59

User Datagram Protocol—UDP

Aleksander Malinowski
Bradley University

Bogdan M. Wilamowski
Auburn University

59.1 Introduction

User datagram protocol (UDP) is a part of TCP/IP suite [STD6,C02-1,F10,GW03,PD07]. It provides full transport layer services to applications. It belongs to the transport layer in the TCP/IP suite model, as shown in Figure 59.1. UDP provides a connection between two processes at both ends of transmission. This connection is provided with minimal overhead, without flow control or acknowledgment of received data. The minimal error control is provided by ignoring (dropping) received packets that fail the checksum test.

The UDP is standardized by *Internet Society*, which is an independent international not-for-profit organization. A complete information about UDP-related standards is published by *RFC Editor* [RFC]. Several relevant RFC articles as well as further reading materials are listed at the end of this article.

59.2 Protocol Operation

Underlying IP provides data transport through the network between two hosts. UDP allows addressing particular processes at each host so that incoming data is handled by a particular application. While underlying IP provides logical addressing and is responsible for data transmission through networks [STD5], UDP provides additional addressing that allows setting up connections for multiple applications and services on the same end. Addressing the processes is implemented by assigning a number to each connection to the network, which is called a port number. Figure 59.2 illustrates the concept of port numbers and shows the difference between IP and UDP addressing. UDP packets are also called *datagrams*. Each datagram is characterized by four parameters—the IP address and the port number at each end of the connection [STD6]. Certain class of destination IP addresses may not describe individual computers but a group of processes on multiple hosts that are to receive the same datagram [STD2,RFC3171]. Computers that need to receive multicast messages must "join" the multicast group associated with a particular multicast IP address.

Although the term connection is used here, unlike in case of transmission control protocol (TCP), there is no persistent connection between the two processes. The data exchange may follow certain

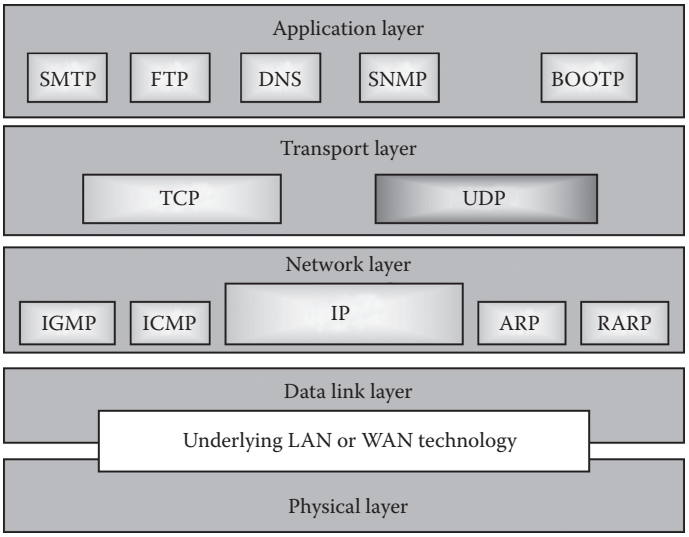

FIGURE 59.1 UDP and TCP/IP in the TCP/IP suite model.

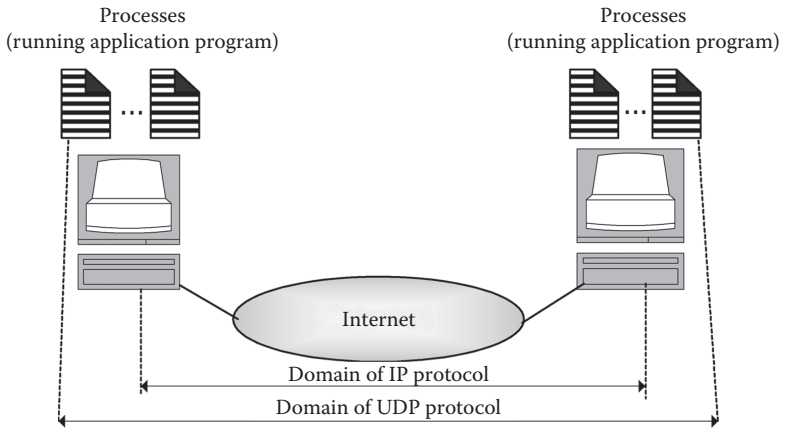

FIGURE 59.2 Comparison of IP and UDP addressing.

pattern or protocol but there is no guarantee that all data reach the destination and that the order of their receiving would be the same as the order they were sent. The client–server approach, which is usually used to describe interaction between two processes, still can be used. The process that may be identified as waiting for the data exchange is called the *server*. The process that may be identified as one that initializes that exchange is called the *client*.

59.2.1 UDP Datagram

AQ1 UDP have a constant size 8 byte header prepended to the transmitted data, as shown in Figure 59.3 [STD6]. The meaning of each header field is described below:

- *Source port address*: This is a 16 bit field that contains a port number of the process that sends options or data in this segment.
- *Destination port address*: This is a 16 bit field that contains a port number of the process that is supposed to receive options or data carried by this segment.

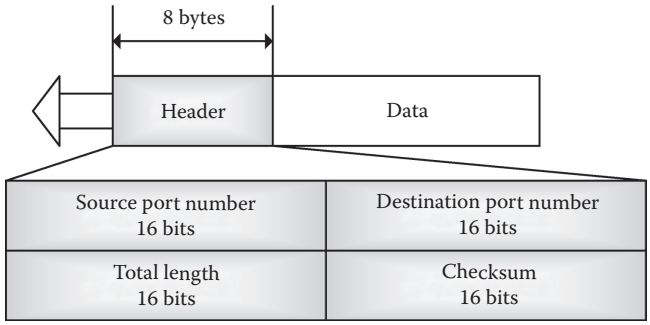

FIGURE 59.3 User datagram format.

- *Total length*: This is a 16 bit field that contains the total length of the packet. Although the number could be in the range from 0 to 65,535, the minimum length is 8 bytes that correspond to the packet with the header and no data. The maximum length is 65,507 because 20 bytes are used by the IP header and 8 bytes by the UDP header. Thus, this information is redundant to the packet length stored in the IP header. An extension to UDP that allows for transmission of larger datagrams over IPv6 packets has been standardized [RFC2675] [RFC2147]. In such a case, the UDP header specified total length is ignored.
- *Checksum*: This 16 bit field contains the checksum. The checksum is calculated by
  - Initially filling it with 0's.
  - Adding a pseudoheader with information from IP, as illustrated in Figure 59.4.
  - Treating the whole segment with the pseudoheader prepended as a stream of 16 bit numbers. If the number of bytes is odd, 0 is appended at the end.
  - Adding all 16 bit numbers using 1's complement binary arithmetic.
  - Complementing the result. This complemented result is inserted into the checksum field.
- *Checksum verification*: The receiver calculates the new checksum for the received packet that includes the original checksum, after adding the so-called pseudoheader (see Figure 59.4). If the new checksum is nonzero, then the datagram is corrupt and is discarded.

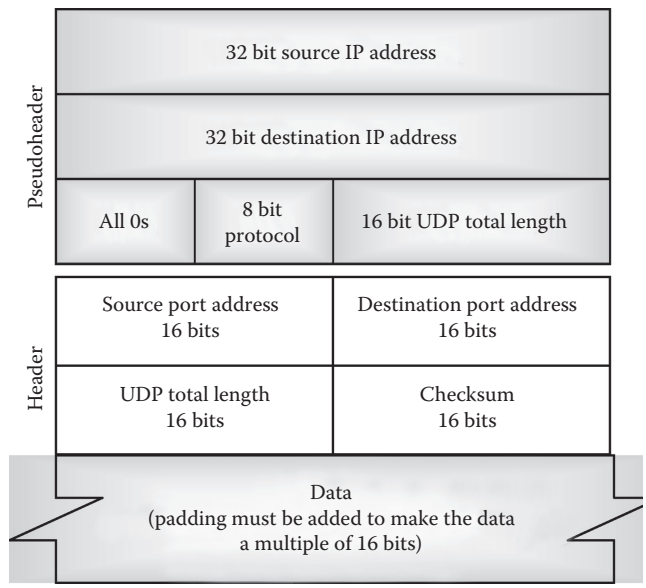

FIGURE 59.4 Pseudoheader added to the UDP datagram.

In case of UDP, the use of checksum is optional. If it is not calculated then the field is filled with 0's. The receiver can determine whether checksum was calculated by inspecting the field. Even in case the checksum is 0 the field does not contain 0 as the calculated checksum is complemented at the end of the process and negative 0 (the field filled with 1's) is stored.

59.2.2 Port Number Assignments

In order to provide platform for an independent process addressing on a host, each connection of the process to the network is assigned a 16 bit number. There are three categories of port numbers: well-known (0-1,023), registered (1,024-49,151), and ephemeral (49,152-6,553). *Well-known ports* have been historically assigned to common services. Table 59.1 shows well-known ports commonly used by UDP. Some operating systems require that the process that utilizes these ports must have administrative privileges. This requirement was historically created to avoid hackers running server imposters on multiuser systems. Well-known ports are registered and controlled by Internet Assigned Numbers Authority (IANA) [IANA] [STD2]. *Registered ports* are also registered by IANA to avoid possible conflicts among different applications attempting to use the same port for listening to incoming connections. *Ephemeral* ports are also called dynamic ports. These ports are used by outgoing connections that are typically assigned to the first available port above 49,151. Some operating systems may not follow IANA recommendations and treat the registered ports range as ephemeral. For example, BSD uses ports 1,024 through 4,999 as ephemeral ports, many Linux kernels use 32,768–61,000, Windows use 1,025–5,000, while FreeBSD follows IANA port range.

AQ2

59.2.3 Connectionless Service—Flow and Error Control

As it was already outlined, UDP provides connectionless communication. Each datagram is not related to another datagrams coming earlier or later from the same source. The datagrams are not numbered. There is no need for establishing or tearing down the connection. In the extreme case, just one datagram might be sent in the process of data exchange between the two processes.

UDP does not provide any flow control. The receiving side may be overflowed with incoming data. Unlike in TCP, which has a windowing mechanism, there is no way to control the sender. There is no

TABLE 59.1 Common Well-Known Ports Used with UDP

Port	Protocol	Description	RFC/STD #
7	Echo	Echoes a received datagram back to its sender	STD20
9	Discard	Discards any datagram that is received	STD21
13	Daytime	Returns the date and the time in ASCII string format	STD25
19	Chargen	Returns a string of characters of random length 0 to 512 characters	STD22
37	Time	Returns the current time in seconds since 1/1/1900 in a 32 bit format	STD26
53	Nameserver	Domain Name Service	STD13
67	DHCP	Dynamic Host Configuration Protocol Server	RFC2131
68	DHCP	Dynamic Host Configuration Protocol Client	RFC2131
69	TFTP	Trivial File Transfer	STD33
111	RPC	SUN Remote Procedure Call	RFC5531
123	NTP	Network Time Protocol	STD12
161	SNMP	Simple Network Management Protocol	STD62
162	SNMP	Simple Network Management Protocol (trap)	STD62
213	IPX	Internetwork Packet Exchange Network Layer Protocol over IP	RFC1234
520	RIP	Routing Information Protocol	SDT56
546	DHCPv6	DHCPv6 client	RFC3315
547	DHCPv6	DHCPv6 server	RFC3315

```
/* * server.c - code for example server program that uses UDP * */
#ifndef unix
#include <winsock2.h>
/* also include Ws2_32.lib library in linking options */
#else
#define closesocket close
#define SOCKET int
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <unistd.h>
/* also include xnet library for linking; on command line add: -lxnet */
#endif

#include <stdio.h>
#include <string.h>

#define PORT 1200             /* server TCP port number             */
int main()
{
    SOCKET  sd;               /* socket descriptor - (integer)      */
    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     alen;             /* length of address                  */
    int     port;             /* protocol port number               */

#ifdef WIN32
    WSADATA wsaData;
    if(WSAStartup(0x0101, &wsaData)!=0)
    { fprintf(stderr, "Windows Socket Init failed: %d\n", GetLastError()); exit(1); }
#endif

    memset((char *)&sad,0,sizeof(sad)); /* clear sockaddr structure */
    port = PROTOPORT;                   /* use predefined port number */
    sad.sin_port = htons((u_short)port); /* set server port number  */
    sad.sin_family = AF_INET;           /* set family to Internet   */
    sad.sin_addr.s_addr = INADDR_ANY;   /* set the local IP address */

    /* Map UDP transport protocol name to protocol number */
    ptrp = getprotobyname("udp");
    if ( ptrp == 0) { fprintf(stderr, "cannot map \"udp\" to number"); exit(1); }

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

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

    {
        char buf[1000];       /* buffer for string the server sends  */
        int  visits = 0;      /* counts client connections           */
        int  n;               /* number of characters received       */
        int  m;               /* number of characters sent back      */

        while (1) { /* Main server loop - accept and handle requests */
            alen = sizeof(cad);
            n = recvfrom(sd,buf,sizeof(buf),0,(struct sockaddr*)&cad,&alen);
            if (n<0) {
                fprintf(stderr,"Error in receiving\n"); continue;
            } else if(n>=0) { /* We could recevie a useful emtpy packet */
                visits++;
                sprintf(buf,"This server has been contacted %d time%s\n",
                    visits,visits==1?".":"s.");
                m = sendto(sd,buf,strlen(buf)+1,0,(struct sockaddr*)&cad,alen);
                if(m<0) { fprintf(stderr,"Error in sending"); continue; }
            }
        }
    }

#ifdef WIN32
    WSACleanup();                       /* release use of winsock.dll */
#endif
    return(0);
}
```

FIGURE 59.5 Sample C program that illustrates a UDP server.

```
/* * code for example client program that uses UDP * */
#ifndef unix
#include <winsock2.h>
/* also include Ws2_32.lib library in linking options */
#else
#define closesocket close
#define SOCKET int
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <unistd.h>
/* also include xnet library for linking; on command line add: -lxnet */
#endif

#include <stdio.h>
#include <string.h>

#define PORT 1200             /* server TCP port number             */
char    hostname[] = "localhost";    /* server host name            */
int main()
{
    SOCKET  sd;              /* socket descriptor - (integer)       */
    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     alen;            /* length of address                   */
    char    host[256];       /* pointer to host name                */

#ifdef WIN32
    WSADATA wsaData;
    if(WSAStartup(0x0101, &wsaData)!=0)
    { fprintf(stderr, "Windows Socket Init failed: %d\n", GetLastError()); exit(1); }
#endif

    /* prepare the client socket information */
    memset((char *)&sad,0,sizeof(sad)); /* clear sockaddr structure */
    sad.sin_family = AF_INET;         /* set family to Internet     */

    /* Convert host name to equivalent IP address and copy to sad. */
    strcpy(host, hostname);
    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);

    sad.sin_port = htons((u_short)PORT); /* use the defined port number */
    alen = sizeof(sad);

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

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

    {   int  n;              /* number of characters received       */
        int  m;              /* number of characters sent back      */
        char buf[1000];      /* buffer for data from the server     */

        /* Send data to socket in order to request reply. */
        m = 1;              /* sned zero bytes */
        m = sendto(sd,buf,m,0,(struct sockaddr*)&sad,alen);
        if(m<0) {
            fprintf(stderr,"Error in sending");
        } else {
            /* Read data from socket and write to user's screen. */
            n = recvfrom(sd,buf,sizeof(buf),0,(struct sockaddr*)&sad,&alen);
            if (n > 0) { buf[n]='\0'; printf("%s", buf); }
        }
    }
    closesocket(sd);

#ifdef WIN32
    WSACleanup();                     /* release use of winsock.dll */
#endif
    return(0);
}
```

FIGURE 59.6 Sample C program that illustrates a UDP client.

```
/* * Multicasting listener (server) * */
#ifndef unix
#include <winsock2.h>
#include <ws2tcpip.h>
/* also include Ws2_32.lib library in linking options */
#else
#define closesocket close
#define SOCKET int
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <unistd.h>
/* also include xnet library for linking; on command line add: -lxnet */
#endif

#include <stdio.h>
#include <stdlib.h>

#define DEFAULT_IP "224.2.2.2"
#define DEFAULT_PORT 60001
#define MAX_MSG 100

int main(int argc, char *argv[]) {
    SOCKET          sd;
    int             rc, n;
    struct sockaddr_in cliAddr, servAddr;
    int             cliLen;
    struct in_addr  mcastAddr;
    struct ip_mreq  mreq;
    struct hostent *h;
    unsigned short  p;
    int             ttl;
    int             one;
    char            msg[MAX_MSG+1];

#ifdef WIN32
    WSADATA wsaData;
    if(WSAStartup(0x0101, &wsaData)!=0)
    { fprintf(stderr, "Windows Socket Init failed: %d\n", GetLastError()); exit(1); }
#endif

    /* get mcast address to listen to */
    if(argc>3) {
        printf("usage : %s [mcast address=%s] [port number=%d]\n",
            argv[0], DEFAULT_IP, DEFAULT_PORT); exit(0);
    } else if (argc>2) {
        h=gethostbyname(argv[1]); p =(unsigned short)atol(argv[2]);
    } else if (argc>=2) {
        h=gethostbyname(argv[1]);
    } else {
        h=gethostbyname(DEFAULT_IP); p=DEFAULT_PORT;
    }
    if(h==NULL) { printf("%s : unknown group %s\n", argv[0], argv[1]); exit(1); }
    if(p==0)    { printf("%s : invalid port %s\n", argv[0], argv[2]); exit(1); }
    memcpy((char *) &mcastAddr, h->h_addr_list[0], h->h_length);

    /* check given address is multicast */
    if(!IN_MULTICAST(ntohl(mcastAddr.s_addr)))
    { printf("%s : given address '%s' is not multicast\n",
        argv[0], inet_ntoa(mcastAddr)); exit(1); }

    /* create socket */
    sd = socket(AF_INET,SOCK_DGRAM,0);
    if(sd<0) { printf("%s : cannot create socket\n", argv[0]); exit(1); }

    /* allow multiple bind port - multiple sockets listening simultaneously */
    one = 1; setsockopt(sd, SOL_SOCKET, SO_REUSEADDR, (char *)&one, sizeof(one));

    /* bind port */
    servAddr.sin_family=AF_INET;
    servAddr.sin_addr.s_addr=htonl(INADDR_ANY);
    servAddr.sin_port=htons(p);
    if(bind(sd,(struct sockaddr *) &servAddr, sizeof(servAddr))<0)
    { printf("%s : cannot bind port %d\n", argv[0], p); exit(1); }
```

FIGURE 59.7 Sample C program that illustrates a UDP server utilizing multicasting.

(*continued*)

```
    /* set multicast TTL - range */
    ttl = 1; /* this allows to confine the range of transmission to a network segment */
    rc = setsockopt(sd, IPPROTO_IP, IP_MULTICAST_TTL, (char *)&ttl, sizeof(ttl));
    if (rc<0) { printf("%s : cannot set ttl = %d\n", argv[0], ttl); exit(1); }

    /* join multicast group */
    mreq.imr_multiaddr.s_addr=mcastAddr.s_addr;
    mreq.imr_interface.s_addr=htonl(INADDR_ANY);
    rc = setsockopt(sd, IPPROTO_IP, IP_ADD_MEMBERSHIP, (char *)&mreq, sizeof(mreq));
    if(rc<0) { printf("%s : cannot join multicast group %s\n", argv[0],
inet_ntoa(mcastAddr)); exit(1); }
    printf("%s : listening to mgroup %s:%d\n", argv[0], inet_ntoa(mcastAddr), p);

    /* infinite server loop */
    while(1) {
        cliLen=sizeof(cliAddr);
        n = recvfrom(sd, msg, MAX_MSG, 0, (struct sockaddr *)&cliAddr, &cliLen);
        if(n<0) { printf("%s : cannot receive data\n", argv[0]); continue; }
        msg[n]='\0';

        printf("%s : received from %s:%d: %s\n", argv[0], inet_ntoa(cliAddr.sin_addr),
ntohs(cliAddr.sin_port), msg);
    }/* end of infinite server loop */

    /* leave the group after you are done - actually never happens in this example */
    rc = setsockopt(sd,IPPROTO_IP,IP_DROP_MEMBERSHIP, (char *)&mreq, sizeof(mreq));
    if(rc<0) { printf("%s : cannot leave multicast group %s\n", argv[0],
inet_ntoa(mcastAddr)); exit(1); }

#ifdef WIN32
    WSACleanup();                    /* release use of winsock.dll */
#endif
    return 0;
}
```

FIGURE 59.7 (continued)

other error control than the checksum discussed above. The sender cannot be requested to resend any datagram. However, an upper level protocol that utilizes UDP may implement some kind of control. In that case, unlike in TCP, no data is repeated by resending the same datagram. Instead, the communicating process sends a request to send some information again. Trivial file transfer protocol (TFTP) is a very good example of that situation [STD33]. Since it has its own higher level flow and error control, it can use UDP as a transport layer protocol instead of TCP. Other examples of UDP utilization are simple network management protocol (SNMP) [STD62] or any other protocol that requires only simple short request-response communication.

59.3 Programming Samples

Figures 59.5 and 59.6 contain two C programs that demonstrate use of UDP for communication using
AQ3 client–server paradigm [C02-3]. Socket API is a de facto standard. Programs use conditional compilation directives that include minor variations in the library headers needed for operation under Unix/Linux and Windows. Function calls should be modified, if desired, to run under other operating systems or with proprietary operating systems for embedded devices. Minor modifications need to be performed when utilizing underlying IPv6 API, as described in [RFC3493]. Figure 59.5 shows the server program that waits for incoming datagram from the client program listed in Figure 59.6. Because UDP is connectionless, the server is capable of servicing data from several clients almost simultaneously. In case a distinction is needed among clients, this must be implemented in the code of the server program. Programs show all steps necessary to set up, use, and tear down a UDP-based exchange of data.

Figures 59.7 and 59.8 contain another pair of C programs that demonstrate the use of UDP for communication using multicasting. Several server programs listed in Figure 59.7 receive one message sent by the client program listed in Figure 59.8. The server must subscribe to the multicast IP address, i.e. "join the multicast group" in order to receive data. In case of a client that only sends multicast data this step may be omitted.

Internet socket is an end point of a bidirectional interprocess communication flow across an IP-based computer network. Interface to the socket library is standardized only on Unix/Linux and Windows. Other operating systems including various embedded systems may employ different functions to carry out the same functionality. Programs of the same functionality implemented in other languages may combine several steps into calling a single function. This is especially true for the socket initialization and connection set up.

```
/* * Multicasting sender (client) * */
#ifndef unix
#include <winsock2.h>
#include <ws2tcpip.h>
/* also include Ws2_32.lib library in linking options */
#else
#define closesocket close
#define SOCKET int
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <unistd.h>
/* also include xnet library for linking; on command line add: -lxnet */
#endif

#include <stdio.h>
#include <stdlib.h>

/* #define DEFAULT_IP "224.2.2.2" */
/* #define DEFAULT_PORT 60001 */

int main(int argc, char *argv[]) {

    SOCKET          sd;
    int             rc, i;
    struct sockaddr_in cliAddr, servAddr;
    struct in_addr  mcastAddr;
    struct ip_mreq  mreq;
    struct hostent *h;
    unsigned short  p;
    unsigned long   addr;
    unsigned char   ttl;
    int             one;

#ifdef WIN32
    WSADATA wsaData;
    if(WSAStartup(0x0101, &wsaData)!=0)
    { fprintf(stderr, "Windows Socket Init failed: %d\n", GetLastError()); exit(1); }
#endif

    if(argc<4) { printf("usage %s <maddress> <port> <data1> <data2> ... <dataN>\n",
argv[0]); exit(1); }

    h = gethostbyname(argv[1]);
    if(h==NULL) { printf("%s : unknown host %s\n", argv[0], argv[1]); exit(1); }
    p =(unsigned short)atol(argv[2]);
    if(p==0) { printf("%s : invalid port %s\n", argv[0], argv[2]); exit(1); }
    memcpy((char *) &mcastAddr, h->h_addr_list[0], h->h_length);

    /* check given address is multicast */
    if(!IN_MULTICAST(ntohl(mcastAddr.s_addr))){
        printf("%s : given address '%s' is not multicast\n", argv[0],
                inet_ntoa(mcastAddr)); exit(1); }

    /* create socket */
    sd = socket(AF_INET,SOCK_DGRAM,0);
    if(sd<0) { printf("%s : cannot create socket\n", argv[0]); exit(1); }


    /* enabling multicasting on a network interface is required only if sending data */

    /* allow multiple bind port to allow multiple instances on the same computer */
    one = 1; setsockopt(sd, SOL_SOCKET, SO_REUSEADDR, (char *)&one, sizeof(one));

    /* bind port */
    cliAddr.sin_family=AF_INET;
    cliAddr.sin_addr.s_addr=htonl(INADDR_ANY);
    cliAddr.sin_port=htons(p);
    if(bind(sd,(struct sockaddr *) &cliAddr, sizeof(cliAddr))<0)
    { printf("%s : cannot bind port %d\n", argv[0], p); exit(1); }
```

FIGURE 59.8 Sample C program that illustrates a UDP client utilizing multicasting.

(*continued*)

```
    /* set multicast TTL - range */
    ttl = 1;
    rc = setsockopt(sd, IPPROTO_IP, IP_MULTICAST_TTL, (char *)&ttl, sizeof(ttl));
    if (rc<0) { printf("%s : cannot set ttl = %d\n", argv[0], ttl); exit(1); }

    /* join multicast group */
    mreq.imr_multiaddr.s_addr=mcastAddr.s_addr;
    mreq.imr_interface.s_addr=htonl(INADDR_ANY);
    rc = setsockopt(sd, IPPROTO_IP, IP_ADD_MEMBERSHIP, (char *)&mreq, sizeof(mreq));
    if(rc<0) {
        printf("%s : cannot join multicast group %s\n",
                argv[0], inet_ntoa(mcastAddr)); exit(1); }

    servAddr.sin_family = h->h_addrtype;
    memcpy((char *) &servAddr.sin_addr.s_addr, h->h_addr, h->h_length);
    servAddr.sin_port = htons(p);

    printf("%s : sending data on multicast group %s (%s) port %d\n", argv[0],
        h->h_name, inet_ntoa(*(struct in_addr *) h->h_addr_list[0]),
        ntohs(servAddr.sin_port) );

    /* send data */
    for(i=3;i<argc;i++) {
        rc = sendto(sd, argv[i], strlen(argv[i])+1, 0,
                    (struct sockaddr *) &servAddr, sizeof(servAddr));

        if (rc<0) {
          printf("%s : cannot send data number %d\n", argv[0], i-2); break;
        } else {
          printf("%s : sent %d bytes in message %d\n", argv[0], strlen(argv[i])+1, i-2);
        }

    }/* end for */

    closesocket(sd);
#ifdef WIN32
    WSACleanup();                       /* release use of winsock.dll */
#endif
    exit(0);
}
```

FIGURE 59.8 (continued)

References

[C02–1] D. Comer, I*nternetworking with TCP/IP Vol. 1: Principles, Protocols, and Architecture*, 5th edn., Prentice Hall, Upper Saddle River, NJ, 2005.

[C02–3] D. Comer et al., *Internetworking with TCP/IP*, Vol. III: *Client-Server Programming and Applications, Linux/Posix Sockets Version*, Prentice Hall, Upper Saddle River, NJ, 2000.

[F10] B.A. Forouzan, *TCP/IP Protocol Suite*, 4th edn., McGraw-Hill, New York, 2010.

[GW03] A. Leon-Garcia, I. Widjaja, *Communication Networks*, 2nd edn., McGraw-Hill, New York, 2003.

[IANA] Web Site for *IANA - Internet Assigned Numbers Authority*, http://www.iana.org/

[PD07] L.L. Peterson, B.S. Davie, *Computer Networks: A System Approach*, 4th edn., Morgan Kaufmann Publishers, San Francisco, CA, 2007.

[RFC] Web Site for *RFC Editor*, http://www.rfc-editor.org/

[RFC2147] D. Borman et al., TCP and UDP over IPv6 Jumbograms, May 1997. http://www.rfc-editor.org/

[RFC2675] D. Borman et al., IPv6 Jumbograms, August 1999. http://www.rfc-editor.org/

[RFC3171] Z. Albana, K. Almeroth, D. Meyer, M. Schipper, IANA Guidelines for IPv4 Multicast Address Assignments, August 2001. http://www.rfc-editor.org/

[RFC3493] R. Gilligan et al., Basic Socket Interface Extensions for IPv6, February 2003. http://www.rfc-editor.org/

[STD2] J. Reynolds, J. Postel, Assigned Numbers, October 1994. http://www.rfc-editor.org/

[STD5] J. Postel, Internet Protocol, September 1981. http://www.rfc-editor.org/

[STD6] RFC0768/STD0006, J. Postel, User Datagram Protocol, August 1980. http://www.rfc-editor.org/

[STD33] K. Sollins, The Tftp Protocol (Revision 2), July 1992. http://www.rfc-editor.org/

[STD62] D. Harrington, R. Presuhn, B. Wijnen, An Architecture for Describing Simple Network Management Protocol (SNMP) Management Frameworks, December 2002. http://www.rfc-editor.org/