Multicast UDP surprises

Posted on 2015-04-24T19:07Z. Updated on 2018-05-15T21:17Z.

In my job I have to deal with UDP multicast traffic constantly. Creating software that communicates using multicast messages in C or C++ using the BSD sockets API can lead to surprising situations and behavior due to the way the API and the kernel work (Linux in my case). I think some of those aspects are interesting and will try to detail them here. I'll suppose you are somewhat familiar with the sockets API and I'll try to escalate from TCP to "What you should know if you're going to use UDP multicast". Everything here refers to IPv4. I've detected small differences in IPv6, but I didn't have time to experiment fully with it.

TCP

Most people start socket programming with TCP as it's easy and convenient. You start by creating a socket with the socket() system call and from there proceed to do the following.

In the client, you call connect() to connect to a server at a given address. After that, the socket can be used like a file with read(), write() and close(). You can call bind() before calling connect() but in most cases it's not needed.

In the server, you first bind the socket to a listening address with bind(). At that point you normally specify a listening port and a listening IP address. The listening address can be INADDR_ANY to listen on any (all) IP addresses at that port. If you run `netstat --listen --tcp -n` on your machine and you see an asterisk before the port name, that means the process specified INADDR_ANY. You can also specify another listening address, like 127.0.0.1 or the IP address of a network interface, which you can dynamically obtain prior to calling bind(), so as to restrict traffic to a given interface and avoid listening to the world.

After that, you call listen() to put the socket in listening mode and accept() to accept new connections. The initial socket corresponds to the listening part you can see with netstat, while new sockets returned by accept() represent a specific connection with a specific client. Like before, you can call write(), read() and close().

Congratulations after writing a couple of client/server programs doing that. You're now at the beginner level of socket programming. You can move up the stair by trying select(), poll(), non-blocking reads, threading or multi-processing to handle client connections on the server, etc.

UDP

Coming from a TCP background, a few things may surprise you. In the client you usually call bind() with port 0 and INADDR_ANY to bind the socket to a given address that will be used to both send messages and receive responses. Being connection-less, this is a needed step. After that, sendto() and recvfrom() will allow you to send data to other processes and receive data from them.

In the server, you normally bind() to a specific port and use a specific IP address or INADDR_ANY as before, and you sendto() and recvfrom() as in the client.

Few things appear to have changed yet there is an important difference. In TCP, the operating system queues data for you grouping it by connection (i.e. client-server socket) and each one is essentially a queue of bytes. If you receive 20 bytes but request to read 40, your read operation will block unless you switched the socket to non-blocking mode. In the latter case, you'll read 20 immediately. Likewise, if you request to read 10 you'll read 10, and the 10 remaining bytes will wait for a future read().

In UDP, by contrast, there's only one server socket and the operating system queues your messages by order of arrival (they are not grouped by any connection because they don’t exist) and handles each message (UDP datagram) as an indivisible block. It’s a queue of messages instead of a queue of bytes. If it was a queue of bytes and you only have one socket, you couldn’t really be sure which bytes belong to which client.

So if you receive a 20-bytes datagram and request to read 10, the operating system will dequeue that message and serve you the first 10 bytes, discarding the other 10 (it can also return an error indicating the buffer is too small). If you request to read 40 instead, the operation will not block unless there are no datagrams pending and you’re in blocking mode, and in our case it will immediately return with the 20-bytes datagram.

This normally means when designing a UDP protocol you either create fixed-size datagrams or always use a large buffer and specify the maximum read size with each recvfrom() call. In my case, radars serve my software variable-sized data up to a maximum size of 64 KiB, which more or less matches the maximum size of a normal UDP datagram, so I usually declare reception buffers with that size and use it in the recvfrom() call.

Unless you’ve been warned or explained this, it usually catches you by surprise in your first UDP programs.

Multicast UDP

Sending data

If you want to create a program that will send traffic to a destination multicast address, an additional step is needed. After creating the socket with socket() and binding it to a local address with bind(), you need to call setsockopt() to specify the IP address of the network interface you want outgoing multicast traffic to go through. Take into account that, by being multicast traffic, the operating system cannot decide which interface to send the message from using the routing table. Multicast traffic can go through any given interface. See the following code:

```
int fd;
int ret;

// Create socket.
int fd = socket(AF_INET, SOCK_DGRAM, 0);
if (fd == -1)
{
        /* Handle error. */
}

// Bind to local address.
struct sockaddr_in bind_addr = {
        .sin_family = AF_INET,
        .sin_port = 0,
        .sin_addr = {
                .s_addr = INADDR_ANY
        }
};
ret = bind(fd, (const struct sockaddr *)(&bind_addr), sizeof(bind_addr));
if (ret == -1)
{
        // Handle error.
}

// Set multicast interface.
struct in_addr if_ip;
if_ip.s_addr = ...;

ret = setsockopt(fd, IPPROTO_IP, IP_MULTICAST_IF,
                 (const void *)(&if_ip), sizeof(if_ip));
if (ret == -1)
{
        // Handle error.
}
```

After that, you can use the socket normally and indicate a multicast IP address as the destination address. To send multicast traffic through several interfaces, the easiest option is to create one socket per interface.

Receiving data

The next step is creating a program that will receive multicast traffic, say to IP address 230.1.1.1 and port 4001. There are several steps involved. A program wanting to receive multicast traffic needs to, at least, add a multicast subscription to that IP address on a given interface (specified by IP address too). That subscription is added to a socket and is usually related to the bind address of the socket, but operated independently as we will see later.

In the bind() call, you must use the given port (4001 in this case). The bind address is usually set to INADDR_ANY or the interface address if you want to receive unicast traffic on that socket too, in addition to the multicast traffic. If you don't want to receive unicast traffic on the socket and restrict it to multicast traffic, you normally specify the multicast IP address as the bind address.

Edit: Christian Wills points out you can also bind the socket to INADDR_ANY and set the IP_MULTICAST_ALL option to zero. In that case, you will not receive multicast traffic for multicast groups you haven't added to the socket.

But that's not enough to receive multicast traffic on that port. The operating system needs more information. It won't let multicast traffic in through an interface unless a running process has explicitly requested to receive multicast traffic to that IP address (multicast group) through that interface. This needs, again, a call to setsockopt() to add a multicast group subscription through that interface.

```
// Suppose we have already called socket() and bind().
struct ip_mreq group;
group.imr_interface.s_addr = /* Interface IP address in network order. */;
group.imr_multiaddr.s_addr = /* 230.1.1.1 in network order. */;

ret = setsockopt(fd, IPPROTO_IP, IP_ADD_MEMBERSHIP,
                 (const void *)(&group), sizeof(group));
if (ret == -1)
{
        // Handle error.
}
```

After a running process has successfully added a multicast group to a socket, the subscription will be listed by `netstat -gn`, the command you can run to check everything is working.

As soon as you want to receive traffic from more than one multicast IP address to the same port, some questions arise. If you bind() to INADDR_ANY, you will open yourself to unicast traffic. If you bind() to the multicast address, you can only receive from that multicast address. Normally, the easy solution is to create one socket per multicast address, binding each socket to each address and adding the corresponding multicast subscription to each socket.

SO_REUSEADDR

Depending on the way you manage your sockets, you **might** have to activate the SO_REUSEADDR option for your sockets. This option is set with setsockopt(), as before, and needs to be set before calling bind().

```
int one = 1;
ret = setsockopt(fd, SOL_SOCKET, SO_REUSEADDR,
                 (const void *)(&one), sizeof(one));
if (ret == -1)
{
        // Handle error.
}
```

SO_REUSEADDR, in the context of multicast UDP, allows sockets to share the same listening address. If you create several sockets and all of them listen on a different multicast IP address, there will be no conflict. But, for example, if you want to receive unicast traffic to port 4001 by binding to INADDR_ANY and have a **separate** socket for multicast traffic, binding to *:4001 and 230.1.1.1:4001 at the same time will only be possible by specifying SO_REUSEADDR on both sockets. Either no reuse is allowed and there's only one listening socket or reuse is allowed by everyone and there may be several listening sockets. That's one example conflict but your application may have others that force you to use SO_REUSEADDR.

Relationship between the multicast address, the port and the interface address

Here comes the surprise. One of the most common mistakes is to think that all three elements (multicast address, port and interface address) are a single entity that acts as a filter to the traffic you will receive on a given socket. After all, the three of them are specified related to the socket. The multicast address and port are used when binding, and the multicast address and interface address are used when subscribing to the multicast group.

So if you bound to 230.1.1.1:4001 and subscribed to 230.1.1.1/192.168.1.1 on a socket, many people would expect that socket to receive traffic through the interface with address 192.168.1.1 going to address 230.1.1.1 and port 4001, and **only** that traffic. But it doesn't work like that.

When you bind, you specify your listening address. You cannot received traffic that goes to incompatible addresses. When you subscribe to a multicast group your are saying two things to the operating system. On the one hand, traffic to that multicast group should be allowed through that interface because there's a process interested in receiving it (we're not taking into account firewalling rules, of course). On the other hand, that allowance will last as long as the socket is open.

So when the operating system sees multicast traffic coming through that interface, it will first check if someone has requested traffic to that multicast group through that interface. If **someone** has, it will let those datagrams in. Immediately and surprisingly, the operating system will *forget* (or stop caring about) which interface the traffic came through. It will deliver it to every socket bound to a compatible destination address and port, including any socket bound to INADDR_ANY on that port unless the IP_MULTICAST_ALL option has been set to zero for that socket.

So if you want to receive traffic to a given multicast group on several interfaces and you create one socket per interface, binding with reuse and adding the multicast group on one interface per socket, you will be surprised when you receive **traffic from every interface replicated on all those sockets**. And it doesn't stop per-process. One program of yours listens for multicast traffic and enables the multicast group on a given interface. Another process, outside your control, subscribes to the same multicast group on a completely different interface and suddenly **both processes will start receiving all traffic**.

If you're thinking now about the **security implications** of the previous paragraphs you're on the good track. You cannot isolate a process receiving multicast traffic to a specific interface from the source code alone. A third-party outside your control can enable that multicast group on another interface and it will reach your process. I've experimented this first hand. Comments are welcome in this regard.

If you want to test all of this by yourself, I have created a repository in GitHub containing a couple of example test programs for sending and receiving multicast traffic. All my tests ran on real hardware with real network interfaces. I haven't bothered to test if everything works the same way in virtual machines.

Many thanks to W. Richard Stevens for extracting the multicast API documentation from Steve Deering's original README published in 1989 and to Gary R. Wright for restoring Stevens' kohala.com site after him passing away.

Load 5 comments