

CSIS0234B Computer and Communication Networks (Class B)

Reading for Tutorial 10

Multicast programming

Multicasting can reduce bandwidth consumption when sending messages to many computers, without causing too much load to other computers in the network as in broadcasting. We will investigate the programming interface for multicasting in this tutorial, and multicast routing during the next.

1. Background knowledge

1.1. Multicast abstraction and addresses

Class D IP addresses, i.e., those addresses starting with the bit pattern 1110 (between 224.0.0.0 and 239.255.255.255), are used for IP multicasts. Each class D address represents a **multicast group**. A computer multicasts to the group by sending a datagram with the destination IP address set to that group. Only those hosts which have **joined** the group will get the datagram. So hosts uninterested in a particular group do not need to spend CPU cycles to process such packets.

There are two complementary schemes that are needed to prevent the packets to reach irrelevant hosts. First, multicast packets must be routed in a specialized way, which we will examine in the next tutorial. But on a **broadcast network** like Ethernet, even when there is no router at all the frame generated corresponding to the IP packet reaches all computers in the network. To reduce the CPU load, we need a method that the network cards can select the right frames to forward to the OS.

To allow this, the datalink layer also has multicast addresses, similar to hardware addresses of the network cards. They correspond to multicast IP addresses. E.g., in Ethernet cards, all MAC addresses with 7th bit set (like 01:00:00:00:00:00) are multicast addresses. Unlike hardware addresses of the network cards, they are not hard-coded into the network card. Instead, the OS can ask the network card to forward frames of some specific multicast addresses and filter off the rest. Each card uses its own scheme to perform the filtering, but of course it cannot have a table of 2^{47} entries to record which are wanted and which are not. So the filtering is not perfect. Upon receiving a frame, the OS will examine the IP address to perform further filtering.

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 designated routers send routing information to routers of the network.

1.2. Transport layer protocols with multicast

Recall that the unicast address of a host must be shared by multiple processes. So a port number is attached to each datagram and form an IP packet. A process can **create a socket** and **bind** it to a particular port, so that packets it generates will carry that port in the source address, and arriving packets with that port in the destination address will be delivered to the process.

Sending multicast UDP datagrams is no different from sending unicast UDP datagrams. All needed is a multicast address.

As compared to receiving unicasts, receiving multicasts needs one additional step: the process must **configure the socket to join a multicast group**. It is also possible (and usually desirable) to **bind the multicast address** to the socket (apart from just the UDP port number). This makes sure that the process **only listens to that multicast address**, and as a result **multiple processes** can listen to the same port as long as **different multicast addresses** are binded.

On the other hand, TCP cannot be used for multicast. TCP is a connection oriented protocol, but in multicast we don't have connections. Indeed, the sending host won't know even whether there is any receiving host. So multicasts are intrinsically unreliable.

2. Receiving multicasts in our programs

2.1. Overall picture

Let's begin by looking at the steps needed for multicast communication in UDP. We will only examine the simpler case where we bind the multicast address to the socket.¹

1. As in unicast UDP servers, we use `getaddrinfo()` to get an `addrinfo` structure containing information needed for the needed system calls. The `getaddrinfo()` call should use the multicast group as the `hostname` argument, the desired port number as the `service` argument, and `set hints` to use a `SOCK_DGRAM` socket type. Note that, since we will bind to the multicast address, we will not specify `AI_PASSIVE` in the flags.
2. As in unicast UDP servers, we use `socket()` to create a socket with the domain and socket type obtained from the `addrinfo` structure.
3. As in unicast UDP servers, we use `bind()` to associate the created socket to a particular multicast address and port number.
4. We need an additional step here, to set additional options of the sockets to join the multicast group, using `setsockopt()`.
5. Now the program can use the `recvfrom()` system call to receive multicast datagrams. `sendto()` can still be used to send datagrams to particular destinations (usually, multicast addresses).

The procedure is mostly the same as a UDP server, except that we need to join the multicast group in step 4. We will look at them in the next subsection.

¹If we do not bind to the multicast address, we will have to call `getaddrinfo()` twice, first to get the multicast address for joining the group later, and second to get an empty address for use with the `bind()` system call.

2.2. Joining multicast groups

We need to use `setsockopt ()` to join a multicast group. It has the following prototype:

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

Unluckily, the interface for IPv4 and IPv6 are not unified. For multicasting on IPv4, `level = SOL_IP` (IP level configuration), `optname = IP_ADD_MEMBERSHIP`. The `optval` should be a pointer to a structure of type `struct ip_mreqn`, and `optlen` is its size. For multicasting on IPv6, `level = SOL_IPV6` (IPv6 level configuration), `optname = IPV6_ADD_MEMBERSHIP`, and the option is of type `struct ipv6_mreq`. The structures are 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 */
};
struct ipv6_mreqn {
    struct in_addr ipv6mr_multiaddr;            /* IP multicast address of group */
    unsigned int  ipv6mr_ifindex;              /* Interface index */
};
```

We are only interested in the `imr_multiaddr` and `ipv6mr_multiaddr` fields. Those fields that we are not interested in can be set to all 0s, achievable using a simple `memset ()` call.

Suppose we already have a multicast address stored in a `sockaddr` structure, presumably obtained from a call to `getaddrinfo ()`. We can now check whether the address is an IPv4 or IPv6 address, by examining its `sin_family` field and see whether it is `AF_INET` or `AF_INET6`. After that, we can cast the address structure into the right address type, assign it to a multicast request structure, and make the needed call to `setsockopt ()`. The code looks like the following:

```
int multicast_join(int sockfd, struct sockaddr *mcast_addr) {
    if (mcast_addr->sa_family == AF_INET) {
        struct ip_mreqn mreq;
        memset(&mreq, 0, sizeof(mreq));
        mreq.imr_multiaddr
            = ((struct sockaddr_in *)mcast_addr)->sin_addr;
        return setsockopt(sockfd, SOL_IP, IP_ADD_MEMBERSHIP,
            &mreq, sizeof(mreq));
    } else if (mcast_addr->sa_family == AF_INET6) {
        struct ipv6_mreq mreq;
        memset(&mreq, 0, sizeof(mreq));
        mreq.ipv6mr_multiaddr
            = ((struct sockaddr_in6 *)mcast_addr)->sin6_addr;
        return setsockopt(sockfd, SOL_IPV6, IPV6_ADD_MEMBERSHIP,
            &mreq, sizeof(mreq));
    } else {
        errno = EINVAL;
        return -1;
    }
}
```

The same socket can be used to join multiple multicast group, up to a maximum of 20 (IP_MAX_MEMBERSHIPS).

After joining a group, the OS configures the network interface to listen to the specified multi-cast 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, reversed from the network order which we normally see, so it is not exactly easy to read). Once all sockets joining that multicast address are either closed or left that group (using the `IP_DROP_MEMBERSHIP` socket option similar to `IP_ADD_MEMBERSHIP`), the entry will be removed.