6. Multicast programming.

Multicast programming... or writing your own multicast applications.

Several extensions to the programming API are needed in order to support multicast. All of them are handled via two system calls: `setsockopt()` (used to pass information to the kernel) and `getsockopt()` (to retrieve information regarded multicast behavior). This does *not* mean that 2 new system calls were added to support multicast. The pair `setsockopt()`/`getsockopt()` has been there for years. Since 4.2 BSD at least. The addition consists on a new set of options (multicast options) that are passed to these system calls, that the kernel must understand.

The following are the `setsockopt()`/`getsockopt()` function prototypes:

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

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

The first parameter, `s`, is the socket the system call applies to. For multicasting, it must be a socket of the family `AF_INET` and its type may be either `SOCK_DGRAM` or `SOCK_RAW`. The most common use is with `SOCK_DGRAM` sockets, but if you plan to write a routing daemon or modify some existing one, you will probably need to use `SOCK_RAW` ones.

The second one, `level`, identifies the layer that is to handle the option, message or query, whatever you want to call it. So, `SOL_SOCKET` is for the socket layer, `IPPROTO_IP` for the IP layer, etc... For multicast programming, `level` will always be `IPPROTO_IP`.

`optname` identifies the option we are setting/getting. Its value (either supplied by the program or returned by the kernel) is `optval`. The optnames involved in multicast programming are the following:

	setsockopt()	getsockopt()
IP_MULTICAST_LOOP	yes	yes
IP_MULTICAST_TTL	yes	yes
IP_MULTICAST_IF	yes	yes
IP_ADD_MEMBERSHIP	yes	no
IP_DROP_MEMBERSHIP	yes	no

`optlen` carries the size of the data structure `optval` points to. Note that in `getsockopt()` it is a value-result rather than a value: the kernel writes the value of `optname` in the buffer pointed by `optval` and informs us of that value's size via `optlen`.

Both `setsockopt()` and `getsockopt()` return 0 on success and -1 on error.

6.1 IP_MULTICAST_LOOP.

You have to decide, as the application writer, whether you want the data you send to be looped back to your host or not. If you plan to have more than one process or user "listening", loopback must be enabled. On the other hand, if you are sending the images your video camera is producing, you probably don't want loopback, even if you want to see yourself on the screen. In that latter case, your application will probably receive the images from a device attached to the computer and send them to the socket. As the application already "has" that data, it is improbable it wants to receive it again on the socket. Loopback is by default enabled.

Regard that `optval` is a pointer. You can't write:

```
setsockopt(socket, IPPROTO_IP, IP_MULTICAST_LOOP, 0, 1);
```

to disable loopback. Instead write:

```
u_char loop;
setsockopt(socket, IPPROTO_IP, IP_MULTICAST_LOOP, &loop, sizeof(loop));
```

and set `loop` to 1 to enable loopback or 0 to disable it.

To know whether a socket is currently looping-back or not use something like:

```
u_char loop;
int size;

getsockopt(socket, IPPROTO_IP, IP_MULTICAST_LOOP, &loop, &size)
```

6.2 IP_MULTICAST_TTL.

If not otherwise specified, multicast datagrams are sent with a default value of 1, to prevent them to be forwarded beyond the local network. To change the TTL to the value you desire (from 0 to 255), put that value into a variable (here I name it "ttl") and write somewhere in your program:

```
u_char ttl;
setsockopt(socket, IPPROTO_IP, IP_MULTICAST_TTL, &ttl, sizeof(ttl));
```

The behavior with `getsockopt()` is similar to the one seen on IP_MULTICAST_LOOP.

6.3 IP_MULTICAST_IF.

Usually, the system administrator specifies the default interface multicast datagrams should be sent from. The programmer can override this and choose a concrete outgoing interface for a given socket with this option.

```
struct in_addr interface_addr;
setsockopt (socket, IPPROTO_IP, IP_MULTICAST_IF, &interface_addr, sizeof(interface_addr));
```

>From now on, all multicast traffic generated in this socket will be output from the interface chosen. To revert to the original behavior and let the kernel choose the outgoing interface based on the system administrator's configuration, it is enough to call `setsockopt()` with this same option and `INADDR_ANY` in the interface field.

In determining or selecting outgoing interfaces, the following `ioctl`s might be useful: `SIOCGIFADDR` (to get an interface's address), `SIOCGIFCONF` (to get the list of all the interfaces) and `SIOCGIFFLAGS` (to get an interface's flags and, thus, determine whether the interface is multicast capable or not -the `IFF_MULTICAST` flag-).

If the host has more than one interface and the IP_MULTICAST_IF option is not set, multicast transmissions are sent from the default interface, although the remainding interfaces might be used for multicast *forwarding* if the host is acting as a multicast router.

6.4 IP_ADD_MEMBERSHIP.

Recall that you need to tell the kernel which multicast groups you are interested in. If no process is interested in a group, packets destined to it that arrive to the host are discarded. In order to inform the kernel of your interests and, thus, become a member of that group, you should first fill a `ip_mreq` structure which is passed later to the kernel in the `optval` field of the `setsockopt()` system call.

The ip_mreq structure (taken from `/usr/include/linux/in.h`) has the following members:

```
struct ip_mreq
{
        struct in_addr imr_multiaddr;   /* IP multicast address of group */
        struct in_addr imr_interface;   /* local IP address of interface */
};
```

(Note: the "physical" definition of the structure is in the file above specified. Nonetheless, you should not include `<linux/in.h>` if you want your code to be portable. Instead, include `<netinet/in.h>` which, in turn, includes `<linux/in.h>` itself).

The first member, `imr_multiaddr`, holds the group address you want to join. Remember that memberships are also associated with interfaces, not just groups. This is the reason you have to provide a value for the second member: `imr_interface`. This way, if you are in a multihomed host, you can join the same group in several interfaces. You can always fill this last member with the wildcard address (`INADDR_ANY`) and then the kernel will deal with the task of

choosing the interface.

With this structure filled (say you defined it as: `struct ip_mreq mreq;`) you just have to call `setsockopt()` this way:

```
setsockopt (socket, IPPROTO_IP, IP_ADD_MEMBERSHIP, &mreq, sizeof(mreq));
```

Notice that you can join several groups to the same socket, not just one. The limit to this is `IP_MAX_MEMBERSHIPS` and, as of version 2.0.33, it has the value of 20.

6.5 IP_DROP_MEMBERSHIP.

The process is quite similar to joining a group:

```
struct ip_mreq mreq;
setsockopt (socket, IPPROTO_IP, IP_DROP_MEMBERSHIP, &mreq, sizeof(mreq));
```

where `mreq` is the same structure with the same data used when joining the group. If the `imr_interface` member is filled with `INADDR_ANY`, the first matching group is dropped.

If you have joined a lot of groups to the same socket, you don't need to drop memberships in all of them in order to terminate. When you close a socket, all memberships associated with it are dropped by the kernel. The same occurs if the process that opened the socket is killed.

Finally, keep in mind that a process dropping membership for a group does not imply that the host will stop receiving datagrams for that group. If another socket joined that group in that same interface previously to this `IP_DROP_MEMBERSHIP`, *the host* will keep being a member of that group.

Both ADD_MEMBERSHIP and DROP_MEMBERSHIP are nonblocking operations. They should return immediately indicating either success or failure.
