21.1 Introduction

As shown in Figure 20.1, a unicast address identifies a *single* IP interface, a broadcast address identifies *all* IP interfaces on the subnet, and a multicast address identifies a *set* of IP interfaces. Unicasting and broadcasting are the extremes of the addressing spectrum (one or all) and the intent of multicasting is to allow addressing something in between. A multicast datagram should be received only by those interfaces interested in the datagram, that is, by the interfaces on the hosts running applications wishing to participate in the multicast group. Also, broadcasting is normally limited to a LAN, whereas multicasting can be used on a LAN or across a WAN. Indeed, applications multicast across a subset of the Internet on a daily basis.

The additions to the sockets API to support multicasting are simple; they comprise nine socket options: three that affect the sending of UDP datagrams to a multicast address and six that affect the host's reception of multicast datagrams.

[Team LiB] PREVIOUS NEXT

21.2 Multicast Addresses

When describing multicast addresses, we must distinguish between IPv4 and IPv6.

IPv4 Class D Addresses

Class D addresses, in the range 224.0.0.0 through 239.255.255.255, are the multicast addresses in IPv4 (Figure A.3). The low-order 28 bits of a class D address form the multicast *group ID* and the 32-bit address is called the *group address*.

Figure 21.1 shows how IP multicast addresses are mapped into Ethernet multicast addresses. This mapping for IPv4 multicast addresses is described in RFC 1112 [Deering 1989] for Ethernets, in RFC 1390 [Katz 1993] for FDDI networks, and in RFC 1469 [Pusateri 1993] for token-ring networks. We also show the mapping for IPv6 multicast addresses to allow easy comparison of the resulting Ethernet addresses.

Figure 21.1. Mapping of IPv4 and IPv6 multicast address to Ethernet addresses.

graphics/21fig01.gif

Considering just the IPv4 mapping, the high-order 24 bits of the Ethernet address are always `01:00:5e`. The next bit is always 0, and the low-order 23 bits are copied from the low-order 23 bits of the multicast group address. The high-order 5 bits of the group address are ignored in the mapping. This means that 32 multicast addresses map to a single Ethernet address: The mapping is not one-to-one.

The low-order 2 bits of the first byte of the Ethernet address identify the address as a universally administered group address. *Universally administered* means the high-order 24 bits have been assigned by the IEEE and group addresses are recognized and handled specially by receiving interfaces.

There are a few special IPv4 multicast addresses:

- 224.0.0.1 is the *all-hosts* group. All multicast-capable nodes (hosts, routers, printers, etc.) on a subnet must join this group on all multicast-capable interfaces. (We will talk about what it means to join a multicast group shortly.)
- 224.0.0.2 is the *all-routers* group. All multicast-capable routers on a subnet must join this group on all multicast-capable interfaces.

The range 224.0.0.0 through 224.0.0.255 (which we can also write as 224.0.0.0/24) is called *link local*. These addresses are reserved for low-level topology discovery or maintenance protocols, and datagrams destined to any of these addresses are never forwarded by a multicast router. We will say more about the scope of various IPv4 multicast addresses after looking at IPv6 multicast addresses.

IPv6 Multicast Addresses

The high-order byte of an IPv6 multicast address has the value `ff`. Figure 21.1 shows the mapping from a 16-byte IPv6 multicast address into a 6-byte Ethernet address. The low-order 32 bits of the group address are copied into the low-order 32 bits of the Ethernet address. The high-order 2 bytes of the Ethernet address are `33:33`. This mapping for Ethernets is specified in RFC 2464 [Crawford 1998a], the same mapping for FDDI is in RFC 2467 [Crawford 1998b], and the token-ring mapping is in RFC 2470 [Crawford, Narten, and Thomas 1998].

The low-order two bits of the first byte of the Ethernet address specify the address as a locally administered group address. *Locally administered* means there is no guarantee that the address is unique to IPv6. There could be other protocol suites besides IPv6 sharing the network and using the same high-order two bytes of the Ethernet address. As we mentioned earlier, group addresses are recognized and handled specially by receiving interfaces.

Two formats are defined for IPv6 multicast addresses, as shown in Figure 21.2. When the *p* flag is 0, the *T* flag differentiates between a *well-known* multicast group (a value of 0) and a *transient* multicast group (a value of 1). A *P* value of 1 designates a multicast address that is assigned based on a unicast prefix (defined in RFC 3306 [Haberman and Thaler 2002]). If the *P* flag is 1, the *T* flag also must be 1 (i.e., unicast-based multicast addresses are always transient), and the *plen* and *prefix* fields are set to the prefix length and value of the unicast prefix, respectively. The upper two bits of this field are reserved. IPv6 multicast addresses also have a 4-bit *scope* field that we will discuss shortly. RFC 3307 [Haberman 2002] describes the allocation mechanism for the low-order 32 bits of an IPv6 group address (the *group ID*), independent of the setting of the *P* flag.

Figure 21.2. Format of IPv6 multicast addresses

graphics/21fig02.gif

There are a few special IPv6 multicast addresses:

- `ff01::1` and `ff02::1` are the *all-nodes* groups at interface-local and link-local scope. All nodes (hosts, routers, printers, etc.) on a subnet must join these groups on all multicast-capable interfaces. This is similar to the IPv4 224.0.0.1 multicast address. However, since multicast is an integral part of IPv6, unlike IPv4, this is not optional.

  Although the IPv4 group is called the *all-hosts* group and the IPv6 group is called the *all-nodes* group, they serve the same purpose. The group was renamed in IPv6 to make it clear that it is intended to address routers, printers, and any other IP devices on the subnet as well as hosts.

- `ff01::2`, `ff02::2` and `ff05::2` are the *all-routers* groups at interface-local, link-local, and site-local scopes. All routers on a subnet must join these groups on all multicast-capable interfaces. This is similar to the IPv4 224.0.0.2 multicast address.

Scope of Multicast Addresses

IPv6 multicast addresses have an explicit 4-bit *scope* field that specifies how "far" the multicast packet will travel. IPv6 packets also have a *hop limit* field that limits the number of times the packet is forwarded by a router. The following values have been assigned to the scope field:

1:	interface-local
2:	link-local
4:	admin-local
5:	site-local
8:	organization-local
14:	global

The remaining values are unassigned or reserved. An interface-local datagram must not be output by an interface and a link-local datagram must never be forwarded by a router. What defines an admin region, a site, or an organization is up to the administrators of the multicast routers at that site or organization. IPv6 multicast addresses that differ only in scope represent different groups.

IPv4 does not have a separate scope field for multicast packets. Historically, the IPv4 TTL field in the IP header has doubled as a multicast scope field: A TTL of 0 means interface-local; 1 means link-local; up through 32 means site-local; up through 64 means region-local; up through 128 means continent-local (meaning avoiding low-rate or highly congested links, intercontinental or not); and up through 255 are unrestricted in scope (global). This double usage of the TTL field has led to difficulties, as detailed in RFC 2365 [Meyer 1998].

Although use of the IPv4 TTL field for scoping is accepted and recommended practice, administrative scoping is preferred when possible. This defines the range 239.0.0.0 through 239.255.255.255 as the *administratively scoped IPv4 multicast space* (RFC 2365 [Meyer 1998]). This is the high end of the multicast address space. Addresses in this range are assigned locally by an organization, but are not guaranteed to be unique across organizational boundaries. An organization must configure its boundary routers (multicast routers at the boundary of the organization) not to forward multicast packets destined to any of these addresses.

Administratively scoped IPv4 multicast addresses are divided into local scope and organization-local scope, the former being similar (but not semantically equivalent) to IPv6 site-local scope. We summarize the different scoping rules in Figure 21.3.

Figure 21.3. Scope of IPv4 and IPv6 multicast addresses.

graphics/21fig03.gif

Multicast Sessions

Especially in the case of streaming multimedia, the combination of an IP multicast address (either IPv4 or IPv6) and a transport-layer port (typically UDP) is referred to as a *session*. For example, an audio/video teleconference may comprise two sessions; one for audio and one for video. These sessions almost always use different ports and sometimes also use different groups for flexibility in choice when receiving. For example, one client may choose to receive only the audio session, and one client may choose to receive both the audio and the video session. If the sessions used the same group address, this choice would not be possible.

[Team LiB] PREVIOUS NEXT

21.3 Multicasting versus Broadcasting on a LAN

We now return to the examples in Figures 20.3 and 20.4 to show what happens in the case of multicasting. We use IPv4 for the example shown in Figure 21.4, but the steps are similar for IPv6.

Figure 21.4. Multicast example of a UDP datagram.

graphics/21fig04.gif

The receiving application on the rightmost host starts and creates a UDP socket, binds port 123 to the socket, and then joins the multicast group 224.0.1.1. We will see shortly that this "join" operation is done by calling `setsockopt`. When this happens, the IPv4 layer saves the information internally and then tells the appropriate datalink to receive Ethernet frames destined to `01:00:5e:00:01:01` (Section 12.11 of TCPv2). This is the Ethernet address corresponding to the multicast address that the application has just joined using the mapping we showed in Figure 21.1.

The next step is for the sending application on the leftmost host to create a UDP socket and send a datagram to 224.0.1.1, port 123. Nothing special is required to send a multicast datagram: The application does not have to join the multicast group. The sending host converts the IP address into the corresponding Ethernet destination address and the frame is sent. Notice that the frame contains both the destination Ethernet address (which is examined by the interfaces) and the destination IP address (which is examined by the IP layers).

We assume that the host in the middle is not IPv4 multicast-capable (since support for IPv4 multicasting is optional). This host ignores the frame completely because: (i) the destination Ethernet address does not match the address of the interface; (ii) the destination Ethernet address is not the Ethernet broadcast address; and (iii) the interface has not been told to receive any group addresses (those with the low-order bit of the high-order byte set to 1, as in Figure 21.1).

The frame is received by the datalink on the right based on what we call *imperfect filtering*, which is done by the interface using the Ethernet destination address. We say this is imperfect because it is normally the case that when the interface is told to receive frames destined to one specific Ethernet multicast address, it can receive frames destined to other Ethernet multicast addresses, too.

> When told to receive frames destined to a specific Ethernet multicast address, many current Ethernet interface cards apply a hash function to the address, calculating a value between 0 and 511. One of 512 bits in an array is then turned ON. When a frame passes by on the cable destined for a multicast address, the same hash function is applied by the interface to the destination address (which is the first field in the frame), calculating a value between 0 and 511. If the corresponding bit in the array is ON, the frame is received; otherwise, it is ignored. Older interface cards reduce the size of the bit array from 512 to 64, increasing the probability that an interface will receive frames in which it is not interested. Over time, as more and more applications use multicasting, this size will probably increase even more. Some interface cards today already have perfect filtering (the ability to filter out datagrams addressed to all but the desired multicast addresses). Other interface cards have no multicast filtering at all, and when told to receive a specific multicast address, must receive all multicast frames (sometimes called *multicast promiscuous* mode). One popular interface card does perfect filtering for 16 multicast addresses as well as having a 512-bit hash table. Another does perfect filtering for 80 multicast addresses, but then has to enter multicast promiscuous mode. Even if the interface performs perfect filtering, perfect software filtering at the IP layer is still required because the mapping from the IP multicast address to the hardware address is not one-to-one.

Assuming that the datalink on the right receives the frame, since the Ethernet frame type is IPv4, the packet is passed to the IP layer. Since the received packet was destined to a multicast IP address, the IP layer compares this address against all the multicast addresses that applications on this host have joined. We call this *perfect filtering* since it is based on the entire 32-bit class D address in the IPv4 header. In this example, the packet is accepted by the IP layer and passed to the UDP layer, which in turn passes the datagram to the socket that is bound to port 123.

There are three scenarios that we do not show in Figure 21.4:

1. A host running an application that has joined the multicast address 225.0.1.1. Since the upper five bits of the group address are ignored in the mapping to the Ethernet address, this host's interface will also be receiving frames with a destination Ethernet address of `01:00:5e:00:01:01`. In this case, the packet will be discarded by the perfect filtering in the IP layer.
2. A host running an application that has joined some multicast group whose corresponding Ethernet address just happens to be one that the interface receives when it is programmed to receive `01:00:5e:00:01:01`. (i.e., the interface card performs imperfect filtering). This frame will be discarded either by the datalink layer or by the IP layer.
3. A packet destined to the same group, 224.0.1.1, but a different port, say 4000. The rightmost host in Figure 21.4 still receives the packet, which is accepted by the IP layer, but assuming a socket does not exist that has bound port 4000, the packet will be discarded by the UDP layer.

> This demonstrates that for a process to receive a multicast datagram, the process must join the group and bind the port.

[Team LiB] PREVIOUS NEXT

[Team LiB] PREVIOUS NEXT

21.5 Source-Specific Multicast

Multicasting on a WAN has been difficult to deploy for several reasons. The biggest problem is that the *MRP*, described in Section 21.4, needs to get the data from all the senders, which may be located anywhere in the network, to all the receivers, which may similarly be located anywhere. Another large problem is multicast address allocation: There are not enough IPv4 multicast addresses to statically assign them to everyone who wants one, as is done with unicast addresses. To send wide-area multicast and not conflict with other multicast senders, you need a unique address, but there is not yet a global multicast address allocation mechanism.

Source-specific multicast, or SSM [Holbrook and Cheriton 1999], provides a pragmatic solution to these problems. It combines the group address with a system's source address, which solves the problems as follows:

- The receivers supply the sender's source address to the routers as part of joining the group. This removes the rendezvous problem from the network, as the network now knows exactly where the sender is. However, it retains the scaling properties of not requiring the sender to know who all the receivers are. This simplifies multicast routing protocols immensely.
- It redefines the identifier from simply being a multicast group address to being a combination of a unicast source and multicast destination (which SSM now calls a *channel*. This means that the source may pick any multicast address since it becomes the (source, destination) combination that must be unique, and the source already makes it unique. An SSM session is the combination of source, destination, and port.

SSM also provides a certain amount of anti-spoofing, that is, it is harder for source 2 to transmit on source 1's channel since source 1's channel includes source 1's source address. Spoofing is still possible, of course, but is much harder.

[Team LiB] PREVIOUS NEXT

[Team LiB]

21.11 Simple Network Time Protocol (SNTP)

NTP is a sophisticated protocol for synchronizing clocks across a WAN or a LAN, and can often achieve millisecond accuracy. RFC 1305 [Mills 1992] describes the protocol in detail and RFC 2030 [Mills 1996] describes SNTP, a simplified but protocol-compatible version intended for hosts that do not need the complexity of a complete NTP implementation. It is common for a few hosts on a LAN to synchronize their clocks across the Internet to other NTP hosts and then redistribute this time on the LAN using either broadcasting or multicasting.

In this section, we will develop an SNTP client that listens for NTP broadcasts or multicasts on all attached networks and then prints the time difference between the NTP packet and the host's current time-of-day. We do not try to adjust the time-of-day, as that takes superuser privileges.

The file `ntp.h`, shown in Figure 21.20, contains some basic definitions of the NTP packet format.

Figure 21.20 `ntp.h` header: NTP packet format and definitions.

ssntp/ntp.h

```
 1 #define JAN_1970    2208988800UL     /* 1970 - 1900 in seconds */

 2 struct l_fixedpt {               /* 64-bit fixed-point */
 3     uint32_t int_part;
 4     uint32_t fraction;
 5 };

 6 struct s_fixedpt {               /* 32-bit fixed-point */
 7     uint16_t int_part;
 8     uint16_t fraction;
 9 };

10 struct ntpdata {                 /* NTP header */
11     u_char  status;
12     u_char  stratum;
13     u_char  ppoll;
14     int     precision:8;
15     struct s_fixedpt distance;
16     struct s_fixedpt dispersion;
17     uint32_t refid;
18     struct l_fixedpt reftime;
19     struct l_fixedpt org;
20     struct l_fixedpt rec;
21     struct l_fixedpt xmt;
22 };

23 #define VERSION_MASK    0x38
24 #define MODE_MASK       0x07

25 #define MODE_CLIENT     3
26 #define MODE_SERVER     4
27 #define MODE_BROADCAST  5
```

2–22 `l_fixedpt` defines the 64-bit fixed-point values used by NTP for timestamps and `s_fixedpt` defines the 32-bit fixed-point values that are also used by NTP. The `ntpdata` structure is the 48-byte NTP packet format.

Figure 21.21 shows the `main` function.

Get multicast IP address

12–14 When the program is executed, the user must specify the multicast address to join as the command-line argument. With IPv4, this would be 224.0.1.1 or the name `ntp.mcast.net`. With IPv6, this would be `ff05::101` for the site-local scope NTP. Our `udp_client` function allocates space for a socket address structure of the correct type (either IPv4 or IPv6) and stores the multicast address and port in that structure. If this program is run on a host that does not support multicasting, any IP address can be specified, as only the address family and port are used from this structure. Note that our `udp_client` function does not `bind` the address to the socket; it just creates the socket and fills in the socket address structure.

Figure 21.21 `main` function.

ssntp/main.c

```
 1 #include    "sntp.h"
 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     sockfd;
 6     char    buf[MAXLINE];
 7     ssize_t n;
 8     socklen_t salen, len;
 9     struct ifi_info *ifi;
10     struct sockaddr *mcastsa, *wild, *from;
11     struct timeval now;

12     if (argc != 2)
13         err_quit("usage: ssntp <IPaddress>");

14     sockfd = Udp_client(argv[1], "ntp", (void **) &mcastsa, &salen);

15     wild = Malloc(salen);
16     memcpy(wild, mcastsa, salen);   /* copy family and port */
17     sock_set_wild(wild, salen);
18     Bind(sockfd, wild, salen);  /* bind wildcard */

19 #ifdef  MCAST
20         /* obtain interface list and process each one */
21     for (ifi = Get_ifi_info(mcastsa->sa_family, 1); ifi != NULL;
22          ifi = ifi->ifi_next) {
23         if (ifi->ifi_flags & IFF_MULTICAST) {
24             Mcast_join(sockfd, mcastsa, salen, ifi->ifi_name, 0);
25             printf("joined %s on %s\n",
26                    Sock_ntop(mcastsa, salen), ifi->ifi_name);
27         }
28     }
29 #endif

30     from = Malloc(salen);
31     for ( ; ; ) {
32         len = salen;
33         n = Recvfrom(sockfd, buf, sizeof(buf), 0, from, &len);
34         Gettimeofday(&now, NULL);
35         sntp_proc(buf, n, &now);
36     }
37 }
```

Bind wildcard address to socket

15–18 We allocate space for another socket address structure and fill it in by copying the structure that was filled in by `udp_client`. This sets the address family and port. We call our `sock_set_wild` function to set the IP address to the wildcard and then call `bind`.

Get interface list

20–22 Our `get_ifi_info` function returns information on all the interfaces and addresses. The address family that we ask for is taken from the socket address structure that was filled in by `udp_client` based on the command-line argument.

Join multicast group

23–27 We call our `mcast_join` function to join the multicast group specified by the command-line argument for each multicast-capable interface. All these joins are done on the one socket that this program uses. As we said earlier, there is normally a limit of `IP_MAX_MEMBERSHIPS` (often 20) joins per socket, but few multihomed hosts have that many interfaces.

Read and process all NTP packets

30–36 Another socket address structure is allocated to hold the address returned by `recvfrom` and the program enters an infinite loop, reading all the NTP packets that the host receives and calling our `sntp_proc` function (described next) to process the packet. Since the socket was bound to the wildcard address, and since the multicast group was joined on all multicast-capable interfaces, the socket should receive any unicast, broadcast, or multicast NTP packet that the host receives. Before calling `sntp_proc`, we call `gettimeofday` to fetch the current time, because `sntp_proc` calculates the difference between the time in the packet and the current time.

Our `sntp_proc` function, shown in Figure 21.22, processes the actual NTP packet.

Validate packet

10–21 We first check the size of the packet and then print the version, mode, and server stratum. If the mode is `MODE_CLIENT`, the packet is a client request, not a server reply, and we ignore it.

Obtain transmit time from NTP packet

22–33 The field in the NTP packet that we are interested in is `xmt`, the transmit timestamp, which is the 64-bit fixed-point time at which the packet was sent by the server. Since NTP timestamps count seconds beginning in 1900 and Unix timestamps count seconds beginning in 1970, we first subtract `JAN_1970` (the number of seconds in these 70 years) from the integer part.

The fractional part is a 32-bit unsigned integer between 0 and 4,294,967,295, inclusive. This is copied from a 32-bit integer (`useci`) to a double-precision floating-point variable (`usecf`) and then divided by 4,294,967,296 (2^{32}). The result is greater than or equal to 0.0 and less than 1.0. We multiply this by 1,000,000, the number of microseconds in a second, storing the result as a 32-bit unsigned integer in the variable `useci`. This is the number of microseconds and will be between 0 and 999,999 (see Exercise 21.5). We convert to microseconds because the Unix timestamp returned by `gettimeofday` is returned as two integers: the number of seconds since January 1, 1970, UTC, along with the number of microseconds. We then calculate and print the difference between the host's time-of-day and the NTP server's time-of-day, in microseconds.

Figure 21.22 `sntp_proc` function: processes the NTP packet.

ssntp/sntp_proc.c

```
 1 #include    "sntp.h"

 2 void
 3 sntp_proc(char *buf, ssize_t n, struct timeval *nowptr)
 4 {
 5     int     version, mode;
 6     uint32_t nsec, useci;
 7     double  usecf;
 8     struct timeval diff;
 9     struct ntpdata *ntp;

10     if (n < (ssize_t) sizeof(struct ntpdata)) {
11         printf("\npacket too small: %d bytes\n", n);
12         return;
13     }

14     ntp = (struct ntpdata *) buf;
15     version = (ntp->status & VERSION_MASK) >> 3;
16     mode = ntp->status & MODE_MASK;
17     printf("\nv%d, mode %d, strat %d, ", version, mode, ntp->stratum);
18     if (mode == MODE_CLIENT) {
19         printf("client\n");
20         return;
21     }

22     nsec = ntohl(ntp->xmt.int_part) - JAN_1970;
23     useci = ntohl(ntp->xmt.fraction);   /* 32-bit integer fraction */
24     usecf = useci;                 /* integer fraction -> double */
25     usecf /= 4294967296.0;       /* divide by 2**32 -> [0, 1.0) */
26     useci = usecf * 1000000.0;   /* fraction -> parts per million */

27     diff.tv_sec = nowptr->tv_sec - nsec;
28     if ( (diff.tv_usec = nowptr->tv_usec - useci) < 0) {
29         diff.tv_usec += 1000000;
30         diff.tv_sec--;
31     }
32     useci = (diff.tv_sec * 1000000) + diff.tv_usec; /* diff in microsec */
33     printf("clock difference = %d usec\n", useci);
34 }
```

One thing that our program does not take into account is the network delay between the server and the client. But we assume that the NTP packets are normally received as a broadcast or multicast on a LAN, in which case, the network delay should be only a few milliseconds.

If we run this program on our host `macosx` with an NTP server on our host `freebsd4`, which is multicasting NTP packets to the Ethernet every 64 seconds, we have the following output:

```
macosx # ssntp 224.0.1.1
joined 224.0.1.1.123 on lo0
joined 224.0.1.1.123 on en1

v4, mode 5, strat 3, clock difference = 661 usec

v4, mode 5, strat 3, clock difference = -1789 usec
```

```
v4, mode 5, strat 3, clock difference = -2945 usec

v4, mode 5, strat 3, clock difference = -3689 usec

v4, mode 5, strat 3, clock difference = -5425 usec

v4, mode 5, strat 3, clock difference = -6700 usec

v4, mode 5, strat 3, clock difference = -8520 usec
```

To run our program, we first terminated the normal NTP server running on this host, so when our program starts, the time is very close to the server's time. We see this host lost 9181 microseconds in the 384 seconds we ran the program, or about 2 seconds in 24 hours.

[Team LiB] PREVIOUS NEXT