Network Programming

Dr. Thaier Hayajneh

Computer Engineering Department

Multicasting II

Outline

- Multicasting (Chapter 21)
  - Sending and Receiving Messages
  - Multicasting on a LAN
  - Multicasting on a WAN
  - Multicast Issues
  - Examples

Sending & Receiving Multicast Messages

Receiving Multicast Messages

- Create a UDP socket
- Bind it to a UDP port, e.g., 123
  - All processes must bind to the same port in order to receive the multicast messages
- Join a multicast group address
- Use *recv* or *recvfrom* to read the messages

Sending Multicast Messages

- You may use the same socket (you used for receiving) for sending multicast messages or you can use any other UDP socket (it does not have to join any multicast group)

Multicast on a LAN 1/4

- Receiving application creates a UDP socket, binds to port 123 and joins multicast group 224.0.1.1
  - IPv4 layers saves the information internally and tells appropriate datalink to receive Ethernet frames destined to 01:00:5E:00:01:01
- Sending applications creates a UDP socket and sends a datagram to 224.0.1.1, port 123
- Ethernet frame contains destination Ethernet address, destination IP address, and destination port
- A host on the LAN that did not express interest in receiving multicast from that group will ignore such datagram
  - Destination Ethernet address does not match the interface address
  - Destination Ethernet address is not the Ethernet broadcast address
  - The interface has not been told to receive any group addresses

Multicast on a LAN 2/4

•Ethernet frame received by datalink of receiver based on *imperfect filtering* (When interface told to receive frames destined to one specific Ethernet multicast address, it can receive frames destined to other Ethernet multicast addresses)

- ➢Ethernet interface cards apply a hash function to group address, calculating a value between 0 and 511. This information turns on a bit in a 512-bit array
- ➢Small size bit-array implies receiving unwanted frames (old cards)
- ➢Some network cards provide *perfect filtering*
- ➢Some network cards have no multicast filtering at all (multicast promiscuous mode)

•Packet passed to IP layer (IP layer compares group address against all multicast addresses that applications on this host have joined → *perfect filtering*)

•Packet passed to UDP layer, which passes it to socket bound to port 123

Multicast on a LAN 3/4

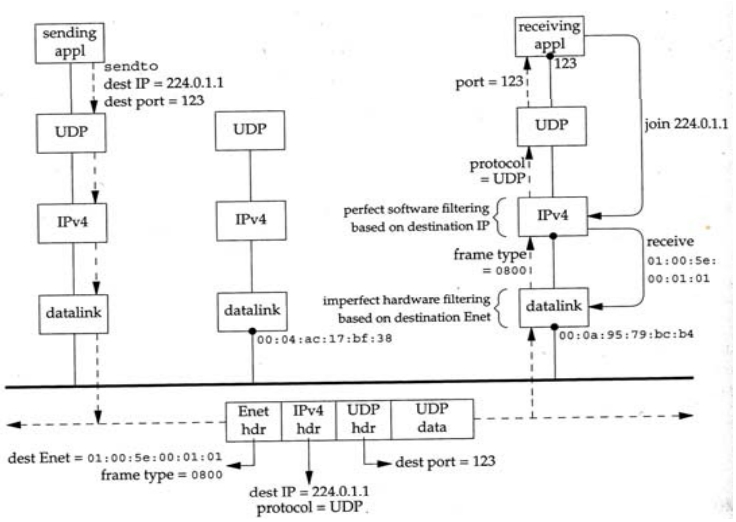
Some Other scenarios

•A host running an application that has joined 225.0.1.1 → Ethernet address 01:00:5E:00:01:01. Packet will be discarded by perfect filtering in IP layer

•A host running an application that has joined some multicast group which the Ethernet address produces the same hash value as 01:00:5E:00:01:01. Packet will be discarded by datalink layer or by IP layer

•A packet destined to the same group, but a different port. Accepted by IP layer, but discarded by UDP layer (no socket has bound the different port)

Multicast on a LAN 4/4



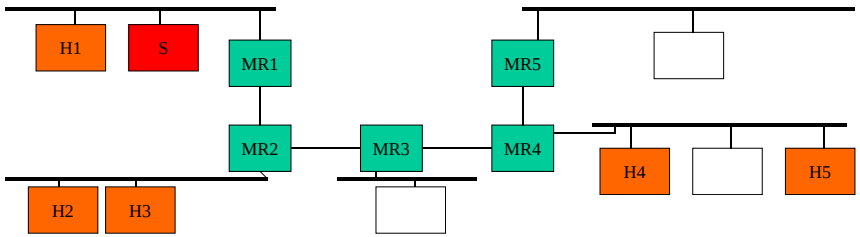
Multicast on a WAN

•A program started on five hosts belonging to different LANs

•Multicast routers communicate with neighbor routers using a multicast routing protocol (MRP)

•When a process on a host joins a multicast group, that host sends an IGMP message to any attached multicast routers, which in turn exchange this information using MRP with neighbor routers

•When a sender sends a multicast message, multicast routing information is used to direct the message



Some Multicast Issues

Time To Live

Set TTL for outgoing multicast datagrams (default is 1 → local subnet)

Loopback mode

- Enable or disable local loopback of multicast datagrams
- By default loopback is enabled
- A copy of each multicast datagram sent by a process on the host will also be looped back and processed as a received datagram by that host

Port Reuse

- Allow the same multicast application to have several instances running on the same host
- In Java, Port reuse is enabled by default, in C it is not

Socket Options 1/2

- Various attributes that are used to determine the behavior of sockets (see chapter 7)

```
#include <sys/socket.h>
int getsockopt (int sockfd, int level, int optname, void * optval,
socklen_t *optlen);
int setsockopt (int sockfd, int level, int optname, const void * optval,
socklen_t optlen);
```

Both return 0 if OK, -1 on error

- *sockfd*: an open socket descriptor
- *level*: code in the system that interprets the option (general socket code, or protocol-specific code) (SOL_SOCKET, IPPROTO_IP, IPPROTO_IPv6, IPPROTO_TCP are examples)
- *optname*: see page 193-figure 7.1, and page 194-figure 7.2

Socket Options 2/2

Some socket options examples (see table on page 193 and 194)

For multicast socket options see section 21.6 on page 559

For multicast group membership socket options, see page 560

- *Socket Level*
  - SO_SNDBUF, SO_RCVBUF, SO_KEEPALIVE, SO_BROADCAST, SO_REUSEADDR, SO_RESUEPORT
- *IP Level*
  - IP_TTL, IPMULTICAST_IF, IPMUTLICAST_TTL, IP_MULTICAST_LOOP, IP_ADD_MEMBERSHIP, IP_DROP_MEMBERSHIP
- *TCP Level*
  - TCP_KEEPALIVE, TCP_MAXSEG, TCP_NODELAY

Add Membership Socket Option 1/2

- *Option*: IP_ADD_MEMBERSHIP
- *Parameter*: Multicast address structure
- *Operation*
  - ✓Supports "JoinHostGroup" of RFC 1112 - allows a host's interface to join a multicast group
    - ❑Required to receive multicast datagrams
    - ❑Not required to send multicast datagrams
  - ✓Each interface can be in multiple groups
  - ✓Multiple interfaces can be in the same group
  - ✓Causes host to send IGMP report if this is a new group address for this host
  - ✓Tells network adapter multicast group address

Add Membership Socket Option 2/2

Example call to setsockopt():

```
setsockopt(
        sock,                   /* socket */
        IPPROTO_IP,             /* level */
        IP_ADD_MEMBERSHIP,      /*option */
        (char *) &mreq,         /* argument */
        sizeof(mreq)            /* argument
size*/
);
```


Multicast Address Structure

- specifies the multicast group address and the interface
  - Interface specified as an IP address
  - INADDR_ANY specifies use of the default multicast interface

```
struct ip_mreq {
    struct in_addr imr_multiaddr; // group
    struct in_addr imr_interface; // interface
}

char group[]="234.5.6.7";
mreq.imr_multiaddr.s_addr = inet_addr(group);
mreq.imr_interface.s_addr = INADDR_ANY;
```


Reusing Port Numbers 1/2

- What if you want to have multiple sockets on the same host listen to the same multicast group?
  - Need to bind the same port number to all sockets
  - This will cause an error when bind is called for the second and later sockets … unless socket has been set to reuse address
- Set SO_REUSEADDR socket option → allows completely duplicate bindings
  - A bind on an IP address and a port, when that same IP address and port are already bound to another socket (only for UDP sockets for multicast)

```
OptValue = 1;
setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, (char *)
&OptValue, sizeof(OptValue));
```


Reusing Port Numbers 2/2

- SO_REUSEPORT
  - Can use SO_REUSEPORT socket option which was introduced in 4.4 BSD
  - Allows completely duplicate bindings, only if each socket that wants to bind the same IP address and port specify this socket option
  - Not supported by all systems
  - SO_REUSEADDR considered equivalent to SO_REUSEPORT if the IP address being bound is a multicast address
  - *Conclusion → When writing a multicast application that can run multiple times on the same host at the same time, set SO_REUSEADDR option and bind group's multicast address as the local IP address*

Drop Membership Socket Option 1/2

- *Option*: IP_DROP_MEMBERSHIP
- *Parameter*: Multicast address structure
- *Operation*
  - ✓Supports "LeaveHostGroup" of RFC 1112- allows host to leave a multicast group
  - ✓Host's TCP/IP implementation maintains a counter for each group address
    - ❑Incremented for IP_ADD_MEMBERSHIP
    - ❑Decremented for IP_DROP_MEMBERSHIP
  - ✓If count reaches zero
    - ❑Tells adapter to drop multicast address
    - ❑Won't report group address for IGMP query
  - ✓When a socket is closed, membership dropped automatically

Drop Membership Socket Option 2/2

- Drop membership socket option
  - Need to set group address and interface in ip_mreq structure (same values as used with IP_ADD_MEMBERSHIP)
  - Example call to setsockopt():

```
setsockopt(
    sock,                   /* socket */
    IPPROTO_IP,             /* level */
    IP_DROP_MEMBERSHIP,     /* option */
    (char *) &mreq,         /* argument */
    sizeof(mreq)            /* argument size */
);
```


Receiving Multicast Data

- Create a standard SOCK_DGRAM socket
- Set SO_REUSEADDR option for socket
- Bind address to socket
  - Specify IP address as multicast address
  - Specify port
- Set IP_ADD_MEMBERSHIP option for socket
  - Specify host group address
- After these steps completed successfully, receive multicast data for specified group address and port using recvfrom()
- Drop group membership when finished using IP_DROP_MEMBERSHIP option

Sending Multicast Data

- Use standard SOCK_DGRAM socket
- Sending alone does not require group membership
- To send multicast datagrams:
  - Use sendto() to send to appropriate group address and port number, or
  - Use connect() to set group address and port and then use send()
- Concerns (controlled with socket options)
  - Interface used to send: IP_MULTICAST_IF (relevant to hosts with multiple interfaces)
  - Extent of multicast: IP_MULTICAST_TTL
  - Receiving own data: IP_MULTICAST_LOOP

Time to Live Socket Option 1/2

- *Option*: IP_MULTICAST_TTL
- *Parameter*: TTL value (int)
- *Operation*
  - ✓Controls the time-to-live (TTL) value that IP will use for multicast datagrams
  - ✓Default TTL is 1 — multicast datagrams will not leave the local network
  - ✓To send multicast datagrams beyond the local network …
    - ❑TTL must be greater than 1, *and*
    - ❑Intermediate routers must support multicast
  - ✓Group address 224.0.0.0 — 224.0.0.255 not routed, regardless of TTL value
  - ✓A TTL = 0 will confine to local host

Time to Live Socket Option 2/2

```
int ttl = 5;

setsockopt(
        sock,                   /* socket */
        IPPROTO_IP,             /* level */
        IP_MULTICAST_TTL,       /* option */
        (char *) &ttl,          /* argument */
        sizeof(ttl)             /*argument size*/
);
```


Multicast Loopback Socket Option 1/2

- *Option*: IP_MULTICAST_LOOP
- *Parameter*: Boolean (TRUE to enable)
- *Operation*
  - ✓If enabled (*default*), socket will receive a copy of multicast datagrams that were sent on that socket
  - ✓Even if disabled, host with two interfaces may receive a copy on the other interface(s)
  - ✓This is an internal loopback performed at the IP layer

Multicast Loopback Socket Option 2/2

```
BOOL opt = FALSE;

setsockopt(
        sock,                   /* socket */
        IPPROTO_IP,             /* level */
        IP_MULTICAST_LOOP,      /* option */
        (char *) &opt,          /* argument */
        sizeof(opt)             /* argument size */
);
```

Textbook Multicast Helper Functions

➢See section 21.7 on page 565

int mcast_join (int *sockfd*, const struct sockaddr**grp*, socklen_t *grplen*, const char* *ifname*, u_int *ifindex*);

int mcast_leave(int *sockfd*, const struct sockaddr **grp*, socklen_t *grplen*);

int mcast_set_loop (int *sockfd*, int *flag*)

int mcast_set_ttl (int *sockfd*, int *ttl*);

//All Above return 0 if OK, -1 on error

int mcast get_loop (int *sockfd*);

int mcast_get_ttl (int *sockfd*);

//return value is OK, -1 on error

Example for Sending and Receiving

•Section 21.10 page 575

•Function *udp_client* introduced in 11.14 on page 334

➢Creates an unconnected UDP socket

➢Return value is the socket descriptor

•A program to send and receive multicast datagrams

➢Send datagram to a specific group every five seconds (datagram contains sender's hostname and process ID)

➢An infinite loop that joins the multicast group to which the sending part is sending and prints every received datagram

•Create a UDP socket then set multicast socket options for address reuse, joining the group, and setting loopback

•See mcast/main.c, mcast/send.c, and mcast/recv.c

Sockets Timeout

•Section 14.2 page 381

•There are three ways to place a timeout on an I/O operation involving a socket.

1. Call alarm which generates the SIGALARM signal when the specified time has expired.
2. Block waiting for I/O in select, which has a time limit build-in
3. Use the newer SO_RCVTIMEO and SO_SNDTIMEO socket options.

Connect with a Timeout Using SIGALRM (figure 13.1)

```
#include     "unp.h"

static void  connect_alarm(int);

int connect_timeo(int sockfd, const SA *saptr, socklen_t salen, int nsec)
{
   Sigfunc   *sigfunc;
   int               n;

   sigfunc = Signal(SIGALRM, connect_alarm);
   if (alarm(nsec) != 0)
     err_msg("connect_timeo: alarm was already set");

   if ( (n = connect(sockfd, (struct sockaddr *) saptr, salen)) < 0) {
     close(sockfd);
     if (errno == EINTR)
             errno = ETIMEDOUT;
   }
   alarm(0);                                   /* turn off the alarm */
   Signal(SIGALRM, sigfunc);  /* restore previous signal handler */

   return(n);
}

static void
connect_alarm(int signo)
{
   return;            /* just interrupt the connect() */
}
```

recvfrom with a Timeout Using SIGALRM (figure 13.2)

```
#include "unp.h"

static void      sig_alrm(int);

void dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
{
   int  n;
   char sendline[MAXLINE], recvline[MAXLINE + 1];

   Signal(SIGALRM, sig_alrm);

   while (Fgets(sendline, MAXLINE, fp) != NULL) {

         Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);

         alarm(5);
         if ( (n = recvfrom(sockfd, recvline, MAXLINE, 0, NULL, NULL)) < 0) {
                  if (errno == EINTR)
                           fprintf(stderr, "socket timeout\n");
                  else
                           err_sys("recvfrom error");
         } else {
                  alarm(0);
                  recvline[n] = 0; /* null terminate */
                  Fputs(recvline, stdout);
         }
   }
}

static void sig_alrm(int signo)
{
   return;                        /* just interrupt the recvfrom() */
}
```


recvfrom with a Timeout Using select (figure 13.3)

```
#include   "unp.h"

int
readable_timeo(int fd, int sec)
{
  fd_set                  rset;
  struct timeval  tv;

  FD_ZERO(&rset);
  FD_SET(fd, &rset);

  tv.tv_sec = sec;
  tv.tv_usec = 0;

  return(select(fd+1, &rset, NULL, NULL, &tv));
    /* 4> 0 if descriptor is readable */
}
```