How do you get more performance out of the Internet? The Internet uses Transaction Control Protocol (or TCP) for most of its communications. The direct benefit is the data reliability and streaming: all communication channels appear to be a continuous flow of information like two people talking on a telephone. While greatly useful, TCP adds a lot of overhead for handshaking between the two hosts that may not be what you want in sending messages, especially in this age of peer-to-peer (P2P) transmissions. This article is the first of a two part series describing datagrams, multicasting, and improving reliability.

TCP is one of a few protocols you can find in the TCP/IP stack. Each protocol fits a specific need or function. TCP makes the endless sequence of messages (packets) appear like a continuous flow of bi-directional, highly reliable data. Another protocol, the User Datagram Protocol (UDP), lets you send and receive individual datagrams which are a sequence of packets grouped into discrete messages. Datagrams are very fast, requiring no handshaking overhead.

**The Internet Datagram (UDP)**

Datagrams are similar to passing notes in the classroom. Each note is self-contained, having the destination and source addresses and the body of the message. Just like passing notes, you cannot be sure that: first, it gets to the destination; second, its contents are unrevised during transit; and third, only the recipient reads the contents.

TCP can guarantee the integrity of the communication path, but cannot guarantee against snooping. Still, TCP is essentially continuous, the snooper would have to listen to the whole conversation. The datagram message is self-contained, so easier to decipher. Also, TCP ensures that the data that the reader gets is exactly what was sent in the right order. UDP offers no such guarantees.

Determining when to use one protocol or another depends on the data itself. Transferring files may not be a good use of UDP, because you probably can’t send the whole file all at once. The datagram is limited to less than 64KB of data. So, a program designed to transmit a file would have to break it up into smaller pieces and send each piece in succession. However, sending individual pieces may not work, because UDP does not ensure that the datagrams would arrive in the same order as they were sent or even arrive at all (often called lossy).

For all the limitations, UDP is very powerful and fast. Two data types lend themselves to datagrams very well are temporal (useful for a time period) and audio/visual samplings. Current weather conditions, the correct time, and current stock prices are all examples of temporal data: once the server takes the next sampling, previous data is less meaningful. Audio/visual samplings fit rather nicely into datagram protocols. Like temporal data,
audio and visual sampled data loses importance or significance when the next sample is taken. The network or client can drop samples without degrading too badly what the person sees and hears. As you program tools for P2P communications, you can take advantage of the brain’s capability to fill in gaps. For example, a server can send an audio stream in something like two channels. One channel is low quality (4KB/s), and the second channel is high quality (32KB/s). Each channel representing the time frame – just that the second channel would generate 8x the data. With this format, you can use the first channel as a scaling factor for the second channel, thus increasing compression. Also, if either channel’s packet gets lost in the network, the other channel would pick up the slack. UDP’s fast but lossy data flow is a natural fit in A/V streaming.

**Writing a Datagram Program**

A peer-to-peer program has two roles, a sender and a receiver. The sender transmits the message, and the receiver accepts the message. In its simplest form, the two roles are separated into two source listings. More complicated forms often use multitasking to help split the roles while maintaining one code base. This section shows you how write a simple transmitter/receiver and assumes you know C fairly well.

Like writing TCP programs, UDP uses the standard C system call `socket()`. The `socket()` system call returns a socket descriptor (an index into the various I/O channels the program is connected to) through which you can send and receive datagram messages. Unlike TCP sockets, you cannot treat the socket descriptor like a file descriptor, because file descriptor functions require streaming connections.

The sender’s initialization looks like this:

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

int sd;
struct sockaddr_in addr;
sd = socket(PF_INET, SOCK_DGRAM, 0); /* create datagram socket */
memset(&addr, 0, sizeof(addr)); /* clear dest’s addr */
addr.sin_family = AF_INET; /* indicate IP */
addr.sin_port = htons(DEST_PORT); /* set dest’s port number */
inet_aton(DEST_ADDR, &addr.sin_addr); /* set dest’s address */
```

Every network program, like this one, uses the `socket()` system call. This code fragment shows you how to create the socket and populate the destination address structure, `addr`. The example uses a library call to help with converting the byte-orders, `htons()`. This library call is from the abbreviation: “host to network short”, meaning that it converts two bytes in the right order for network use. You can find a complete list of system and library calls in the Linux Socket Programming from SAMS Publishing.

After creating the socket and getting the address ready, the next step is sending a message. POSIX-compliant operating systems include a large set of tools to work with...
Datagram/Multicasting (Part 1)

sockets. Two tools for sending and receiving messages are sendto() and recvfrom(). Expanding the fragment, above, you add a sendto() system call:

```c
/* send the data in buf (buflen bytes) to dest at addr */
sendto(sd, buf, buflen, &addr, sizeof(addr));
```

It’s that simple.

Of course, these code fragments do not include any form of error checking – which you may want to do for program integrity. Some really nasty defects come from not checking the return codes of these system calls.

**Catching the Message**

The receiver gets the message that the sender sends, and the receiver’s program looks similar to the code fragment for initializing the sender. Essentially, you change the last line (note the bolded changes) and add one more:

```c
addr.sin_addr.s_addr = INADDR_ANY; /* accept from any network */
bind(sd, &addr, sizeof(addr)); /* set/publish the local port */
```

The first change tells the bind() system call to accept messages from any source. Most operating systems allow you to assign more than one network IP address. If you have more than one network addresses, you can designate one or all of the addresses to receive messages.

The bind() call tells the OS that you want DEST_PORT for the receiver. This essentially publishes that port for other programs to send messages to. This is one instance when you must check the return value of bind(), because the program is most likely to fail here when setting up a socket. If the call returns anything but zero, you have some error.

The sender transmits the datagram to this port. The receiver can pick up the incoming packet with a special read call:

```c
char buf[MAXBUF];
int len = sizeof(addr);
/* wait for and read a datagram from the network */
int bytesread = recvfrom(sd, buf, bufsize, 0, &addr, &len);
```

When the receiver program gets to this point it gets the message from a queue. If no message is ready, it waits until one arrives. The addr variable gets the sender’s address. You definitely want to capture this information, because you may want to reply to the incoming message.

As stated above, the maximum datagram size is 64Kbytes. This can get a little tricky. Even though you that much space, if the receiver does not provide a buffer big enough for the incoming message, the rest is truncated and lost.
Datagram/Multicasting (Part 1)

You can probably see why these algorithms separate programs or multitasking: waiting in a `recvfrom()` system call make it difficult to send other messages. You could use nonblocking I/O (see the book), but creating a separate thread or task is much easier. This is especially true while connected to a multicast channel.

**Multicasting Datagrams**

Surprisingly, you don’t need to do any more to the sender to transmit a multicast message. Currently, all multicasting is UDP only (IPv6 has included support for multicast TCP). To send a multicast message, you just transmit to one of the multicast addresses (224.0.0.0 – 239.255.255.255). If you are connected to the Internet, be careful when you choose the destination address, because some address ranges are defined as “global” or WAN. Sending test messages all over the work is somewhat network unfriendly.

The receiver listening to a multicast address has to configure its socket a little more. While the program’s socket has its own address and port, to get multicast message from the network you need to use `setsockopt()` (set socket option) to subscribe to the service. This call requires the address and network interface (like the `bind()` system call). The following code sample demonstrates how to fill the data structure and to set the socket option:

```
inet_aton("224.0.0.1", &mreq.imr_multiaddr); /* select address */
mreq.imr_interface.s_addr = INADDR_ANY; /* any interface */
setsockopt(sd, SOL_IP, IP_ADD_MEMBERSHIP, &mreq, sizeof(mreq));
```

If successful, this option setting tells all the routers within the network scope to watch for and redirect messages with the specified address. Just like `bind()`, you can specify a particular local host address through which the program accepts the messages. Multicasting works through the routers. The routers are responsible for tracking subscriptions and forwarding all messages that match selected address. From the instant when the program completes the subscription the operating system receives and forwards on every message that matches the address. What can be confusing for the programmer is that the program cannot distinguish between a multicast message and a directed (P2P) message.

You need to be aware of an important note about ports. The operating system still recognizes UDP’s use of ports, so you need to select a port using `bind()` just like normal P2P messaging, described earlier in this article. From the perspective of multicast programming support, you simply add the subscription statements after calling `bind()`. This is a significant limitation to multicasting’s design: from the point that you subscribe, you get all messages for that address regardless of the port selection. The operating system filters out all port-mismatched messages. This means that you can potentially and significantly lose network performance due to flooding (overflow of unwanted/unsolicited messages).
Datagram/Multicasting (Part 1)

**Aligning Functionality**

Even with the limitations of multicast, you can tune-up your broadcast messaging like virtual conferences and file distribution. Remember, use the transport protocol (either TCP, UDP, or multicast) based on the best fit for the application. If you try to shoehorn one protocol into a ill-suited implementation, you may have to pad it with custom programming or get unexpected results.

TCP fits best with a connected, reliable communication when knowing where you are is important. The datagram protocols are best for single, or one-shot, messages that need only a one-shot reply (if at all). Lastly, multicasting increases your potential data inflow or even flooding but giving you the capability to blast a message generally to anyone listening.

This article described the different parts of datagram programming. The next article takes the discussion further by describing how to create an intermediate protocol, a reliable datagram protocol. It also demonstrates the functionality with a tool that allows very large files (like ISO CDROM images) be transmitted to several destinations without hogging the server or network. The book Linux Socket Programming from SAMS Publishing goes into much greater detail on all levels of socket programming, and you can get all source code from www.cs.utah.edu/~swalton.