

# CSIS0234B Computer and Communication Networks (Class B)

## Reading for Tutorial 10

### Multicast programming

So far, all network programs that we write only perform unicasting. This week we will try to write a program that perform multicasting, i.e., to send packets to a group of hosts which have shown interests in it.

#### 1. Background knowledge

Before we see how to write a program to send and listen to multicast addresses, let's quickly look at how multicast works in principle.

##### 1.1. Multicast abstraction and address

An IP address between 224.0.0.0 and 239.255.255.255 (i.e., those start with the bits 1110) is said to be an IP multicast address. No host owns such addresses. Instead, the address is used for performing multicasting. Each *multicast group* is represented by one of these addresses. A computer can multicast to all computers of a group by sending a datagram to one of these addresses. Those hosts in that group (i.e., those that listen to the address of the group) will get the datagram, and those that doesn't listen to such address will not. In this way, those hosts uninterested in a particular group do not need to spend CPU cycles to process frames that are sent to that group.

Typically, the network interface hardware will screen out most of the frames, so as to reduce the CPU load of the hosts. For example, for Ethernet, those hardware addresses with the 7th bit set (e.g., 01:00:00:00:00:00) are multicast addresses. When a program indicates that it wants to listen to a particular address, the OS kernel will configure the network card to listen to the corresponding multicast addresses. However, there is a card-specific limit on the number of multicast addresses that can be configured this way. If the number is exceeded, the host will configure the network card to receive all multicast packets, and selects them in the kernel. The Internet uses Ethernet multicast addresses in the range 01:00:5e:00:00:00 to 01:00:5e:7f:ff:ff for IP multicasting. This means the 28 unspecified bits in the IP multicast address have to map to the 23 bits of the Ethernet multicast address. In practice, the lower-order 23 bits are mapped. E.g., An Ethernet interface configured to listen to 224.0.0.1 will also listen to 231.128.0.1 (which are screened out by the kernel), but not 224.0.0.2.

Some multicast addresses are allocated for particular purposes. A list of such groups can be found in the "Assigned Numbers" RFC (RFC 1700). For example,

- 224.0.0.1 is the *all-hosts* group. If you ping that group, all multicast-capable hosts on the network should answer, as every multicast-capable host joins that group when they start their multicast-capable network interfaces.
- 224.0.0.2 is the *all-routers* group. All multicast routers join that group on all their multicast-capable interfaces.
- 224.0.0.4 is the *all DVMRP routers* group. A DVMRP router is a specific kind of router that route multicast packets across networks.
- 224.0.0.5 is the *all OSPF routers* group. We already see in the last tutorial that all OSPF traffic including HELLO, LS update, LS acknowledgement, etc., are sent to the group.

## 1.2. Scope of multicast

When we say that “those routers that listen to that multicast address will receive the datagram”, we don’t mean that all computers in the planet has such privilege. After all, if every OSPF routers in the world listen to the datagrams sent by every OSPF routers, then the network should saturate completely and cannot deliver anything! The scope of a packet is controlled in two ways. Firstly, some multicast addresses has specific range. In particular:

- 224.0.0.0 through 224.0.0.255 are *link local*, meaning that they will never be forwarded by any IP-level router. Therefore, if you have several Ethernet network combined using a IP router, broadcast in such addresses by one of network will never reach another.
- 239.0.0.0 through 239.255.255.255 has *administrative* scope, meaning that they will never be forwarded by any IP-level router **across** organization. The system administrators would decide whether his routers are connecting two networks of the same or different organizations, and thus configure the routers to forward or not to forward multicast. (Most local multicast group should use one of these addresses.)

For other addresses, the **sender** is responsible for determining how far should the packet go. This is done by setting the TTL field of the IP header, using an IP option as we will soon discuss.

- $TTL < 32$ : *Site local*. The datagram is restricted to the same organization, like the addresses in 239.0.0.0 through 239.255.255.255. As usual, everytime the datagram is forwarded, TTL is reduced by 1. So smaller number means less computers can receive the packet. In particular, if  $TTL = 0$ , it will never be sent over a network interface, and the multicast is said to be *Node local*. If  $TTL = 1$ , the datagram is restricted to the same subnet, and won’t be forwarded by an IP router. So such datagrams are link local.
- $32 \leq TTL < 64$ ,  $64 \leq TTL < 128$ : *Regional local, continental local*. The datagram is restricted to the same region or continent. Seldom useful.
- $TTL \geq 64$ : *Global* and unrestricted in scope.

Such use of TTL is troublesome for routers (it makes it impossible to deal with routing loops). If possible, one should use the link-local and administrative scope addresses mentioned above.

## 1.3. UDP and TCP with multicast

Recall that when we perform unicast, each host has only one address, and must be shared by multiple processes. So a port number is attached to each datagram and form a UDP packet. Different processes can thus listen to different UDP ports. It is similar for multicasting: each process can create a socket and bind it to a particular port of some multicast addresses (on some of the interfaces). The OS will then deliver to the process those packets of those multicast addresses sent to that port. The normal rule that only one process can listen to each port applies: two processes cannot listen to the same UDP port, even if they want to listen from two different multicast addresses (or one to a multicast address while the other just want unicast).

In unicast, unreliable IP datagrams can be made reliable by using a connection between two hosts, using TCP. This cannot be done in multicast, since there is not a connection per-se. In particular, the sending host won’t know how many hosts, or even whether there is any host, listening to its multicast, and therefore cannot maintain a connection. In other words, multicasts are intrinsically unreliable.

## 2. Multicast programming

Recall that, if you want to write a program that sends UDP packets, the following steps should be taken:

1. With the `socket()` system call, create a socket, with the `PF_INET` domain and `SOCK_DGRAM` type.
2. If a program requires a fixed port number (typically needed by a server), one should create an address structure and `bind()` the port number to the socket. Otherwise the kernel selects an unused port number when the first datagram is sent using the socket.
3. Now the program can send and receive datagrams using `sendto()` and `recvfrom()`.
4. One can use `connect()` to avoid having to specify the address of the other side every time a datagram is sent or received.

### 2.1. Sending and receiving multicasts

To send and receive multicast packets is similar. Obviously, when sending a packet, one has to specify a multicast address as the recipient address. Indeed, if you use the default option, this is the only thing needed to send a multicast datagram. On the other hand, socket options can be set to specify the TTL of datagrams to send and what interface to send it to.

On the other hand, receiving a datagram is slightly more complicated. You have to configure the socket before the OS knows that you want to receive from a multicast address. In the terminology of IP multicast, you need to **join** a multicast group before you can receive multicast packets, and you may choose to **leave** a multicast group if at the middle you decide not to listen to messages of the group. For some boring reasons, this is not done by `bind()`. Instead, it is again done by some socket option.

As usual, socket options are set using `setsockopt()`. Recall that the prototype is

```
int getsockopt(int s, int level, int optname,  
              void* optval, socklen_t *optlen);  
int setsockopt(int s, int level, int optname,  
              const void* optval, socklen_t optlen);
```

For all of the options for multicast, `level` should be set to `SOL_IP` (IP level configuration). The `optname` and the corresponding data type of `optval` is as follows:

Option	Datatype	Description
<code>IP_ADD_MEMBERSHIP</code>	<code>struct ip_mreqn</code>	Join a multicast group
<code>IP_DROP_MEMBERSHIP</code>	<code>struct ip_mreqn</code>	Leave a multicast group
<code>IP_MULTICAST_IF</code>	<code>struct ip_mreqn</code>	Interface to send multicasts
<code>IP_MULTICAST_TTL</code>	<code>u_char</code>	TTL for outgoing multicasts
<code>IP_MULTICAST_LOOP</code>	<code>u_char</code>	Whether to send (loopback) outgoing messages to oneself

## 2.2. Joining and leaving groups

Both `IP_ADD_MEMBERSHIP` and `IP_DROP_MEMBERSHIP` are “write-only” options, i.e., `getsockopt()` cannot be called on them. Both takes a `struct ip_mreqn` (“multicast request—new style”) type argument, defined in `<netinet/in.h>` as follows.

```
struct ip_mreqn {           /* older implementation uses ip_mreq */
    struct in_addr imr_multiaddr; /* IP multicast address of group */
    struct in_addr imr_address;   /* local IP address of interface */
    int imr_ifindex;            /* Interface index */
};
```

Here `imr_multiaddr` specifies the multicast group to join. `imr_address` (or, in older implementations, `imr_interface`) specifies the local address of the interface that you want to listen to, and `imr_ifindex` should store the corresponding index. For simple programs one can specify `INADDR_ANY` and `0` respectively, so that the system choose a multicast-capable interfaces for you.

The same socket can join multiple multicast group, up to a maximum of `IP_MAX_MEMBERSHIPS` (which is currently 20).

After joining a group, the OS configures the network interface to listen to the specified multicast addresses. The OS keeps a list of such addresses, in Linux it can be examined by reading the file `/proc/net/igmp` (although it is listed as a plain 32-bit hexadecimal number in host order, so it is not exactly easy to read). An entry will be removed, and hence the network interface can stop picking up datagrams to a joined multicast address, only when all sockets joining that multicast address are either closed or leave the group.

## 3. How the system chooses an interface?

We say that if `imr_address` and `imr_ifindex` are not specified, then the interface that is listened is chosen by the OS. In Linux, by default this is chosen to be the first interface which support broadcasting, typically `eth0`. This default can be changed by adding an entry in the routing table. E.g., the following will make sure every multicast packet will go through the `lo` interface instead:

```
route add -net 224.0.0.0 netmask 240.0.0.0 dev lo
```

This interface is also used as the default interface for sending multicasts.

A program can send to and receive from a different interface. For sending, one can use the socket option `IP_MULTICAST_IF` mentioned in the last section. For receiving, one can specify it when joining groups using `IP_ADD_MEMBERSHIP`. In either case, the program needs to use the `ioctl` system call, to make the requests `SIOCGIFADDR` (get interface’s address), `SIOCGIFCONF` (list all interfaces) and `SIOCGIFFLAGS` (get interface flags to see whether multicast is supported). They are advanced usage, and we will not cover them in this course.