: %d\n",WSAGetLastError());
   perror("revffrom failed.");
   ExitProcess(STATUS_FAILED);
  }
  decode_resp(recvbuf,bread,&from);
  sleep(1000);
 }
  closesocket(sockRaw);
  xfree(icmp_data);
  xfree(recvbuf );
  WSACleanup();    /* clean up ws2_32.dll */
  return 0;
}
```

ping.c

```
void fill_icmp_head(char * icmp_data){

  IcmpHeader *icmp_hdr;

  icmp_hdr = (IcmpHeader*)icmp_data;
  icmp_hdr->i_type = ICMP_ECHO;
  icmp_hdr->i_code = 0;
  icmp_hdr->i_cksum = 0;
  icmp_hdr->i_id = (USHORT)GetCurrentProcessId();
  icmp_hdr->i_seq = 0;
 }
```

ping.c

```
/*      The response is an IP packet. We must decode    */
/*      the IP header to locate the ICMP data           */

void decode_resp(char *buf, int bytes,struct sockaddr_in *from)
 {
        IpHeader *iphdr;
        IcmpHeader *icmphdr;
        unsigned short iphdrlen;

        iphdr = (IpHeader *)buf;
        iphdrlen = iphdr->h_len * 4 ; // number of 32-bit words *4 = bytes

        if (bytes  < iphdrlen + ICMP_MIN) {
                printf("Too few bytes from %s\n",inet_ntoa(from->sin_addr));
        }
```

ping.c

```
        icmphdr = (IcmpHeader*)(buf + iphdrlen);

    if (icmphdr->i_type != ICMP_ECHOREPLY) {
        fprintf(stderr,"non-echo type %d recvd\n",icmphdr->i_type);
        return;
    }
    if (icmphdr->i_id != (USHORT)GetCurrentProcessId())
    {
        fprintf(stderr,"someone else's packet!\n");
        return ;
     }
    printf("%d bytes from %s:",bytes, inet_ntoa(from->sin_addr));
    printf(" icmp_seq = %d. ",icmphdr->i_seq);
    printf(" time: %d ms ",GetTickCount()-icmphdr->timestamp);
    printf("\n");
}
```

ping.c

```
 USHORT checksum(USHORT *buffer, int size)
{
 unsigned long cksum=0;

 while(size >1) {
        cksum+=*buffer++;
        size -=sizeof(USHORT);
 }

 if(size) {
        cksum += *(UCHAR*)buffer;
 }

 cksum = (cksum >> 16) + (cksum & 0xffff);
 cksum += (cksum >>16);
 return (USHORT)(~cksum);
}
```