Chapter 11 – UDP (User Datagram Protocol)

A transport layer (i.e. end-to-end or application-to-application) protocol

Service characteristics

Unreliable
- May lose packets
- Generally will not duplicate or deliver damaged packets

Connectionless
- Each application request to send must contain full destination address

Not flow controlled in the general case
- Maximum rate transmission can lead to huge packet loss rates

If the destination is on the same LAN as the source transmission may (or may not) be flow controlled by the system software

UDP’s assigned protocol number is 17.
UDP is described in RFC 768

"Internal" users of UDP

- RIP – The routing information protocol
- NFS – The network file system
- DNS – The domain name services

Packet contents

- IP header
- UDP header
- UDP data

UDP header format

```
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| Source Port | Destination Port |
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| Length | Checksum |
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| data octets ... |
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
```

User Datagram Protocol Header
Port numbers are used in routing to a specific application

Demultiplexing of incoming data

First by protocol number
Second by port number
Pseudo-connections and interface level bindings complicate the issue and solutions may be implementation dependent.

So TCP and UDP can use identical port numbers for different applications
Inclusion of the source port in the header makes it possible for the receiver to be able to reply to the sender

Length is the length of the UDP header + data in the packet.

Checksum

covers the UDP IP pseudo–header, UDP header, and the UDP data.
is computed in usual IP way.

Here is an IP pseudo header

```
0 7 8 15 16 23 24 31
+------------------------+
| IP source address |
+------------------------+
| IP destination address |
+------------------------+
| zero |protocol| UDP length |
+------------------------+
```

Special checksum values

| 0x0000 | Checksum was not computed by sender (UDP checksum is optional) |
| 0xffff | Checksum was computed but are all 0’s. |

Sending UDP datagrams

The NETAPI function sendto is used

```c
int sendto( int s, /* UDP socket */
char *msg, /* -> data to send */
int len, /* # bytes to send */
int flags, /* 0 works well! */
struct sockaddr_in *to, /* Dest address */
int tolen) /* Sizeof addr struct */
```
Datagram send example

/* dgsend.c */
/* Send a UDP datagram to a specified host/port */

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>

char buf[1024*64] = "This is some data";
struct hostent *hp;
struct hostent *gethostbyname();
struct sockaddr_in name;

main(argc, argv)
int argc;
char *argv[];
{
  int i;
  unsigned char c;
  int sock;
  int status;
  int msglen;

  sock = socket(PF_INET, SOCK_DGRAM, 0);
  if (sock < 0)
  {
    printf("Socket create failed \n");
    exit(1);
  }

  Ask DNS resolver to resolve the destination host name

  hp = gethostbyname(argv[1]);
  if (hp == 0)
  {
    printf("Host %s not found\n", argv[1]);
    exit(1);
  }

  printf("Length = %d | Addr = ", hp->h_length);
  for (i = 0; i < hp->h_length; i++)
  {
    c = *((char *)&name.sin_addr + i);
    printf(" %2x", c);
  }
  printf("\n");
Fill in the `sockaddr_in` structure and send the packet

```c
bcopy((char *)hp->h_addr, (char *)&name.sin_addr, hp->h_length);
name.sin_family = AF_INET;
name.sin_port = htons(atoi(argv[2]));
msglen = atoi(argv[3]);

status = sendto(sock, buf, msglen, 0, (struct sockaddr *)&name, sizeof(name));

printf("status = %d \n", status);
```

Sample output

```
networks/unixcom ==> dgsend jmw2 21234 9002
Length = 4 | Addr = 82 7f 30 71
status = -1

networks/unixcom ==> dgsend jmw2 21234 9000
Length = 4 | Addr = 82 7f 30 71
status = 9000

networks/unixcom ==> dgsend jmw2 21234 9001
Length = 4 | Addr = 82 7f 30 71
status = -1

networks/unixcom ==> dgsend jmw 20001 400
Length = 4 | Addr = 82 7f 30 18
status = 400
```

Items to note:

`sendto` returns the number of bytes sent
for this SunOS 4.3.1 system the maximum length is 9000 bytes
for Linux 2.0 and SunOS 5.? the maximum length is 65507

```
20 IP header len
8  UDP header len
65507
----------
65535 or 0xffff Max IP datagram length.
```

your mileage may vary.
Receiving UDP datagrams

The NETAPI function `recvfrom` is used

```c
int recvfrom(
    int s,            /* Socket handle     */
    char *buf,        /* Input buffer       */
    int len,          /* Buffer length      */
    int flags,        /* 0 works           */
    struct sockaddr *from, /* Who sent it.       */
    int *fromlen)     /* Sender addr len    */
```

Sample datagram receive application (This page is basically the same as for send.)

```c
/* dgrecv.c */
/* Demonstates the use of recvfrom for datagram transfer */

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>

char buf[100];
struct hostent *hp;
struct hostent *gethostbyname();
struct sockaddr_in name;
struct sockaddr_in sname;
int namelen;

main(argc, argv)
int argc;
char *argv[];
{
    int i;
    unsigned char c;
    int sock;
    int netaddr;
    int status;

    /* Create datagram socket */

    sock = socket(PF_INET, SOCK_DGRAM, 0);
    if (sock < 0)
    {
        printf("Socket create failed \n");
        exit(1);
    }
```
/* Get host network address from command line parm */

    hp = gethostbyname(argv[1]);
    if (hp == 0)
    {
        printf("Host %s not found\n", argv[1]);
        exit(1);
    }
    bcopy((char *)hp->h_addr,
          (char *)&name.sin_addr, hp->h_length);

/* Fill in protocol family and port # from command line then */
/* bind the socket to the specified address. */

    name.sin_family = AF_INET;
    name.sin_port = htons(atoi(argv[2]));
    status = bind(sock, (struct sockaddr *)&name, sizeof(name));
    printf("Bind status = %d \n", status);
    if (status < 0)
        exit(1);

/* Wait for someone to send me a datagram */

    namelen = sizeof(struct sockaddr_in);
    status = recvfrom(sock, buf, sizeof(buf), 0,
                       (struct sockaddr *)&sname, &namelen);

    printf("status = %d \n", status);
    printf("Msg = %s \n", buf);

    printf("Sender net address = ");
    for (i = 0; i < hp->h_length; i++)
    {
        c = *((char *)&sname.sin_addr + i);
        printf(" %2x", c);
    }
    printf("\n");
    printf("Sender port address = %x \n", sname.sin_port);
}

Output from the program

networks/unixcom ==> dgrecv jmw 20001
 Bind status = 0
     status = 100
     Msg = This is some data
     Sender net address =  82 7f 30 18
     Sender port address = 8a2

Note that receiver must bind to some port if the sender is to know dest port addr.
Sender does not need to explicitly bind —
    a port 8a2 in the above example was implicitly allocated by the socket API.
Associating a local socket with a specific local IP address and port number

The `bind` function call specifies the local address components
- IP address
- Port address
- Address family (always set to AF_INET == 2 for TCP/IP)

Syntax
```c
status = bind(sock, (struct sockaddr_in *)&name, sizeof(name));
```

Local IP address categories
- Per ethernet interface
  - IP specific
  - IP broadcast
- Loopback
- Per point to point interface
  - IP specific address
  - All zeros (on SunOS 5.x at least ==> *)

Implications of the use of bound sockets in UDP

If a datagram arrives for a bound socket and is not directed to exactly the right address, it will be dropped!

If SO_REUSEADDR is set with setsockopt() then multiple processes can manage the different internet addresses using the same port number. Why??

The `netstat` command can be used to display the state of UDP, TCP and Unix domain sockets

```
networks/unixcom ==> dgrecv jmw 2222
netstat -a -n -f inet
```

```
Active Internet connections (including servers)
Proto Recv-Q Send-Q Local Address      Foreign Address
udp   0      0  *.1360                 *.*
udp   0      0  *.1351                 *.*
udp   0      0  130.127.48.24.2222     *.*
```

but if you were to modify dgrecv.c so that the IP address copy were removed as follows

```c
/* bcopy((char *)hp−>h_addr, (char *)&name.sin_addr, hp−>h_length); */
```

and you were to restart dgrecv and run netstat then you would see

```
udp   0      0  *.2222                 *.*
```

7
Exercise: On your favorite SunOS 5.X machine determine which of the following configurations allow a successful reception. Assume both processes run on the same machine and dgsend sends to the specified dest address and dgrecv binds to the specified source address.

<table>
<thead>
<tr>
<th>Dest IP address:</th>
<th>0</th>
<th>127.0.0.1</th>
<th>interface−addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source IP addr:</td>
<td>0</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td></td>
<td>127.0.0.1</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>interface−addr</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

Associating a local socket with a specific remote IP address and port number

In this case the "name" structure specifies a sender address (which might be local or remote) that the receiver is willing to receive from:

```c
status = connect(sock, (struct sockaddr_in *)&name, sizeof(name));
```

If a datagram arrives for a connected socket and is not from exactly the right address, it will be dropped.

Exercise: On your favorite SunOS 5.X machine complete the 3-dimensional version of the above matrix by adding the connect state of the receiver as the third dimension. If you have this info in your head you are probably within the top of 0.0000001% of humanity in TCP/IP administrivia.

Exercise: Can you wildcard IP address but fix the port — or wildcard the port but fix the IP?

Exercise: What happens if the port address is not set by the sender or the receiver?
Identifying the destination *struct sock*.

The low order bits of the destination port are used as an index to identify the correct hash chain. For each *struct sock* on the chain in which the local port matches the destination port a the value "score" denotes the number of fields matched. If all the above mentioned fields are matched, that *struct sock* is immediately accepted. Otherwise, the *struct sock* that matches largest number of fields is returned. A mismatch with a specified field is an immediate disqualifier.

```c
211 struct sock *udp_v4_lookup_longway(u32 saddr, u16 sport, u32 daddr, u16 dport, int dif)
212 {
213     struct sock *sk, *result = NULL;
214     unsigned short hnum = ntohs(dport);
215     int badness = -1;
216     for(sk = udp_hash[hnum & (UDP_HTABLE_SIZE - 1)]; sk != NULL; sk = sk->next) {
217         if(sk->num == hnum) {
218             int score = 0;
219             if(sk->rcv_saddr) {
220                 if(sk->rcv_saddr != daddr)
221                     continue;
222                 score++;
223             }
224             if(sk->daddr) {
225                 if(sk->daddr != saddr)
226                     continue;
227                 score++;
228             }
229             if(sk->dport) {
230                 if(sk->dport != sport)
231                     continue;
232                 score++;
233             }
234             if(sk->bound_dev_if) {
235                 if(sk->bound_dev_if != dif)
236                     continue;
237                 score++;
238             }
239             if(score == 4) {
240                 result = sk;
241                 break;
242             } else if(score > badness) {
243                 result = sk;
244                 badness = score;
245             }
246         }
247     }
248     return result;
249 }
```
**IP Fragmentation**

Fragmentation

May be done anywhere along the path
Some IP implementations (p.156 in the text) may guess a smaller size and periodically try to "go back" to using large sizes.
IPV6 doesn’t do fragmentation at all.
Reassembly is always done at the destination host

**IP Header**

```
|Version| IHL |Type of Service|          Total Length         |
| Identification |Flags|      Fragment Offset    |
|  Time to Live |    Protocol   |         Header Checksum       |
|                       Source Address                          |
|                    Destination Address                        |
|                    Options                    |    Padding    |
```

Internet Datagram Header

 Datagram length in bytes           Maximum of 64K
 Datagram id                         Incremented by 1 each time originating host transmits
 Fragmentation flags                 (Reserved, DF, MF)
 Fragment offset                     In 8 byte units
Determining the MTU (Max transmission Unit)

Also known as the *path MTU discovery mechanism*

See RFC 1191  
Send large packet with DF set  
Continually reduce using MTU returned by ICMP DUFR responses

```
0                   1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+
|     Type (3)  |    Code (4)   |          Checksum             |
+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+
|       Unused (0)              |      MTU of next hop          |
+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+
|       IP header + 1st 8 Bytes of original datagram            |
+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+−+
ICMP − Dest Unreachable fragmentation required
```

**Example of section 11.6**

In the diagram:

- **MTUS**
  - 1500
  - 1500
  - ?

- Solaris ────────┐ = netb ────────┐ = sun ────────┐ = bsdi
  ┼<---------------<---------------<---------------
    1500
    ?

- ENet ─────┐ = Slip ─────┐ = Enet
  ┼<---------------<---------------

With a 600 byte ping with DF sent from Solaris to bsdi (above diagram reversed from book):

- tcpdump (run on bsdi) shows
  - ping reaches bsdi
  - bsdi copies DF into the ICMP echo response
  - ping response is sent to solaris
  - ping reaches sun
  - icmp DUFR is received by bsdi (with MTU = 0!)

Morals of the story:

1. MTU may not be same on both ends of the same link
2. Value returned by ICMP may *not* indicate true MTU
3. If you are trying to diagnose this problem from *solaris* may the force be with you
Source host approaches to fragmentation mgmt (fig 11.8) in the book.

Source host tries to send with DF set.
Source host tries to do the fragmentation itself if transmission fails.
Source host periodically sets DF and tries to up the MTU to see if the MTU has increased.

If the intermediate routers support path MTU discovery mechanism
Use the MTU returned by the router.

In any case
Don’t try too often (10 minutes is a recommended retry minimum).

Bottom line:

Its nice to try to offload the routers but

Short packets are inefficient to send
If the small MTU lives near the end of the route its much better to let fragmentation occur there!
Interaction of UDP with ARP

Source host fragments  => multiple ARP’s sent (6 in 4.3 ms —> RFC suggests 1 / sec max)

Source host discards all but the last fragment of the datagram!

ARP/IP interaction rule:
Keep only the last packet to be sent to a specific dest while waiting for ARP reply.

Some hosts (BSD) may not return reassembly timeouts.

In this case it only got the last fragment and ICMP error msg is not required unless the first fragment is received.
But why send’em— original sender has no way to get it anyhow since it can’t receive ICMP messages!
So some implementations don’t send them at all.

Exercise: How does your favorite OS deal with this?

Datagram lengths

Maximum sent by most hosts is "somewhat" greater than 8192 (the size of NFS datagrams)
Absolute max is 65507 (as is the case with Linux and Solaris)

Options for receive if amount read is < amount actually sent

1 - Discard rest of datagram (BSD)
2 - Supply rest on next read. (SVRx == Solaris)

No standard exists.
UDP server architecture

/* udpcs.h */

typedef struct
{
    unsigned int magic;       /* Magic # = 0x1a2b3c4d */
    unsigned int ptype;       /* Packet type */
    unsigned int seq;         /* Packet seq # */
} apphdr_t;

#define PKT_CONN_REQ  101
#define PKT_CONN_ACK  102
#define PKT_CLOSE_REQ 103
#define PKT_CLOSE_ACK 104
#define PKT_DATA      128      /* This is a pure data frame */
#define PKT_DATA_POLL 129      /* Data frame + poll for status */
#define PKT_STAT_RESP 130      /* Status reply to poll */
#define UDPCS_MAGIC   0x1a2b3c4d

/* This structure maps a stat response */

typedef struct
{
    apphdr_t           apphdr;
    unsigned int thisseq;  /* Seq # of poll frame */
    unsigned int drops;    /* Total number of dropped packets */
} statresp_t;
/ * udpserv.c */
/* A udp server which serves by consuming data from a remote */
/* client and informing the client regarding the rate of */
/* packet loss. */
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#include <sys/time.h>
#include <sys/times.h>
#include <pthread.h>
#include <signal.h>
#include "udpcs.h"

char buf[1024];
struct hostent *hp;
struct hostent *gethostbyname();
struct sockaddr_in name;
struct sockaddr_in from;
struct sockaddr_in sname;
int namelen;

struct timeval time1;
struct timeval time2;
static struct timezone dummy = {0,0};

apphdr_t conn_resp;
apphdr_t closeResp;
statresp_t stat_resp;

main(argc, argv)
int argc;
char *argv[];
{
    int i;
    unsigned char c;
    int sock;
    int netaddr;
    int status;
    apphdr_t *apphdr;
    int connid = 0;

    /* Create datagram socket */
    sock = socket(PF_INET, SOCK_DGRAM, 0);
    if (sock < 0)
    {
        printf("Socket create failed \n");
        exit(1);
    }
}
/* Fill in protocol family and port # from command line then */
/* bind the socket to the specified address.               */

name.sin_family = AF_INET;
name.sin_port = htons(atoi(argv[1]));
status = bind(sock, (struct sockaddr *)&name, sizeof(name));
printf("Bind status = %d \n", status);
if (status < 0)
    exit(1);

/* Setup input header pointer and connection response buffer */
/* Header structure is defined in udpcs.h                  */

apphdr = (apphdr_t *)buf;
conn_resp.magic = htonl(UDPCS_MAGIC);
conn_resp.seq = 0;                /* Not used */
conn_respptype = htonl(PKT_CONN_ACK);

close_resp.magic = htonl(UDPCS_MAGIC);
close_resp.seq = 0;                /* Not used */
close_resptype = htonl(PKT_CLOSE_ACK);

stat_resp.apphdr.magic = htonl(UDPCS_MAGIC);
stat_resp.apphdr.seq = 0;          /* Not used */
stat_resp.apphdrptype = htonl(PKT_STAT_RESP);

/* Wait for someone to send me a connection request */

while (1)
{
    namelen = sizeof(struct sockaddr_in);
    status = recvfrom(sock, buf, sizeof(buf), 0,
                        (struct sockaddr *)&sname, &namelen);

    printf("status = %d \n", status);

    printf("Sender net address = ");
    for (i = 0; i < 4; i++)
    {
        c = *((char *)&sname.sin_addr + i);
        printf(" %2x", c);
    }
    printf("\n");
    printf("Sender port address = %x \n", ntohs(sname.sin_port));
/* Ensure that length, magic #, and type are correct */

if (status != sizeof(apphdr_t))
{
    printf("bad message length %d \n", status);
    continue;
}

if (ntohl(apphdr->magic) != UDPCS_MAGIC)
{
    printf("bad magic number %x \n", ntohl(apphdr->magic));
    continue;
}

if (ntohl(apphdr->ptype) != PKT_CONN_REQ)
{
    printf("bad packet type %x \n", ntohl(apphdr->ptype));
    continue;
}

/* If we get here we appear to have a valid connection */
/* request... Create a server to handle it.               */
/* Note that with UDP we must explicitly create a new */
/* socket and connect it to the remote client          */

status = fork();
if (status == 0)
{
    int newsock;

    newsock = socket(PF_INET, SOCK_DGRAM, 0);
    status = connect(newsock, &sname, namelen);

    /* Send a connect ack using the new socket... The client */
    /* will direct the data to be consumed to this new       */
    /* port address.                                        */
    write(newsock, &conn_resp, sizeof(apphdr_t));
    do_serv(newsock, connid, &sname);
}
connid++;
}
/**/  
/* This thread is used to recover from cases in which the client */  
/* dies before sending the close or the close gets lost. */  
/* A client who fails to send for 10 seconds is assumed dead */  

static int alive;

void *keepalive(void *p)  
{
    while (1)  
    {
        alive = 0;
        sleep(10);
        if (alive == 0)
            kill(getpid(), SIGKILL);
    }
}

/**/  
/* This is the function that actually performs the service */

#define BUF_SIZE 2048

int do_serv(
    int sock,
    int connid,
    struct sockaddr_in *sname)
{
    int len;
    int seq;
    int nextseq = 0;
    int drops = 0;
    int recvs = 0;
    double count = 0;
    double tu1;
    double tu2;
    pthread_t tid1;
    pthread_attr_t attr;
    unsigned char *buf = (unsigned char *)malloc(BUF_SIZE);
    apphdr_t *apphdr;
/ Create the thread that will periodically check to /
/* to see if data is still being received.. */

pthread_attr_init(&attr);
pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

fprintf(stderr, "Creating threads \n");
pthread_create(&tid1, &attr, keepalive, (void *)0);

/* This is the main receive loop.. One trip through for every */
/* packet received.. Exit is from the middle on receipt of */
/* a close request */

apphdr = (apphdr_t *)buf;
while (1)
{
    len = read(sock, buf, BUF_SIZE);

    /* Make sure this appears to be a valid packet and not some */
    /* traceroute! */

    if (len < sizeof(apphdr_t))
    {
        printf("Length error %d \n", len);
        continue;
    }

    if (ntohl(apphdr->magic) != UDPCS_MAGIC)
    {
        printf("bad magic number %x \n", ntohl(apphdr->magic));
        continue;
    }

    /* When a close is received compute drop% and bit rate and quit */

    if (ntohl(apphdr->ptype) == PKT_CLOSE_REQ)
    {
        printf("Close req.. recvs = %d drops = %d rate = %6.1f \n", recvs, drops, 100.0 * drops / (drops + recvs));

        tu1 = 1000000.0 * time1.tv_sec;
        tu1 += time1.tv_usec;

        tu2 = 1000000.0 * time2.tv_sec;
        tu2 += time2.tv_usec;

        printf("Bit rate was %12.0f \n",
            (double)8000000 * count / (tu2 − tu1));

        write(sock, &close_resp, sizeof(apphdr_t));
        sleep(1);
        write(sock, &close_resp, sizeof(apphdr_t));
        exit(0);
    }
}
/* Only PKT_DATA and PKT_DATA_POLL remain valid */

    if ((ntohl(apphdr->ptype) & ~1) != PKT_DATA)
    {
        printf("bad packet type %x \n", ntohl(apphdr->magic));
        continue;
    }

/* If we get here this must be a data packet */

count += len;         /* Track number of bytes consumed */
alive = 1;            /* Flag still alive               */

/* If this is the first packet remember start time */
/* else update end time.                        */

    if (recvs == 0)
        gettimeofday(&time1, &dummy);
    else
        gettimeofday(&time2, &dummy);
    recvs += 1;

/* Check for some client builder who forgets to increment */
/* sequence numbers or transmit them in network byte order */

    seq = ntohl(apphdr->seq);
    fprintf(stderr, "%8d\n", seq);
    if (seq < nextseq)
    {
        printf("bad seq number %d last was %d \n", seq, nextseq);
        continue;
    }

/* Adjust drop count */

drops += seq - nextseq;
nextseq = seq + 1;

/* If this is a poll request provide drop status */

    if (ntohl(apphdr->ptype) == PKT_DATA_POLL)
    {
        stat_resp.thisseq = htonl(seq);
        stat_resp.drops   = htonl(drops);
        write(sock, &stat_resp, sizeof(statresp_t));
    }