Table of Contents

Chapter 14. Multicast Sockets253 1
14.1. What Is a Multicast Socket?303689 2
14.2. Working with Multicast Sockets303689 13
14.3. Two Simple Examples303689 22

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

Chapter 14. Multicast Sockets

The sockets in the previous chapters are *unicast*: they provide point-to-point communication. Unicast sockets create a connection with two well-defined endpoints; there is one sender and one receiver and, although they may switch roles, at any given time it is easy to tell which is which. However, although point-to-point communications serve many, if not most needs (people have engaged in one-on-one conversations for millennia), many tasks require a different model. For example, a television station broadcasts data from one location to every point within range of its transmitter. The signal reaches every television set, whether or not it's turned on and whether or not it's tuned to that particular station. Indeed, the signal even reaches homes with cable boxes instead of antennas and homes that don't have a television. This is the classic example of broadcasting. It's indiscriminate and quite wasteful of both the electromagnetic spectrum and power.

Videoconferencing, by contrast, sends an audio-video feed to a select group of people. Usenet news is posted at one site and distributed around the world to hundreds of thousands of people. DNS router updates travel from the site, announcing a change to many other routers. However, the sender relies on the intermediate sites to copy and relay the message to downstream sites. The sender does not address its message to every host that will eventually receive it. These are examples of multicasting, although they're implemented with additional application layer protocols on top of TCP or UDP. These protocols require fairly detailed configuration and intervention by human beings. For instance, to join Usenet you have to find a site willing to send news to you and relay your outgoing news to the rest of the world. To add you to the Usenet feed, the news administrator of your news relay has to specifically add your site to their news config files. However, recent developments with the network software in most major operating systems as well as Internet routers have opened up a new possibility—true multicasting, in which the routers decide how to efficiently move a message to individual hosts. In particular, the initial router sends only one copy of the message to a router near the receiving hosts, which then makes multiple copies for different recipients at or closer to the destinations. Internet multicasting is built on top of UDP. Multicasting in Java uses the `DatagramPacket` class introduced in Chapter 13, along with a new `MulticastSocket` class.

14.1. What Is a Multicast Socket?

Multicasting is broader than unicast, point-to-point communication but narrower and more targeted than broadcast communication. Multicasting sends data from one host to many different hosts, but not to everyone; the data only goes to clients that have expressed an interest by joining a particular multicast group. In a way, this is like a public meeting. People can come and go as they please, leaving when the discussion no longer interests them. Before they arrive and after they have left, they don't need to process the information at all: it just doesn't reach them. On the Internet, such "public meetings" are best implemented using a multicast socket that sends a copy of the data to a location (or a group of locations) close to the parties that have declared an interest in the data. In the best case, the data is duplicated only when it reaches the local network serving the interested clients: the data crosses the Internet only once. More realistically, several identical copies of the data traverse the Internet; but, by carefully choosing the points at which the streams are duplicated, the load on the network is minimized. The good news is that programmers and network administrators aren't responsible for choosing the points where the data is duplicated or even for sending multiple copies; the Internet's routers handle all that.

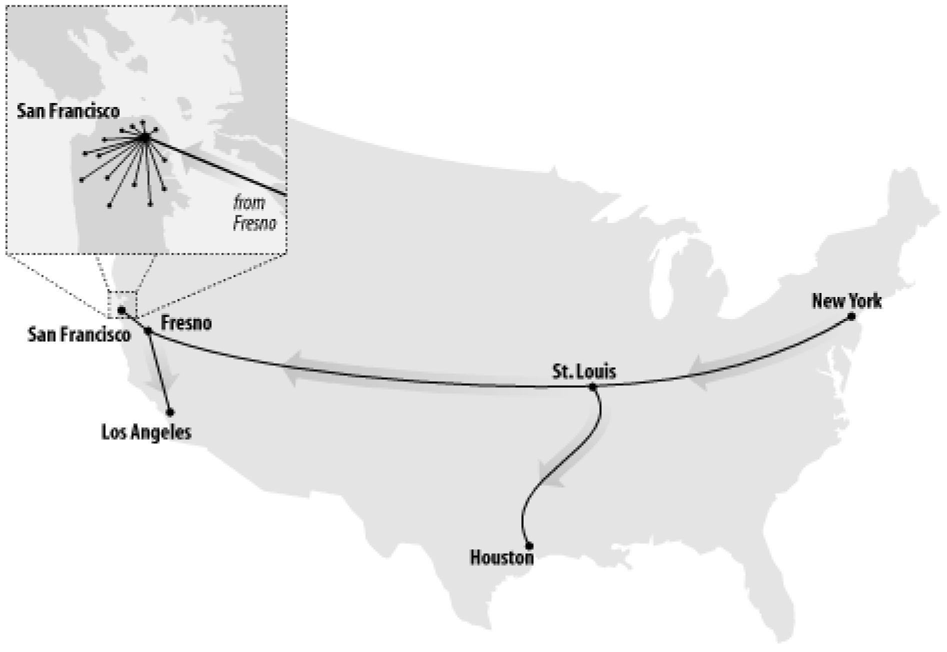
IP also supports broadcasting, but the use of broadcasts is strictly limited. Protocols require broadcasts only when there is no alternative, and routers limit broadcasts to the local network or subnet, preventing broadcasts from reaching the Internet at large. Even a few small global broadcasts could bring the Internet to its knees. Broadcasting high-bandwidth data such as audio, video, or even text and still images is out of the question. A single email spam that goes to millions of addresses is bad enough. Imagine what would happen if a real-time video feed were copied to all six hundred million Internet users, whether they wanted to watch it or not.

However, there's a middle ground between point-to-point communications and broadcasts to the whole world. There's no reason to send a video feed to hosts that aren't interested in it; we need a technology that sends data to the hosts that want it, without bothering the rest of the world. One way to do this is to use many unicast streams. If 1,000 clients want to listen to a RealAudio broadcast, the data is sent a thousand times. This is inefficient, since it duplicates data needlessly, but it's orders-of-magnitude more efficient than broadcasting the data to every host on the Internet. Still, if the number of interested clients is large enough, you will eventually run out of bandwidth or CPU power—probably sooner rather than later.

Another approach to the problem is to create static *connection trees*. This is the solution employed by Usenet news and some conferencing systems (notably CUseeMe). Data is fed from the originating site to other servers, which replicate it to still other servers, which eventually replicate it to clients. Each client connects to the nearest server. This is more efficient than sending everything to all interested clients via multiple unicasts, but the scheme is kludgy and beginning to show its age. New sites need to find a place to hook into the tree manually. The tree does not necessarily reflect the best possible topology at any one time, and servers still need to maintain many point-to-point connections to their clients, sending the same data to each one. It would be better to allow the routers in the Internet to dynamically determine the best possible routes for transmitting distributed information and to replicate data only when absolutely necessary. This is where multicasting comes in.

For example, if you're multicasting video from New York and 20 people attached to one LAN are watching the show in Los Angeles, the feed will be sent to that LAN only once. If 50 more people are watching in San Francisco, the data stream will be duplicated somewhere (let's say Fresno) and sent to the two cities. If a hundred more people are watching in Houston, another data stream will be sent there (perhaps from St. Louis); see Figure 14-1. The data has crossed the Internet only three times—not the 170 times that would be required by point-to-point connections, or the millions of times that would be required by a true broadcast. Multicasting is halfway between the point-to-point communication common to the Internet and the broadcast model of television and it's more efficient than either. When a packet is multicast, it is addressed to a multicast group and sent to each host belonging to the group. It does not go to a single host (as in unicasting), nor does it go to every host (as in broadcasting). Either would be too inefficient.

Figure 14-1. Multicast from New York to San Francisco, Los Angeles, and Houston

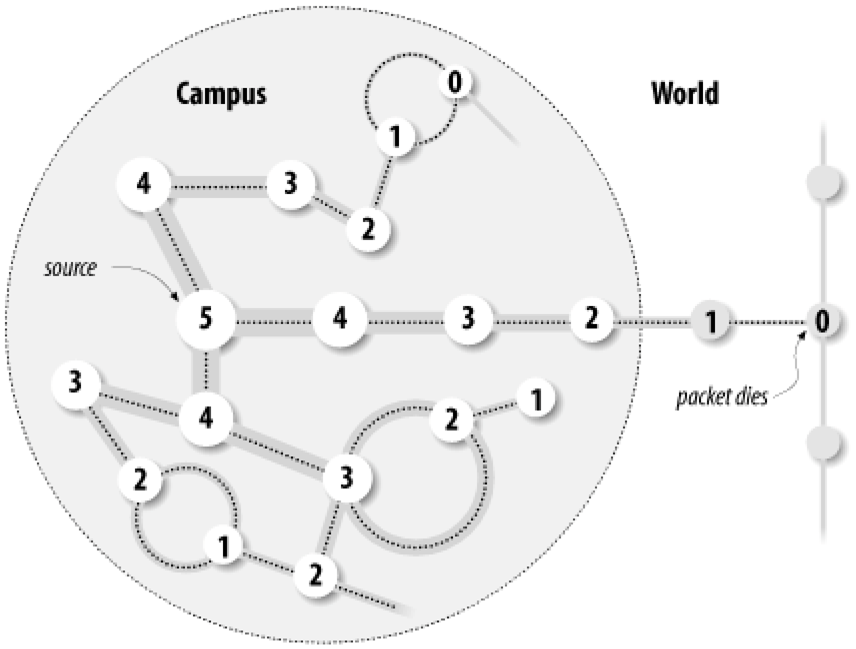


When people start talking about multicasting, audio and video are the first applications that come to mind; however, they are only the tip of the iceberg. Other possibilities include multiplayer games, distributed filesystems, massively parallel computing, multiperson conferencing, database replication, and more. Multicasting can be used to implement name services and directory services that don't require the client to know a server's address in advance; to look up a name, a host could multicast its request to some well-known address and wait until a response is received from the nearest server. Apple's Rendezvous (a.k.a. Zeroconf) and Sun's Jini both use IP multicasting to dynamically discover services on the local network.

Multicasting should also make it easier to implement various kinds of caching for the Internet, which will be important if the Net's population continues to grow faster than available bandwidth. Martin Hamilton has proposed using multicasting to build a distributed server system for the World Wide Web. ("Evaluating Resource Discovery Applications of IP Multicast", http://martinh.net/eval/eval.html, 1995.) For example, a high-traffic web server could be split across multiple machines, all of which share a single hostname, mapped to a multicast address. Suppose one machine chunks out HTML files, another handles images, and a third processes servlets. When a client makes a request to the multicast address, the request is sent to each of the three servers. When a server receives the request, it looks to see whether the client wants an HTML file, an image, or a servlet response. If the server can handle the request, it responds. Otherwise, the server ignores the request and lets the other servers process it. It is easy to imagine more complex divisions of labor between distributed servers.

Multicasting has been designed to fit into the Internet as seamlessly as possible. Most of the work is done by routers and should be transparent to application programmers. An application simply sends datagram packets to a multicast address, which isn't fundamentally different from any other IP address. The routers make sure the packet is delivered to all the hosts in the multicast group. The biggest problem is that multicast routers are not yet ubiquitous; therefore, you need to know enough about them to find out whether multicasting is supported on your network. As far as the application itself, you need to pay attention to an additional header field in the datagrams called the Time-To-Live (TTL) value. The TTL is the maximum number of routers that the datagram is allowed to cross; when it reaches the maximum, it is discarded. Multicasting uses the TTL as an ad hoc way to limit how far a packet can travel. For example, you don't want packets for a friendly on-campus game of Dogfight reaching routers on the other side of the world. Figure 14-2 shows how TTLs limit a packet's spread.

Figure 14-2. Coverage of a packet with a TTL of five



14.1.1. Multicast Addresses and Groups

A *multicast address* is the shared address of a group of hosts called a *multicast group*. We'll talk about the address first. Multicast addresses are IP addresses in the range 224.0.0.0 to 239.255.255.255. All addresses in this range have the binary digits 1110 as their first four bits. They are called Class D addresses to distinguish them from the more common Class A, B, and C addresses. Like any IP address, a multicast address can have a hostname; for example, the multicast address 224.0.1.1 (the address of the Network Time Protocol distributed service) is assigned the name *ntp.mcast.net*.

A multicast group is a set of Internet hosts that share a multicast address. Any data sent to the multicast address is relayed to all the members of the group. Membership in a multicast group is open; hosts can enter or leave the group at any time. Groups can be either permanent or transient. Permanent groups have assigned addresses that remain constant, whether or not there are any members in the group. However, most multicast groups are transient and exist only as long as they have members. All you have to do to create a new multicast group is pick a random address from 225.0.0.0 to 238.255.255.255, construct an `InetAddress` object for that address, and start sending it data.

A number of multicast addresses have been set aside for special purposes. *all-systems.mcast.net*, 224.0.0.1, is a multicast group that includes all systems that support multicasting on the local subnet. This group is commonly used for local testing, as is *experiment.mcast.net*, 224.0.1.20. (There is no multicast address that sends data to all hosts on the Internet.) All addresses beginning with 224.0.0 (i.e., addresses from 224.0.0.0 to 224.0.0.255) are reserved for routing protocols and other low-level activities, such as gateway discovery and group membership reporting. Multicast routers never forward datagrams with destinations in this range.

The IANA is responsible for handing out permanent multicast addresses as needed; so far, a few hundred have been specifically assigned. Most of these begin with 224.0., 224.1., 224.2., or 239. Table 14-1 lists a few of these permanent addresses. A few blocks of addresses ranging in size from a few dozen to a few thousand addresses have also been reserved for particular purposes. The complete list is available from http://www.iana.org/assignments/multicast-addresses. The remaining 248 million Class D addresses can be used on a temporary basis by anyone who needs them. Multicast routers (*mrouters* for short) are responsible for making sure that two different systems don't try to use the same Class D address at the same time.

Table 14-1. Common permanent multicast addresses

Domain name	IP address	Purpose
BASE-ADDRESS.MCAST.NET	*224.0.0.0*	The reserved base address. This is never assigned to any multicast group.
ALL-SYSTEMS.MCAST.NET	*224.0.0.1*	All systems on the local subnet.
ALL-ROUTERS.MCAST.NET	*224.0.0.2*	All routers on the local subnet.
DVMRP.MCAST.NET	*224.0.0.4*	All Distance Vector Multicast Routing Protocol (DVMRP) routers on this subnet. An early version of the DVMRP protocol is documented in RFC 1075; the current version has changed substantially.
MOBILE-AGENTS.MCAST.NET	*224.0.0.11*	Mobile agents on the local subnet.
DHCP-AGENTS.MCAST.NET	*224.0.0.12*	This multicast group allows a client to locate a Dynamic Host Configuration Protocol (DHCP) server or relay agent on the local subnet.
PIM-ROUTERS.MCAST.NET	*224.0.0.13*	All Protocol Independent Multicasting (PIM) routers on this subnet.

Domain name	IP address	Purpose
RSVP-ENCAPSULATION.MCAST.NET	*224.0.0.14*	RSVP encapsulation on this subnet. RSVP stands for Resource reSerVation setup Protocol, an effort to allow people to reserve a guaranteed amount of Internet bandwidth in advance for an event.
NTP.MCAST.NET	*224.0.1.1*	The Network Time Protocol.
SGI-DOG.MCAST.NET	*224.0.1.2*	Silicon Graphics Dogfight game.
NSS.MCAST.NET	*224.0.1.6*	The Name Service Server.
AUDIONEWS.MCAST.NET	*224.0.1.7*	Audio news multicast.
SUB-NIS.MCAST.NET	*224.0.1.8*	Sun's NIS+ Information Service.
MTP.MCAST.NET	*224.0.1.9*	The Multicast Transport Protocol.
IETF-1-LOW-AUDIO.MCAST.NET	*224.0.1.10*	Channel 1 of low-quality audio from IETF meetings.
IETF-1-AUDIO.MCAST.NET	*224.0.1.11*	Channel 1 of high-quality audio from IETF meetings.
IETF-1-VIDEO.MCAST.NET	*224.0.1.12*	Channel 1 of video from IETF meetings.
IETF-2-LOW-AUDIO.MCAST.NET	*224.0.1.13*	Channel 2 of low-quality audio from IETF meetings.
IETF-2-AUDIO.MCAST.NET	*224.0.1.14*	Channel 2 of high-quality audio from IETF meetings.
IETF-2-VIDEO.MCAST.NET	*224.0.1.15*	Channel 2 of video from IETF meetings.
MUSIC-SERVICE.MCAST.NET	*224.0.1.16*	Music service.

Domain name	IP address	Purpose
SEANET-TELEMETRY.MCAST.NET	*224.0.1.17*	Telemetry data for the U.S. Navy's SeaNet Project to extend the Internet to vessels at sea. See http://web.nps.navy.mil/~seanet/Distlearn/cover.htm.
SEANET-IMAGE.MCAST.NET	*224.0.1.18*	SeaNet images.
MLOADD.MCAST.NET	*224.0.1.19*	MLOADD measures the traffic load through one or more network interfaces over a number of seconds. Multicasting is used to communicate between the different interfaces being measured.
EXPERIMENT.MCAST.NET	*224.0.1.20*	Experiments that do not go beyond the local subnet.
XINGTV.MCAST.NET	*224.0.1.23*	XING Technology's Streamworks TV multicast.
MICROSOFT.MCAST.NET	*224.0.1.24*	Used by Windows Internet Name Service (WINS) servers to locate one another.
MTRACE.MCAST.NET	*224.0.1.32*	A multicast version of traceroute.
JINI-ANNOUNCEMENT.MCAST.NET	*224.0.1.84*	JINI announcements.
JINI-REQUEST.MCAST.NET	*224.0.1.85*	JINI requests.
	224.2.0.0-224.2.255.255	The Multicast Backbone on the Internet (MBONE) addresses are reserved for multimedia conference calls, i.e., audio, video, whiteboard, and shared web browsing between many people.
	224.2.2.2	Port 9,875 on this address is used to broadcast the currently available MBONE programming. You can look at this with the X Window utility sdr or the Windows/Unix multikit program.

Domain name	IP address	Purpose
	239.0.0.0-239.255.255.255	Administrative scope, in contrast to TTL scope, uses different ranges of multicast addresses to constrain multicast traffic to a particular region or group of routers. For example, the IP addresses from 239.178.0.0 to 239.178.255.255 might be an administrative scope for the state of New York. Data addressed to one of those addresses would not be forwarded outside of New York. The idea is to allow the possible group membership to be established in advance without relying on less-than-reliable TTL values.

The MBONE (or Multicast Backbone on the Internet) is the range of Class D addresses beginning with 224.2. that are used for audio and video broadcasts over the Internet. The word MBONE is sometimes used less restrictively (and less accurately) to mean the portion of the Internet that understands how to route Class D addressed packets.

14.1.2. Clients and Servers

When a host wants to send data to a multicast group, it puts that data in multicast datagrams, which are nothing more than UDP datagrams addressed to a multicast group. Most multicast data is audio or video or both. These sorts of data tend to be relatively large and relatively robust against data loss. If a few pixels or even a whole frame of video is lost in transit, the signal isn't blurred beyond recognition. Therefore, multicast data is sent via UDP, which, though unreliable, can be as much as three times faster than data sent via connection-oriented TCP. (If you think about it, multicast over TCP would be next to impossible. TCP requires hosts to acknowledge that they have received packets; handling acknowledgments in a multicast situation would be a nightmare.) If you're developing a multicast application that can't tolerate data loss, it's your responsibility to determine whether data was damaged in transit and how to handle missing data. For example, if you are building a distributed cache system, you might simply decide to leave any files that don't arrive intact out of the cache.

Earlier, I said that from an application programmer's standpoint, the primary difference between multicasting and using regular UDP sockets is that you have to worry about the TTL value. This is a single byte in the IP header that takes values from to 255; it is interpreted roughly as the number of routers through which a packet can pass before it is discarded.

Each time the packet passes through a router, its TTL field is decremented by at least one; some routers may decrement the TTL by two or more. When the TTL reaches zero, the packet is discarded. The TTL field was originally designed to prevent routing loops by guaranteeing that all packets would eventually be discarded; it prevents misconfigured routers from sending packets back and forth to each other indefinitely. In IP multicasting, the TTL limits the multicast geographically. For example, a TTL value of 16 limits the packet to the local area, generally one organization or perhaps an organization and its immediate upstream and downstream neighbors. A TTL of 127, however, sends the packet around the world. Intermediate values are also possible. However, there is no precise way to map TTLs to geographical distance. Generally, the farther away a site is, the more routers a packet has to pass through before reaching it. Packets with small TTL values won't travel as far as packets with large TTL values. Table 14-2 provides some rough estimates relating TTL values to geographical reach. Packets addressed to a multicast group from 224.0.0.0 to 224.0.0.255 are never forwarded beyond the local subnet, regardless of the TTL values used.

Table 14-2. Estimated TTL values for datagrams originating in the continental United States

Destinations	TTL value
The local host	0
The local subnet	1
The local campus—that is, the same side of the nearest Internet router—but on possibly different LANs	16
High-bandwidth sites in the same country, generally those fairly close to the backbone	32
All sites in the same country	48
All sites on the same continent	64
High-bandwidth sites worldwide	128
All sites worldwide	255

Once the data has been stuffed into one or more datagrams, the sending host launches the datagrams onto the Internet. This is just like sending regular (unicast) UDP data. The sending host begins by transmitting a multicast datagram to the local network. This packet immediately reaches all members of the multicast group in the same subnet. If the Time-To-Live field of the packet is greater than 1, multicast routers on the local network forward the packet to other networks that have members of the destination group. When the packet arrives at one of the final destinations, the multicast router on the foreign network transmits the packet to each host it serves that is a member of the multicast group. If necessary, the multicast router also retransmits the packet to the next routers in the paths between the current router and all its eventual destinations.

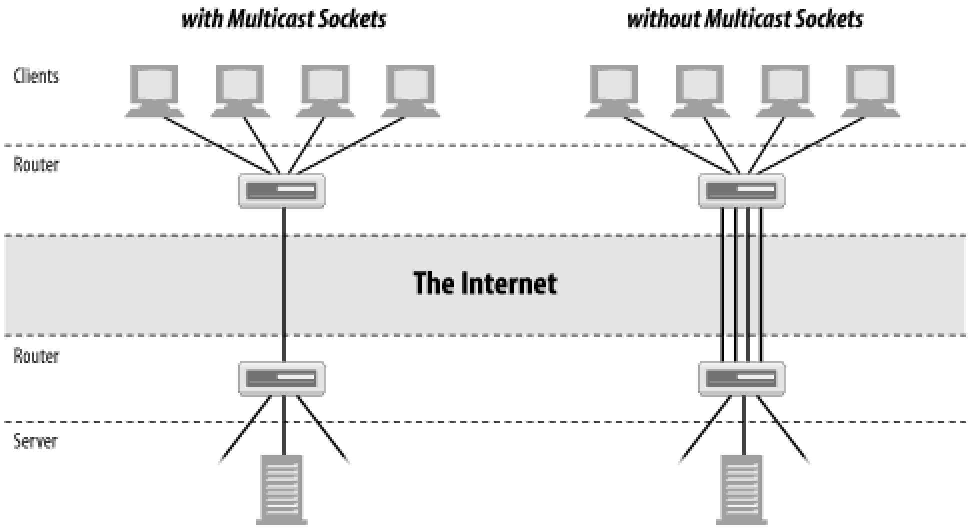
When data arrives at a host in a multicast group, the host receives it as it receives any other UDP datagram—even though the packet's destination address doesn't match the receiving host. The host recognizes that the datagram is intended for it because it belongs to the multicast group to which the datagram is addressed, much as most of us accept mail addressed to "Occupant," even though none of us are named Mr. or Ms. Occupant. The receiving host must be listening on the proper port, ready to process the datagram when it arrives.

14.1.3. Routers and Routing

Figure 14-3 shows one of the simplest possible multicast configurations: a single server sending the same data to four clients served by the same router. A multicast socket sends one stream of data over the Internet to the clients' router; the router duplicates the stream and sends it to each of the clients. Without multicast sockets, the server would have to send four separate but identical streams of data to the router, which would route each stream to a client. Using the same stream to send the same data to multiple clients significantly reduces the bandwidth required on the Internet backbone.

Of course, real-world routes can be much more complex, involving multiple hierarchies of redundant routers. However, the goal of multicast sockets is simple: no matter how complex the network, the same data should never be sent more than once over any given network segment. Fortunately, you don't need to worry about routing issues. Just create a `MulticastSocket`, have the socket join a multicast group, and stuff the address of the multicast group in the `DatagramPacket` you want to send. The routers and the `MulticastSocket` class take care of the rest.

Figure 14-3. With and without multicast sockets



The biggest restriction on multicasting is the availability of special multicast routers (mrouters). Mrouters are reconfigured Internet routers or workstations that support the IP multicast extensions. Many consumer-oriented ISPs quite deliberately do not enable multicasting in their routers. In 2004, it is still possible to find hosts between which no multicast route exists (i.e., there is no route between the hosts that travels exclusively over mrouters).

To send and receive multicast data beyond the local subnet, you need a multicast router. Check with your network administrator to see whether your routers support multicasting. You can also try pinging *all-routers.mcast.net*. If any router responds, then your network is hooked up to a multicast router:

```
% ping all-routers.mcast.net
all-routers.mcast.net is alive
```

This still may not allow you to send to or receive from every multicast-capable host on the Internet. For your packets to reach any given host, there must be a path of multicast-capable routers between your host and the remote host. Alternately, some sites may be connected by special multicast tunnel software that transmits multicast data over unicast UDP that all routers understand. If you have trouble getting the examples in this chapter to produce the expected results, check with your local network administrator or ISP to see whether multicasting is actually supported by your routers.

14.2. Working with Multicast Sockets

Enough theory. In Java, you multicast data using the `java.net.MulticastSocket` class, a subclass of `java.net.DatagramSocket`:

```
public class MulticastSocket extends DatagramSocket
```

As you would expect, `MulticastSocket`'s behavior is very similar to `DatagramSocket`'s: you put your data in `DatagramPacket` objects that you send and receive with the `MulticastSocket`. Therefore, I won't repeat the basics; this discussion assumes that you already know how to work with datagrams. If you're jumping around in this book rather than reading it cover to cover, now might be a good time to go back and read Chapter 13 on UDP.

To receive data that is being multicast from a remote site, first create a `MulticastSocket` with the `MulticastSocket( )` constructor. Next, join a multicast group using the `MulticastSocket`'s `joinGroup( )` method. This signals the routers in the path between you and the server to start sending data your way and tells the local host that it should pass you IP packets addressed to the multicast group.

Once you've joined the multicast group, you receive UDP data just as you would with a `DatagramSocket`. That is, you create a `DatagramPacket` with a byte array that serves as a buffer for data and enter a loop in which you receive the data by calling the `receive ( )` method inherited from the `DatagramSocket` class. When you no longer want to receive data, leave the multicast group by invoking the socket's `leaveGroup()` method. You can then close the socket with the `close( )` method inherited from `DatagramSocket`.

Sending data to a multicast address is similar to sending UDP data to a unicast address. You do not need to join a multicast group to send data to it. You create a new `DatagramPacket`, stuff the data and the address of the multicast group into the packet, and pass it to the `send( )` method. The one difference is that you must explicitly specify the packet's TTL value.

There is one caveat to all this: multicast sockets are a security hole big enough to drive a small truck through. Consequently, untrusted code running under the control of a `SecurityManager` is not allowed to do anything involving multicast sockets. Remotely loaded code is normally allowed to send datagrams to or receive datagrams from the host it was downloaded from. However, multicast sockets don't allow this sort of restriction to be placed on the packets they send or receive. Once you send data to a multicast socket, you have very limited and unreliable control over which hosts receive that data. Consequently, most environments that execute remote code take the conservative approach of disallowing all multicasting.

14.2.1. The Constructors

The constructors are simple. Each one calls the equivalent constructor in the `DatagramSocket` superclass.

14.2.1.1. public MulticastSocket( ) throws SocketException

This constructor creates a socket that is bound to an anonymous port (i.e., an unused port assigned by the system). It is useful for clients (i.e., programs that initiate a data transfer) because they don't need to use a well-known port: the recipient replies to the port contained in the packet. If you need to know the port number, look it up with the `getLocalPort( )` method inherited from `DatagramSocket`. This constructor throws a `SocketException` if the `Socket` can't be created. For example:

```
try {
  MulticastSocket ms = new MulticastSocket( );
  // send some datagrams...
}
catch (SocketException se) {
  System.err.println(se);
}
```

14.2.1.2. public MulticastSocket(int port) throws SocketException

This constructor creates a socket that receives datagrams on a well-known port. The `port` argument specifies the port on which this socket listens for datagrams. As with regular TCP and UDP unicast sockets, on a Unix system a program needs to be run with root privileges in order to create a `MulticastSocket` on a port numbered from 1 to 1,023.

This constructor throws a `SocketException` if the `Socket` can't be created. A `Socket` can't be created if you don't have sufficient privileges to bind to the port or if the port you're trying to bind to is already in use. Note that since a multicast socket is a datagram socket as far as the operating system is concerned, a `MulticastSocket` cannot occupy a port already occupied by a `DatagramSocket`, and vice versa. For example, this code fragment opens a multicast socket on port 4,000:

```
try {
  MulticastSocket ms = new MulticastSocket(4000);
  // receive incoming datagrams...
}
catch (SocketException ex) {
  System.err.println(ex);
}
```

14.2.1.3. public MulticastSocket(SocketAddress bindAddress) throws IOException // Java 1.4

Starting in Java 1.4, you can create a `MulticastSocket` using a `SocketAddress` object. If the `SocketAddress` is bound to a port, then this is pretty much the same as the previous constructor. For example, this code fragment also opens a `MulticastSocket` on port 4000 that listens on all network interfaces and addresses:

```
try {
  SocketAddress address = new InetSocketAddress(4000);
  MulticastSocket ms = new MulticastSocket(address);
  // receive incoming datagrams...
}
catch (SocketException ex) {
  System.err.println(ex);
}
```

However, the `SocketAddress` can also be bound to a specific network interface on the local host, rather than listening on all network interfaces. For example, this code fragment also opens a `MulticastSocket` on port 4000 that only listens to packets arriving on 192.168.254.32:

```
try {
  SocketAddress address = new InetSocketAddress("192.168.254.32", 4000);
  MulticastSocket ms = new MulticastSocket(address);
  // receive incoming datagrams...
}
catch (SocketException ex) {
  System.err.println(ex);
}
```

Finally, you can pass null to this constructor to create an unbound socket, which would later be connected with the `bind()` method. This is useful when setting socket options that can only be set before the socket is bound. For example, this code fragment creates a multicast socket with SO_REUSEADDR disabled (that option is normally enabled by default for multicast sockets):

```
try {
  MulticastSocket ms = new MulticastSocket(null);
  ms.setReuseAddress(false);
  SocketAddress address = new InetSocketAddress(4000);
```

```
    ms.bind(address);
    // receive incoming datagrams...
  }
  catch (SocketException ex) {
    System.err.println(ex);
  }
```

14.2.2. Communicating with a Multicast Group

Once a `MulticastSocket` has been created, it can perform four key operations:

1. Join a multicast group.
2. Send data to the members of the group.
3. Receive data from the group.
4. Leave the multicast group.

The `MulticastSocket` class has methods for operations 1, 2, and 4. No new method is required to receive data. The `receive( )` method of the superclass, `DatagramSocket`, suffices for this task. You can perform these operations in any order, with the exception that you must join a group before you can receive data from it (or, for that matter, leave it). You do not need to join a group to send data to it, and the sending and receiving of data may be freely interwoven.

14.2.2.1. public void joinGroup(InetAddress address) throws IOException

To receive data from a `MulticastSocket`, you must first join a multicast group. To join a group, pass an `InetAddress` object for the multicast group to the `joinGroup( )` method. If you successfully join the group, you'll receive any datagrams intended for that group. Once you've joined a multicast group, you receive datagrams exactly as you receive unicast datagrams, as shown in the previous chapter. That is, you set up a `DatagramPacket` as a buffer and pass it into this socket's `receive( )` method. For example:

```
  try {
    MulticastSocket ms = new MulticastSocket(4000);
    InetAddress ia = InetAddress.getByName("224.2.2.2");
    ms.joinGroup(ia);
    byte[] buffer = new byte[8192];
    while (true) {
      DatagramPacket dp = new DatagramPacket(buffer, buffer.length);
      ms.receive(dp);
      String s = new String(dp.getData( ), "8859_1");
      System.out.println(s);
    }
  }
  catch (IOException ex) {
    System.err.println(ex);
  }
```

If the address that you try to join is not a multicast address (that is, it is not between 224.0.0.0 and 239.255.255.255), the `joinGroup( )` method throws an `IOException`.

A single `MulticastSocket` can join multiple multicast groups. Information about membership in multicast groups is stored in multicast routers, not in the object. In this case, you'd use the address stored in the incoming datagram to determine which address a packet was intended for.

Multiple multicast sockets on the same machine and even in the same Java program can all join the same group. If so, they'll all receive all data addressed to that group that arrives at the local host.

14.2.2.2. public void joinGroup(SocketAddress address, NetworkInterface interface) throws IOException // Java 1.4

Java 1.4 adds this overloaded variant of `joinGroup()` that allows you to join a multicast group only on a specified local network interface. A proxy server or firewall might use this to specify that it will accept multicast data from the interface connected to the LAN, but not the interface connected to the global Internet, for instance.

For example, this code fragment attempts to join the group with IP address 224.2.2.2 on the network interface named "eth0", if such an interface exists. If no such interface exists, then it joins on all available network interfaces:

```
MulticastSocket ms = new MulticastSocket(4000);
SocketAddress group = new InetSocketAddress("224.2.2.2", 40);
NetworkInterface ni = NetworkInterface .getByName("eth0");
if (ni != null) {
  ms.joinGroup(group, ni);
}
else {
  ms.joinGroup(group);
}
```

Other than the extra argument specifying the network interface to listen from, this behaves pretty much like the single argument `joinGroup( )` method. For instance, passing a `SocketAddress` object that does not represent a multicast group as the first argument throws an `IOException`.

14.2.2.3. public void leaveGroup(InetAddress address) throws IOException

The `leaveGroup( )` method signals that you no longer want to receive datagrams from the specified multicast group. A signal is sent to the appropriate multicast router, telling it to stop sending you datagrams. If the address you try to leave is not a multicast address (that is, if it is not between 224.0.0.0 and 239.255.255.255), the method throws an `IOException`. However, no exception occurs if you leave a multicast group you never joined.

14.2.2.4. public void leaveGroup(SocketAddress multicastAddress, NetworkInterface interface) throws IOException // Java 1.4

Java 1.4 also allows you to specify that you no longer want to receive datagrams on one particular network interface. Perhaps you do wish to continue receiving datagrams on other network interfaces. For instance, you could join on all interfaces, and then leave just one. To be honest, this is a bit of a stretch. This method was probably included mostly for symmetry with `joinGroup( ).`

14.2.2.5. public void send(DatagramPacket packet, byte ttl) throws IOException

Sending data with a `MulticastSocket` is similar to sending data with a `DatagramSocket`. Stuff your data into a `DatagramPacket` object and send it off using the `send( )` method inherited from `DatagramSocket`:

```
public void send(DatagramPacket p) throws IOException
```

The data is sent to every host that belongs to the multicast group to which the packet is addressed. For example:

```
try {
  InetAddress ia = InetAddress.getByName("experiment.mcast.net");
  byte[] data = "Here's some multicast data\r\n".getBytes( );
  int port = 4000;
  DatagramPacket dp = new DatagramPacket(data, data.length, ia, port);
  MulticastSocket ms = new MulticastSocket( );
  ms.send(dp);
}
catch (IOException ex) {
  System.err.println(ex);
}
```

However, the `MulticastSocket` class adds an overloaded variant of the `send( )` method that lets you provide a value for the Time-To-Live field `ttl`. By default, the `send( )` method uses a TTL of 1; that is, packets don't travel outside the local subnet. However, you can change this setting for an individual packet by passing an integer from 0 to 255 as the second argument to the `send()` method. For example:

```
DatagramPacket dp = new DatagramPacket(data, data.length, ia, port);
MulticastSocket ms = new MulticastSocket( );
ms.send(dp, 64);
```

14.2.2.6. public void setInterface(InetAddress address) throws SocketException

On a multihomed host, the `setInterface()` method chooses the network interface used for multicast sending and receiving. `setInterface( )` throws a `SocketException` if the `InetAddress` argument is not the address of a network interface on the local machine. It is unclear why the network interface is immutably set in the constructor for unicast `Socket` and `DatagramSocket` objects but is variable and set with a separate method for `MulticastSocket` objects. To be safe, set the interface immediately after constructing a `MulticastSocket` and don't change it thereafter. Here's how you might use `setInterface( )`:

```
MulticastSocket ms;
InetAddress ia;
try {
  ia = InetAddress.getByName("www.ibiblio.org");
  ms = new MulticastSocket(2048);
  ms.setInterface(ia);
  // send and receive data...
}
catch (UnknownHostException ue) {
  System.err.println(ue);
}
catch (SocketException se) {
  System.err.println(se);
}
```

14.2.2.7. public InetAddress getInterface( ) throws SocketException

If you need to know the address of the interface the socket is bound to, call `getInterface ()`. It isn't clear why this method would throw an exception; in any case, you must be prepared for it. For example:

```
try {
  MulticastSocket ms = new MulticastSocket(2048);
  InetAddress ia = ms.getInterface( );
}
catch (SocketException se) {
  System.err.println(ue);
}
```

14.2.2.8. public void setNetworkInterface(NetworkInterface interface) throws SocketException // Java 1.4

The `setNetworkInterface()` method serves the same purpose as the `setInterface ( )` method; that is, it chooses the network interface used for multicast sending and receiving. However, it does so based on the local name of a network interface such as "eth0" (as encapsulated in a `NetworkInterface` object) rather than on the IP address bound to that network interface (as encapsulated in an `InetAddress` object). `setNetworkInterface()` throws a `SocketException` if the `NetworkInterface` passed as an argument is not a network interface on the local machine.

14.2.2.9. public NetworkInterface getNetworkInterface( ) throws SocketException // Java 1.4

The `getNetworkInterface()` method returns a `NetworkInterface` object representing the network interface on which this `MulticastSocket` is listening for data. If no network interface has been explicitly set in the constructor or with `setNetworkInterface( )`, it returns a placeholder object with the address "0.0.0.0" and the index -1. For example, this code fragment prints the network interface used by a socket:

```
NetworkInterface intf = ms.getNetworkInterface( );
System.out.println(intf.getName( ));
```

14.2.2.10. public void setTimeToLive(int ttl) throws IOException // Java 1.2

The `setTimeToLive()` method sets the default TTL value used for packets sent from the socket using the `send(Datagrampacket dp)` method inherited from `DatagramSocket` (as opposed to the `send(Datagrampacket dp, byte ttl)` method in `MulticastSocket`). This method is only available in Java 1.2 and later. In Java 1.1, you have to use the `setTTL( )` method instead:

```
public void setTTL(byte ttl) throws IOException
```

The `setTTL( )` method is deprecated in Java 2 and later because it only allows TTL values from 1 to 127 rather than the full range from 1 to 255.

14.2.2.11. public int getTimeToLive( ) throws IOException // Java 1.2

The `getTimeToLive()` method returns the default TTL value of the `MulticastSocket`. It's not needed very much. This method is also available only in Java 1.2 and later. In Java 1.1, you have to use the `getTTL( )` method instead:

```
public byte getTTL( ) throws IOException
```

The `getTTL( )` method is deprecated in Java 1.2 and later because it doesn't properly handle TTLs greater than 127—it truncates them to 127. The `getTimeToLive( )` method can handle the full range from 1 to 255 without truncation because it returns an `int` instead of a `byte`.

14.2.2.12. public void setLoopbackMode(boolean disable) throws SocketException // Java 1.4

Whether or not a host receives the multicast packets it sends is platform-dependent—that is, whether or not they loop back. Passing `true` to `setLoopback()` indicates you don't want to receive the packets you send. Passing `false` indicates you do want to receive the packets you send. However, this is only a hint. Implementations are not required to do as you request.

14.2.2.13. public boolean getLoopbackMode( ) throws SocketException // Java 1.4

Because loopback mode is only a hint that may not be followed on all systems, it's important to check what the loopback mode is if you're both sending and receiving packets. The `getLoopbackMode()` method returns `true` if packets are not looped back and `false` if they are. (This feels backwards to me. I suspect this method was written by a programmer following the ill-advised convention that defaults should always be true.)

If the system is looping packets back and you don't want it to, you'll need to recognize the packets somehow and discard them. If the system is not looping the packets back and you do want it to, store copies of the packets you send and inject them into your internal data structures manually at the same time you send them. You can ask for the behavior you want with `setLoopback( )`, but you can't count on it.

14.3. Two Simple Examples

Most multicast servers are indiscriminate about who they will talk to. Therefore, it's easy to join a group and watch the data that's being sent to it. Example 14-1 is a `MulticastSniffer` class that reads the name of a multicast group from the command line, constructs an `InetAddress` from that hostname, and creates a `MulticastSocket`, which attempts to join the multicast group at that hostname. If the attempt succeeds, `MulticastSniffer` receives datagrams from the socket and prints their contents on `System.out`. This program is useful primarily to verify that you are receiving multicast data at a particular host. Most multicast data is binary and won't be intelligible when printed as ASCII.

Example 14-1. Multicast sniffer

```
import java.net.*;
import java.io.*;

public class MulticastSniffer {

  public static void main(String[] args) {

    InetAddress group = null;
    int port = 0;

    // read the address from the command line
    try {
      group = InetAddress.getByName(args[0]);
      port = Integer.parseInt(args[1]);
    }  // end try
    catch (Exception ex) {
      // ArrayIndexOutOfBoundsException, NumberFormatException,
      // or UnknownHostException
      System.err.println(
       "Usage: java MulticastSniffer multicast_address port");
      System.exit(1);
    }

    MulticastSocket ms = null;

    try {
      ms = new MulticastSocket(port);
      ms.joinGroup(group);

      byte[] buffer = new byte[8192];
      while (true) {
        DatagramPacket dp = new DatagramPacket(buffer, buffer.length);
        ms.receive(dp);
```

```
            String s = new String(dp.getData( ));
            System.out.println(s);
          }
        }
        catch (IOException ex) {
          System.err.println(ex);
        }
        finally {
          if (ms != null) {
            try {
              ms.leaveGroup(group);
              ms.close( );
            }
            catch (IOException ex) {}
          }
        }

      }

    }
```

The program begins by reading the name and port of the multicast group from the first command-line argument. Next, it creates a new `MulticastSocket` `ms` on the specified port. This socket joins the multicast group at the specified `InetAddress`. Then it enters a loop in which it waits for packets to arrive. As each packet arrives, the program reads its data, converts the data to an ISO Latin-1 `String`, and prints it on `System.out`. Finally, when the user interrupts the program or an exception is thrown, the socket leaves the group and closes itself.

MBONE session announcements are broadcast to the multicast group *sap.mcast.net* on port 9,875. You can use this program to listen to those announcements. Generally, if you're connected to the MBONE (not all sites are), you should see a site announcement pop through within the first minute or two. In fact, you'll probably see a lot more. I collected about a megabyte and a half of announcements within the first couple of minutes I had this program running. I show only the first two here:

```
% java MulticastSniffer sap.mcast.net 9875
úv=0
o=ellery 3132060082 3138107776 IN IP4 131.182.10.250
s=NASA TV - Broadcast from NASA HQ
i=NASA TV Multicasting from NASA HQ
u=http://www.nasa.gov/ntv
e=Ellery.Coleman@hq.nasa.gov     (Ellery D. Coleman)
p=+202 651 8512
t=3138107776 3153918976
r=15811200 15811200 0
a=recvonly
a=tool:FVC.COM I-Caster V3.1/3101, Windows95/NT
a=cat:Corporate/Events
m=audio 23748 RTP/AVP 0
c=IN IP4 224.2.203.38/127
```

```
m=video 60068 RTP/AVP 31
c=IN IP4 224.2.203.37/127
b=AS:380
a=framerate:9
a=quality:8
a=grayed:0
4 224.2.255.115/15
.77/25
4 RTP wbbesteffort
c=IN IP4 224.2.224.41/25

‰Â¡_v=0
o=dax 3137417804 3141052115 IN IP4 horla.enst.fr
s=VREng UDP (Virtual Reality Engine)
i=Virtual Reality Engine: Distributed Interactive 3D Multicast
navigator in Virtual Worlds. For more information and downloading, see
URL: http://www.infres.enst.fr/net/vreng/.
u=http://www.infres.enst.fr/net/vreng/
e=Philippe Dax (ENST) <dax@inf.enst.fr>
p=Philippe Dax (ENST) +33 (0) 145817648
t=0 0
a=tool:sdr v2.9
a=type:test
m=dis 62239 RTP 99
c=IN IP4 224.2.199.133/127
/3
m=mdesk 64538 RTP/AVP mdesk
c=IN IP4 224.2.160.68/3
e please stop your receiving programs and the stream should stop from
coming to you.
u=http://tv.funet.fi/ohjelmat/index.html
e=Harri Salminen <mice-nsc@nic.funet.fi>
p=Harri Salminen +358 400 358 502
t=3085239600 3299658800
a=tool:CDT mAnnouncer 1.1.2
a=type:broadcast
m=audio 4004 RTP/AVP 0
c=IN IP4 239.239.239.239/40
a=ptime:40
m=video 6006 RTP/AVP 31
c=IN IP4 239.239.239.239/40
m=whiteboard 4206 udp wb
c=IN IP4 224.239.239.245/48
```

MBONE session announcements are not pure ASCII text. In particular, they contain a lot of embedded nulls as well as various characters with their high bit set. Consequently, I've had to take a few liberties with the output to print it in this book. To really handle MBONE session announcements, you'd have to parse the relevant ASCII text out of the binary format and display that. Peter Parnes has written a Java program called mSD that does exactly that. If you're interested, you can find it at http://www.cdt.luth.se/~peppar/progs/mSD/. However, since this is a book about network programming and not parsing binary file formats, we'll leave the example here and move on to sending multicast data. Example 14-2 is a `MulticastSender` class that sends data read from the command line to a multicast group. It's fairly simple, overall.

Example 14-2. MulticastSender

```
import java.net.*;
import java.io.*;

public class MulticastSender {

  public static void main(String[] args) {

    InetAddress ia = null;
    int port = 0;
    byte ttl = (byte) 1;

    // read the address from the command line
    try {
      ia = InetAddress.getByName(args[0]);
      port = Integer.parseInt(args[1]);
      if (args.length > 2) ttl = (byte) Integer.parseInt(args[2]);
    }
    catch (Exception ex)  {
      System.err.println(ex);
      System.err.println(
       "Usage: java MulticastSender multicast_address port ttl");
      System.exit(1);
    }

    byte[] data = "Here's some multicast data\r\n".getBytes( );
    DatagramPacket dp = new DatagramPacket(data, data.length, ia, port);

    try {
      MulticastSocket ms = new MulticastSocket( );
      ms.joinGroup(ia);
      for (int i = 1; i < 10; i++) {
        ms.send(dp, ttl);
      }
      ms.leaveGroup(ia);
      ms.close( );
    }
    catch (SocketException ex) {
      System.err.println(ex);
    }
    catch (IOException ex) {
      System.err.println(ex);
    }

  }

}
```

Example 14-2 reads the address of a multicast group, a port number, and an optional TTL from the command line. It then stuffs the string `"Here's some multicast data\r\n"` into the byte array `data` using the `getBytes()` method of `java.lang.String`, and places this array in the `DatagramPacket` `dp`. Next, it constructs the `MulticastSocket` `ms`, which joins the group `ia`. Once it has joined the group, `ms` sends the datagram packet `dp` to the group `ia` 10 times. The TTL value is set to one to make sure that this data doesn't go beyond the local subnet. Having sent the data, `ms` leaves the group and closes itself.

Run `MulticastSniffer` on one machine in your local subnet. Listen to the group *all-systems.mcast.net* on port 4,000, like this:

```
% java MulticastSniffer all-systems.mcast.net 4000
```

Next, send data to that group by running `MulticastSender` on another machine in your local subnet. You can also run it in a different window on the same machine, although that option is not as exciting. However, you must start running the `MulticastSniffer` before you start running the `MulticastSender`. Send to the group *all-systems.mcast.net* on port 4,000, like this:

```
% java MulticastSender all-systems.mcast.net 4000
```

Back on the first machine, you should see this output:

```
Here's some multicast data
Here's some multicast data
Here's some multicast data
Here's some multicast data
Here's some multicast data
Here's some multicast data
Here's some multicast data
Here's some multicast data
Here's some multicast data
```

For this to work beyond the local subnet, the two subnets must each have multicast routers.