LEC 08.
BROADCASTING & MULTICASTING

Bui Trong Tung, SoICT, HUST

Content

- Non-blocking I/O
- Signal-driven I/O
- Some advanced I/O functions

BROADCASTING

Bui Trong Tung, SoICT, HUST

Broadcasting

- Send data to all nodes in the network
- Broadcasting datagram transport such as UDP or raw IP; they cannot work with TCP
- Broadcast address:
  - Subnet-directed broadcast address

NetworkID	**111……..111**

  - Local broadcast address: 255.255.255.255
- Set broadcast mode on UDP socket: `setsopt()` with SO_BROADCAST

Example: UDP broadcasting sender

```
int    sockfd, n, msg_len;
struct sockaddr_in   servaddr, *preply_addr;
const int on = 1;
char sendbuff[BUFF_SIZE], recvbuff[BUFF_SIZE + 1];
socklen_t len, servlen;
//Construct UDP socket
//…
servlen = sizeof(servaddr);
bzero(&servaddr, servlen);
servaddr.sin_family = AF_INET;
servaddr.sin_port = htons(SERV_PORT);
inet_pton(AF_INET, BROADCAST_ADDR, &servaddr.sin_addr);

//set brooadcasting mode on socket
setsockopt(sockfd, SOL_SOCKET, SO_BROADCAST, &on, sizeof(on));

//signal handler is installed for SIGALRM
signal(SIGALRM, recvfrom_alarm);
```

Example: UDP broadcasting sender

```
while(1){
   printf("\nInsert string to send:");
   fgets(sendbuff, BUFF_SIZE, stdin);
   msg_len = strlen(sendbuff);
   if (msg_len == 0) break;

   sendto(sockfd, sendbuff, msg_len, 0,
                        (struct sockaddr*)&servaddr, servlen);
   alarm(5);
   while (1) {
        len = servlen;
        n = recvfrom(sockfd, recvbuff, BUFF_SIZE, 0,
                        (struct sockaddr *)preply_addr, &len);
        if (n < 0) {
           if (errno == EINTR)
                break; /* waited long enough for replies */
           else
                perror("recvfrom error");
        } else {
           recvline[n] = 0;    /* null terminate */
           printf("from %s: %s", recvline);
        }}}
```

MULTICASTING

Bui Trong Tung, SoICT, HUST

Multicast

- Send data to all nodes on the same group
- Multicasting datagram transport such as UDP or raw IP; they cannot work with TCP
- Multicast address:

Scope	**IPv6**	**IPv4**	
		TTL	Địa chỉ
Interface local	1	0	
Link local	2	1	224.0.0.0 – 224.0.0.255
Site-local	5	<32	239.255.0.0 – 239.255.255.255
Organization-local	8		239.192.0.0 – 239.192.255.255
Global	14	<= 255	224.0.1.0 – 238.255.255.255

Multicast – socket option

- Use setsockopt() to control multicasting mode on socket
  - *level*: IPPROTO_IP
- 3 nonmembership-related socket options
  - IP_MULTICAST_IF: choose a concrete outgoing interface for a given socket
    - Datatype: struct in_addr
  - IP_MULTICAST_TTL: change the TTL to the specified value
    - Data type: u_char
  - IP_MULTICAST_LOOP: send multicast message loopback to the host or not:
    - Data type: u_char

9

Multicast – socket option(cont)

6 membership-related socket options for IPv4

- IP_ADD_MEMBERSHIP: join to a multicast address
  - Datatype: ip_mreq
  - the *imr_interface* member can be filled with INADDR_ANY
- IP_DROP_MEMBERSHIP: leave the multicast address
  - Data type: ip_mreq
  - If the *imr_interface* member is filled with INADDR_ANY, the first matching group is dropped.

```
#include <linux/in.h>
struct ip_mreq
{
   struct in_addr imr_multiaddr; // multicast address of group
   struct in_addr imr_interface; // local IP address of interface
};
```

10

Multicast – socket option(cont)

- IP_BLOCK_SOURCE: block receipt of traffic on this socket from a source
  - Datatype: ip_mreq_source
  - If the local interface is specified as INADDR_ANY, the local interface is chosen by the kernel to match the first membership on this socket
  - The specified interface must join into the group
- IP_DROP_MEMBERSHIP: Unblock a previously blocked source
  - Data type: ip_mreq_source

```
#include <linux/in.h>
struct ip_mreq_source {
  struct in_addr   imr_multiaddr;  // IPv4 class D multicast addr
  struct in_addr   imr_sourceaddr; // IPv4 source addr
  struct in_addr   imr_interface;  //IPv4 addr of local interface
};
```

11

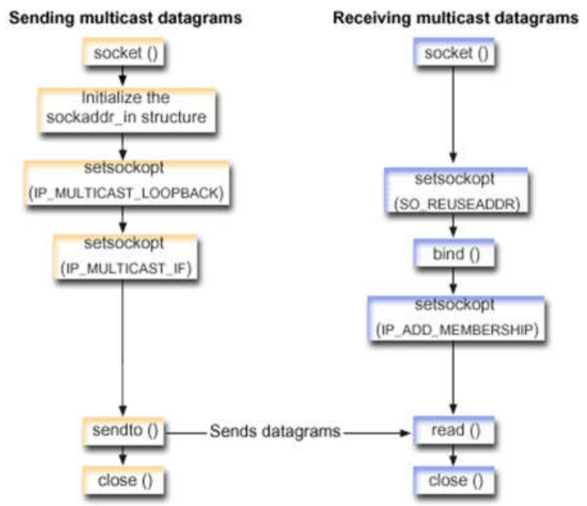
Multicast – socket option(cont)

- IP_ADD_SOURCE_MEMBERSHIP: Join a source-specific group on a specified local interface
  - Datatype: ip_mreq_source
  - If the *imr_interface* member is filled with INADDR_ANY, the local interface is chosen by the kernel
- IP_DROP_SOURCE_MEMBERSHIP: Leave a source-specific group on a specified local interface
  - Data type: ip_mreq_source

```
#include <linux/in.h>
struct ip_mreq_source {
  struct in_addr   imr_multiaddr;  // IPv4 class D multicast addr
  struct in_addr   imr_sourceaddr; // IPv4 source addr
  struct in_addr   imr_interface;  //IPv4 addr of local interface
};
```

12

Multicast - Example


Sending multicast datagrams
socket ()
Initialize the sockaddr_in structure
setsockopt (IP_MULTICAST_LOOPBACK)
setsockopt (IP_MULTICAST_IF)
sendto ()
close ()
Sends datagrams
Receiving multicast datagrams
socket ()
setsockopt (SO_REUSEADDR)
bind ()
setsockopt (IP_ADD_MEMBERSHIP)
read ()
close ()

Socket flow of events: Sending multicast

1. Construct SOCK_DGRAM socket.
2. The sockaddr_in structure specifies the destination IP address and port number. In this example, the address is 225.1.1.1 and the port number is 5555.
3. The setsockopt() API sets the IP_MULTICAST_LOOP socket option so that the sending system does not receive a copy of the multicast datagrams it transmits.
4. The setsockopt() API uses the IP_MULTICAST_IF socket option, which defines the local interface over which the multicast datagrams are sent.
5. The sendto() API sends multicast datagrams to the specified group IP addresses.
6. The close() API closes any open socket descriptors.

Sending multicast

```
int main (int argc, char *argv[])
{
   struct in_addr localInterface;
   struct sockaddr_in groupSock;
   int sendfd;
   int datalen;
   char databuf[32];

   // Create a datagram socket on which to send.
   sendfd = socket(AF_INET, SOCK_DGRAM, 0);

   // Initialize the group sockaddr structure with a
   // group address of GROUP_ADDR and port PORT.
   memset((char *) &groupSock, 0, sizeof(groupSock));
   groupSock.sin_family = AF_INET;
   groupSock.sin_addr.s_addr = inet_addr(GROUP_ADDR);
   groupSock.sin_port = htons(PORT);
```

Sending multicast

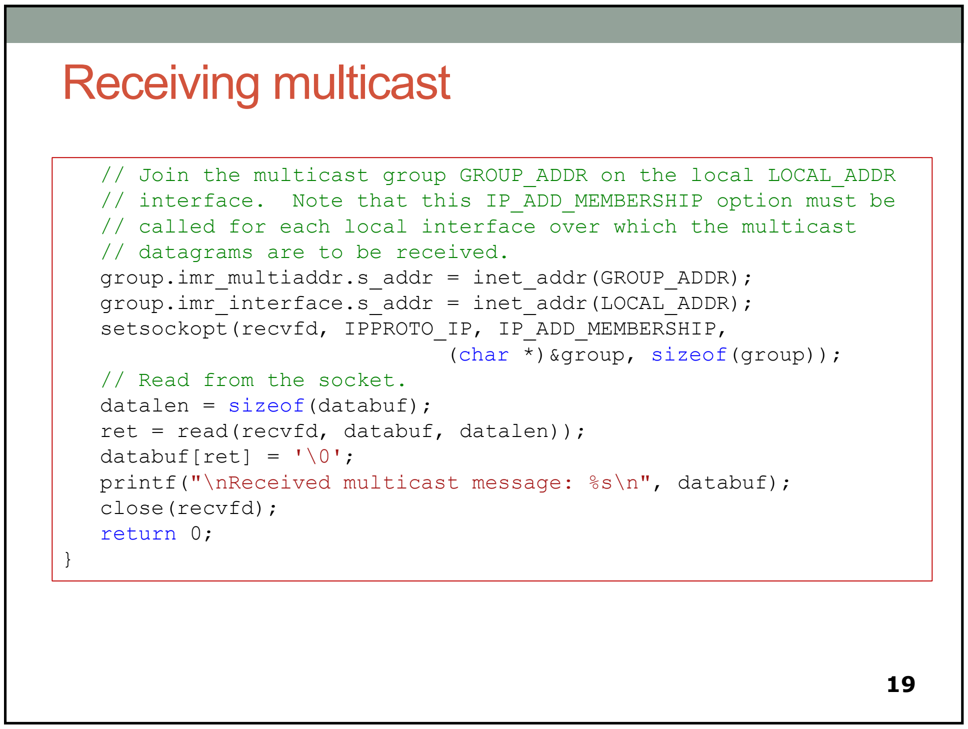
```
   // Disable loopback so do not receive your own datagrams.
   char loopch=0;
   setsockopt(sendfd, IPPROTO_IP, IP_MULTICAST_LOOP,
                           (char *)&loopch, sizeof(loopch));
   // Set local interface for outbound multicast datagrams.
   // The IP address specified must be associated with a
   // local, multicast-capable interface.
   localInterface.s_addr = inet_addr(LOCAL_ADDR);
   setsockopt(sendfd, IPPROTO_IP, IP_MULTICAST_IF,
              (char *)&localInterface, sizeof(localInterface));
   // Send a message to the multicast group specified by the
   // groupSock sockaddr structure.
   strcpy(databuf, MESSAGE);
   datalen = strlen(databuf);
   sendto(sendfd, databuf, datalen, 0,
              (struct sockaddr*)&groupSock, sizeof(groupSock));
   close(sendfd);
   return 0;
}
```

Socket flow of events: Receiving multicast

1. Construct SOCK_DGRAM socket.
2. The setsockopt() API sets the SO_REUSEADDR socket option to allow multiple applications to receive datagrams that are destined to the same local port number.
3. The bind() API specifies the local port number. In this example, the IP address is specified as INADDR_ANY to receive datagrams that are addressed to the multicast group.
4. The setsockopt() API uses the IP_ADD_MEMBERSHIP socket option, which joins the multicast group that receives the datagrams
5. The read() API reads multicast datagrams that are being sent.
6. The close() API closes any open socket descriptors.

Receiving multicast

```
int main (int argc, char *argv[])
{
   struct sockaddr_in localSock;
   struct ip_mreq group;
   int recvfd;
   int datalen, ret;
   char databuf[32];
   // Create a datagram socket on which to receive.
   recvfd = socket(AF_INET, SOCK_DGRAM, 0);
   // Enable SO_REUSEADDR to allow multiple instances of this
   // application to receive copies of the multicast datagrams
   int reuse=1;
   setsockopt(recvfd, SOL_SOCKET, SO_REUSEADDR,
                              (char *)&reuse, sizeof(reuse));
   // Bind to the proper port number with the IP address
   // specified as INADDR_ANY.
   memset((char *) &localSock, 0, sizeof(localSock));
   localSock.sin_family = AF_INET;
   localSock.sin_port = htons(PORT);;
   localSock.sin_addr.s_addr  = INADDR_ANY;
   bind(recvfd, (struct sockaddr*)&localSock, sizeof(localSock));
```



Receiving multicast

```
   // Join the multicast group GROUP_ADDR on the local LOCAL_ADDR
   // interface.  Note that this IP_ADD_MEMBERSHIP option must be
   // called for each local interface over which the multicast
   // datagrams are to be received.
   group.imr_multiaddr.s_addr = inet_addr(GROUP_ADDR);
   group.imr_interface.s_addr = inet_addr(LOCAL_ADDR);
   setsockopt(recvfd, IPPROTO_IP, IP_ADD_MEMBERSHIP,
                              (char *)&group, sizeof(group));
   // Read from the socket.
   datalen = sizeof(databuf);
   ret = read(recvfd, databuf, datalen));
   databuf[ret] = '\0';
   printf("\nReceived multicast message: %s\n", databuf);
   close(recvfd);
   return 0;
}
```

19