Programmer to Programmer™

BEGINNING

Java Networking

Networking in JDK 1.3 and 1.4, Java I/O, Threads, Java security model, JCA, JAAS, JCE and JSSE, TCP, UDP, Multicasting, Java URL handler architecture, Serialization, RMI, CORBA, Servlets, JavaMail and JMS

Chad Darby, John Griffin, Pascal de Haan, Peter den Haan,
Alexander V. Konstantinou, Sing Li, Sean MacLean,
Glenn E. Mitchell II, Joel Peach, Peter Wansch, William Wright

Summary of Contents

11

Multicasting

This chapter will introduce the IP multicast protocol that is used to for one-to-many transmission on the Internet. IP multicast is a network-layer protocol, unlike TCP and UDP that are transport layer protocols. The chapter will:

- Outline how the multicast protocol works
- Explain how to control the scope of multicast transmissions
- Introduce the Java support for UDP/IP multicast programming

Because IP multicast is a network service, it requires support by the operating system and the local network. Platform-specific pointers are provided and a small Java program is introduced that can be used to check local multicast availability. The chapter concludes with an in-depth multicast example that builds a graphical group chat application.

Why Use Multicasting?

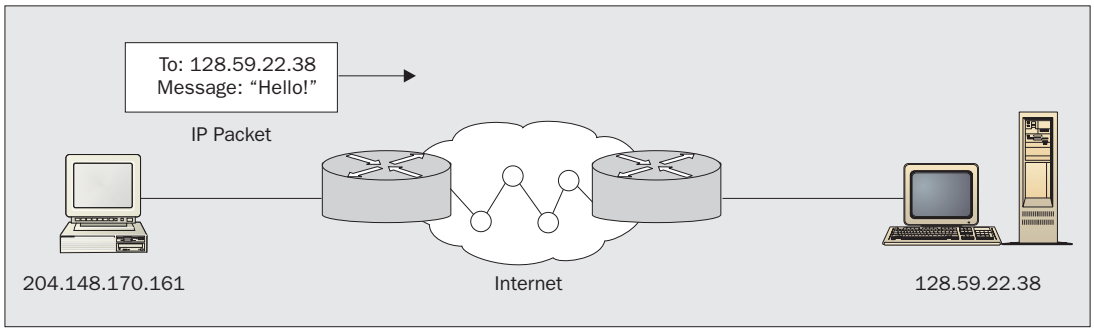
In the previous two chapters, we have demonstrated the use of TCP and UDP for effecting communication between two parties. This model of communication is described as **one-to-one** (also known as **unicast**) because one specific Internet host is the recipient of every communication. Although the one-to-one model is a good fit for many applications, it is not appropriate for others.

Consider a live Internet video service. In the one-to-one model, every video client will have to establish a separate stream with the video server. All these streams will be carrying the same payload information and will just differ in their destination address. As a result, multiple copies of the same information are likely to traverse the same Internet link, en route to different destinations. It would make more sense to send a single copy of the sampled video and deliver this to all receivers, duplicating it as necessary in the course of the routing process. This is the **one-to-many** transmission model (also known as **multicast**).

Multicast is becoming increasingly important in a variety of applications – besides its obvious utility for live streaming applications, multicast is being used to create dynamic systems, such as Jini federations. The Jini technology from Sun Microsystems uses IP multicast to support automated service discovery. Other companies are experimenting with multicast delivered software updates. Unlike the unicast Internet protocols, IP multicast is still not universally available. It is expected that multicast will increasingly be used within individual organization networks. Unfortunately, it is not clear when and if IP multicast will become available as a global Internet-wide service.

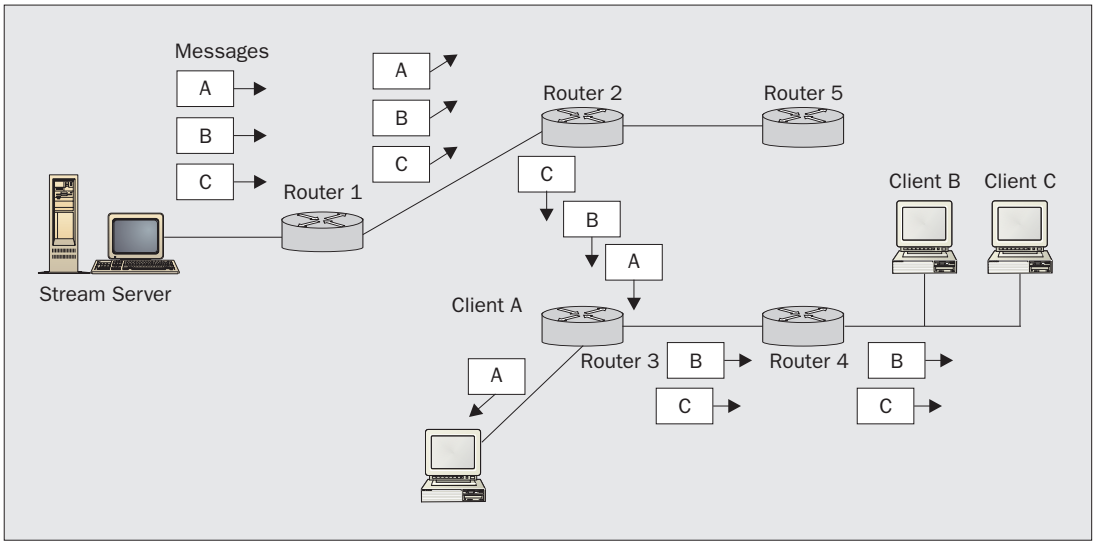
The One-to-One Model – Unicast

So far, we have identified one type of end-to-end communication, called **one-to-one**. In one-to-one communications, an IP datagram originates in a single Internet host, and is marked for delivery to an IP address identifying the unique location of another Internet host. The diagram below illustrates this one-to-one model:



The one-to-one model is appropriate for many types of applications. For example, remote access involves a single user connecting to a specific remote service. Electronic mail is also typically a one-to-one service, unless the message is addressed to multiple users. Web access and file transfer, two essentially similar services, can also be considered as one-to-one applications under certain conditions. If the content transferred, such as a web page or a file, is customized for each user then the end-to-end model is appropriate. Even if the content is not customized for each user, if multiple requests are not likely to coincide in time, then individual transmission may be appropriate.

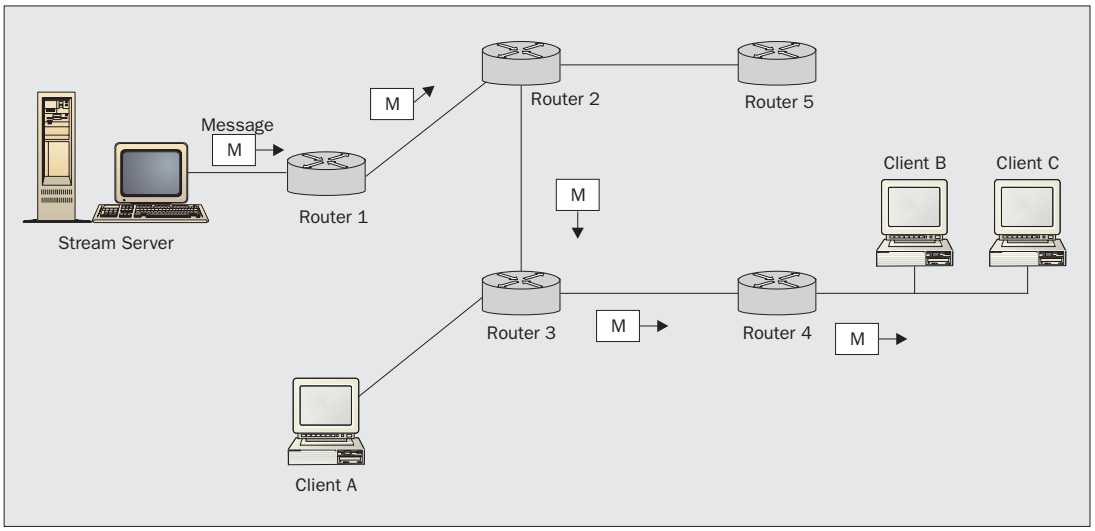
As we have already seen the one-to-one model is not appropriate for many types of application. The problem of multiple instances of identical data traveling over the same connection is illustrated below:



The stream server, on the left, creates a byte-stream by sampling some video source. Consider the case when three clients need to receive the video stream. In the classic IP model, the stream server must transmit three copies of the sampled data over a UDP/IP datagram or a TCP/IP segment – these copies are shown as packets A, B, C in the diagram. The server network's access router (Router-1) will then route these packets to the Internet Service Provider's network. In this example, the packets continue traveling in the same path after the second routing step (Router-2). In the third step, the packet destined for client A is delivered on the local network via Router-3, and the remaining two copies for clients B and C are forwarded to Router-4 for local delivery. Router-5 is not in the path to any of the destinations and hence no datagrams are transmitted on its links.

The One-to-Many Model – Multicasting

How can the one-to-one model be improved? The answer is to use an alternative network model supporting **one-to-many** transmission. In this model, the sender generates a single datagram destined for multiple receivers. In the one-to-many model, it is the responsibility of the network layer to create copies of transmitted datagrams as needed, usually with the goal of optimizing bandwidth usage. Returning to the previous example, the stream server would generate a single packet containing the sampled data. Unlike the one-to-one case, the TCP protocol cannot be used for transport, since there is no single peer with which to negotiate sequencing and retransmission. Hence, some type of connectionless protocol must be used, such as UDP/IP. The one-to many model is illustrated in the diagram below:



A single copy is transmitted, from Router-1 to Router-2 then to Router-3. Router-3 notices that the packet must be sent in two outgoing links, towards client A, as well as towards clients B, C. The solution is to copy the packet into both outgoing links. One copy is then delivered to client A, and another via Router-4 to clients B, and C. Because Router-5 is not on the path to any of the destinations, no traffic crosses its links.

In the above example, data originated from a single source and was sent to multiple destinations. In other types of applications, such as group conferencing, it is desirable to support **many-to-many** transmission. This model is similar to the one-to-many model with the difference that any receiver may also transmit data to all other receivers.

Implementing Multicast Applications

The idea of transmitting a single packet to multiple receivers may sound simple and intuitive, but its implementation involves several challenges:

- **Addressing**: Internet addresses identify a unique location in the network (see Chapter 8). When multiple hosts must be identified, a meaningful way for addressing the group must be devised. A simple solution would be to include all the destination addresses in each packet. However, this solution would not scale as the number of hosts increase. Not only would the data portion grow disproportionately smaller, but also at some point, a single packet would not even be able to fit all addresses (due to the limited IP packet size)! An alternative solution would be to use a unique **group** identifier to represent the multiple destination hosts. In such an approach, the network would be responsible for maintaining the mapping between the unique group identifier and the group members. This latter approach is used in Internet one-to-many communications. A certain range of IP addresses has been reserved to identify groups of Internet hosts, instead of the location of a single Internet host.
- **Membership**: The use of group identifiers for addressing creates the need for a network-layer membership mechanism. In one-to-one transmissions, it is the responsibility of the sender to identify the location of the recipient by storing the destination address in the datagram header. In multicast transmissions, the destination address stored in the multicast datagram no longer identifies a location; instead it identifies a group. Therefore, the network layer must maintain knowledge of the group's member addresses. For this purpose, applications must be provided with a protocol for communicating their group join and leave requests to the network layer. Membership is closely tied to routing which is discussed next.
- **Routing**: One approach to handling packets with multiple destinations is to send such packets to every possible recipient. This approach is also known as **broadcasting**. All members in a network, without distinction, receive broadcasted packets. Besides the obvious security implications, the primary limitation of broadcasting is scalability. Broadcasting is typically applied only in local area networks because they are usually small, and many times the physical medium itself is shared, so every frame (link-layer packet) is broadcasted anyway. On the Internet, broadcasting would mean that all IP hosts would receive IP packets destined to multiple hosts. Clearly that would not be a scalable solution, as every Internet link would have to handle the world's entire broadcast bandwidth!

  The answer is to provide a **multicast** service. In multicasting, packets destined for multiple addresses are only sent to router nodes that are on the path to one of the target recipients. In a sense, multicast is like a smart broadcast in which only the routers required are involved. How can this be accomplished? The answer is by supporting multicast routing at the network layer. Using the information established by multicast membership protocols, routers can decide how to handle packets destined for multiple hosts.
- **Reliability**: The TCP protocol supports reliable data transmission between two Internet hosts. This reliable service is built on top of an unreliable network by requiring data recipients to acknowledge packets received (positive acknowledgement). Both sides in the reliable connection must maintain state for unacknowledged packets to permit retransmission in case of loss detection. This mechanism could ostensibly be used in one-to-many transmissions. However, this approach is almost never used due to its scalability limitations. Positive acknowledgement would nullify many of the advantages of multicast since they would create a flood of acknowledgement packets towards the sender, and would require per-receiver maintenance of state in the sender. Scalable reliable multicasting is a difficult problem that is still the subject of research. The current Internet multicasting protocol provides an unreliable service.

IP Multicasting

The current Internet multicasting protocol was introduced in 1991, and published as IETF RFC 966 & RFC 988. These were superceded by RFC 1112, which is available at http://www.ietf.org/rfc/rfc1112.txt). The IP multicast protocol takes a minimalist approach to multicast service. In terms of the four challenges previously described (addressing, membership, routing, reliability) the IETF IP multicast design can be described as using:

- **IPv4 class D group addressing**: Host groups are identified by class D IP addresses starting with binary "1110", in the range 224.0.0.0–239.255.255.255. This range was allocated in the original IP specification for this use. Unlike unicast addresses, which are uniquely assigned to each organization, most IPv4 multicast group addresses are not allocated to particular organizations (some exceptions to this will be covered later in the chapter). Any Internet node can attempt to send as well as receive datagrams to any group address using any IP-based protocol. The IP multicast layer does not provide any address allocation support; it is the responsibility of multicast applications to coordinate temporary allocation of group addresses.
- **Dynamic membership**: Hosts may join and leave groups at any time. Group membership is not restricted in location or size. Hosts may belong to more than one group at a given time. Group membership is not required for datagram transmission. That is, a host can send datagrams to a group without being a member.
- **Multiple routing protocols**: The original IP multicast specification focused on Internet host multicast support. The specification did not state how routers communicate to maintain the multicast service at the core of the network. Instead, the specification defined the protocol used for communication between a multicast-enabled host and its immediately neighboring router. As a result, multiple standard multicast routing protocols have been defined for router-to-router multicast support.
- **Unreliable delivery**: Multicast packets are delivered with the same "best-effort" reliability as regular unicast IP packets. A multicast datagram may be delivered to all, some, or none of the members in a group. Multicast datagrams may arrive in different ordering at each receiver, and receivers may see a different subset of the datagrams sent due to loss. No standard reliable multicast transport protocol was defined.

Although the IP multicast protocol has been around for about 10 years, it has yet to become universally available. Some people may even choose to characterize it as a failure. To be fair, the problem of providing a scalable universal multicast service is significantly harder than that of providing unicast service. There have been several significant roadblocks to the wide adoption of the IP multicast protocol:

- **Deployment**: IP multicast is a network-layer service. In contrast, most Internet protocols are transport-or application-layer services. An important difference is that network-layer services must be deployed on *all* Internet nodes, while transport-and application-layer services need only be deployed on *some* of the edge nodes. For example, when the WWW was first conceived it only required the installation of an HTTP server on a host, and some HTTP/HTML clients on one or more other hosts. This is simple, whereas to get all core Internet service providers to install experimental software on their prized routers is a lot more difficult. In fact it might not even be possible as most routers are dedicated proprietary devices and not programmable by general users.

- **Scalability**: the original multicast routing protocols were based on a flat network topology. Because IP multicast group addresses were allocated in an arbitrary manner, the size of the multicast routing tables increased proportionally with the number of groups. A new generation of hierarchical multicast routing protocols was developed to address this issue but its deployment has been slow.
- **Unpredictability**: the engineering of IP networks is a difficult task due to the lack of traffic reservation or admission policy mechanisms, which would allow you to control the flow of data. Every Internet host is typically free to pump as much data as its Internet access link will allow. This creates a nightmare situation for Internet Service Providers (ISP) who would like to assure customers that their access is not severely affected by the behavior of other customers. For this reason, ISPs usually overprovision their backbone networks, and then limit the bandwidth in the access points to prevent any single source from flooding the backbone.

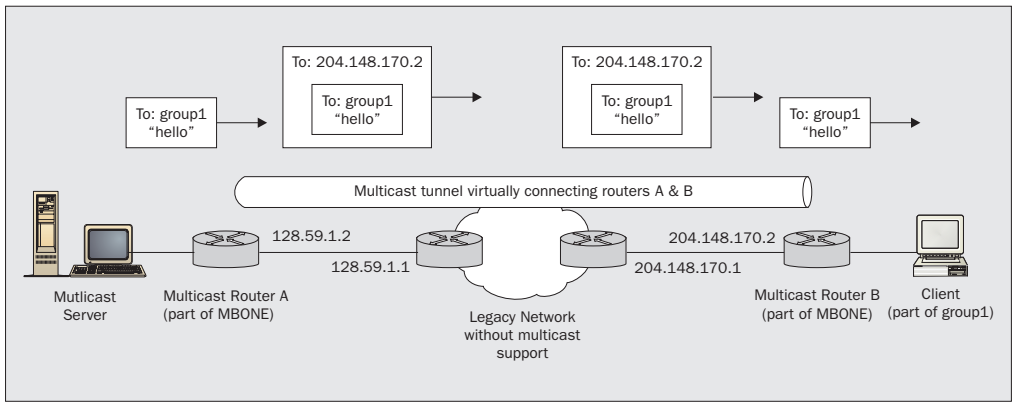
Perversely, even though multicast was created to improve network efficiency, its deployment on ISP backbones is currently deemed as too risky. This mostly stems from the scalability problems, and the immaturity of cross-domain multicast protocols (discussed later). Some Internet Service Providers have addressed this problem by creating a separate backbone dedicated to multicast transmission.

Due to the above-mentioned problems, many networks participating in the Internet do not carry external IP multicast traffic. Even though they may not be routing IP multicast externally, most of these networks support multicast internally. There are many advantages to using IP multicast even within a single administrative network. Besides the typical multicast streaming applications, such as internal conferencing and shared-whiteboard/design applications, multicast can also be useful for resource discovery. Consider a mobile work force that moves frequently within a single administrative network (for example, a corporate network). When users move to a new location, they would ideally like to discover and use the local resources, such as printers, scanners, etc. If all printers would listen to a given multicast group, then the user could multicast a request query in order to discover the locally available resources. The Jini technology from Sun Microsystems generalizes this idea to provide dynamic discovery of network services using multicast.

Multicast Backbone (MBONE)

To combat the seemingly insurmountable deployment problem, the IETF created a semi-permanent IP multicast network called the **MBONE** (**Multicast Backbone**). The MBONE is a **virtual network** that uses the transmission facilities of the Internet to create **virtual links** between nodes. Multicast datagrams traverse virtual links by being encapsulated into unicast datagrams addressed to the virtual link end-point. For example, consider a university whose routers support IP multicasting. We will assume that the university is connected to the Internet via an Internet Service Provider that is not part of the MBONE. The university can join the MBONE by creating a virtual link with another network that is part of the MBONE. The end-points of this virtual link will be routers that are configured to wrap every multicast datagram in a unicast datagram addressed to the other side of the link.

An example of how such a virtual topology operates is shown in the diagram below. Two networks supporting multicast would like to exchange multicast traffic. Unfortunately, they are connected via one or more legacy routers that do not support multicast routing. The solution is to establish a virtual link, also known as a **tunnel**, between the multicast routers A and B. Multicast datagrams that must be transmitted over this virtual link are encapsulated in a unicast datagram that is addressed to the IP address of the other router.



For example, the multicast server on the left sends a datagram to all members of a particular group (called group1 for clarity; in reality it will be a class-D IP address). Upon receipt of the multicast datagram, router A consults its multicast routing table and finds out that the packet must be sent to router B. Because router B is connected over a tunnel, router A creates a new IP datagram destined for router B with the complete datagram packet (IP header + data) as payload. The legacy network then routes the unicast packet just like any other and attempts to deliver it to router B. Upon receipt, router B looks at the packet and figures out that it is tunnel traffic, so it extracts its payload and uses the payload's IP header to transmit the multicast datagram as if directly received from router A.

One of the biggest problems in maintaining a large virtual network, such as the MBONE, is ensuring that the virtual links (tunnels) are created efficiently. Ideally, multiple tunnels should never be layered over a single physical link. In practice, assuring such efficient allocation is difficult, because the physical topology of the Internet is not fixed, and new networks are constantly added to the MBONE. A mapping of the virtual network that made sense at creation time may no longer be efficient due to changes in physical topology.

Although the MBONE has increased in size over the years, its members continue to be mostly research and education organizations. Commercial organizations have not adopted the MBONE largely due to the lack of support by major Internet Service Providers.

> **It is important to distinguish the IP multicast protocols from the multicast backbone (MBONE). Due to the aforementioned problems, IP multicast is not universally routed on the Internet today. Many corporate networks support IP multicast internally, but do not connect to the global MBONE network. Therefore developers cannot assume that any two Internet hosts can communicate using IP multicast.**

IP Multicast Addressing

Unicast IP addresses in the range `0.0.0.0 - 223.255.255.255` uniquely identify the location of an Internet host. In order to support scalable routing, these addresses are allocated in blocks characterized by a shared prefix, as described in Chapter 8. Each block may be further subdivided, for example, an ISP will divide its blocks amongst its customers, who may further subdivide them internally. Using this hierarchical approach, backbone routers can aggregate routing information using shortest prefix summarization, with each organization free to manage its own assigned addresses. If a host is assigned an address outside the allotted blocks, then that host will not be reachable from outside the organization's network.

Unlike unicast IPv4 addresses, IPv4 multicast addresses are not assigned hierarchically. Every multicast address in the range `224.0.0.0 – 239.255.255.255` can be used as a group identifier by any multicast-enabled application anywhere in the world. Clearly, this creates a problem in that no application can be assured exclusive access to an IP multicast group. Because the IP multicast model permits all hosts to join a multicast group, and even allows hosts to send multicast IP packets without joining, every multicast transport or application-layer protocol must be able to detect "foreign" traffic. That is, multicast applications must have a mechanism for identifying traffic conforming to their protocol and session. The main disadvantage of having two multicast applications using the same group address is unnecessary propagation of multicast traffic.

Session Discovery

One problem caused by this arbitrary multicast group address assignment is session discovery. If clients are to discover the current multicast group of a well-known service, a shared directory mechanism must be used. In the unicast Internet, this role is played by the Domain Name System service (DNS). The DNS system, however, is not well suited for storing highly dynamic information, due to its caching architecture. There are two IP multicast address assignment mechanisms in use on the MBONE today static assignments and dynamic reservation.

Static Assignments

The Internet Corporation for Assigned Names and Numbers (ICANN) will assign permanent multicast addresses to well-known protocols, long-lived multicast sessions and large companies. For example, static addresses have been assigned to Sun Microsystems, one of which was used for the Jini multicast discovery protocol. Jini clients can therefore be hard-coded to use that permanently assigned address. It is the responsibility of other multicast applications not to use permanently assigned addresses.

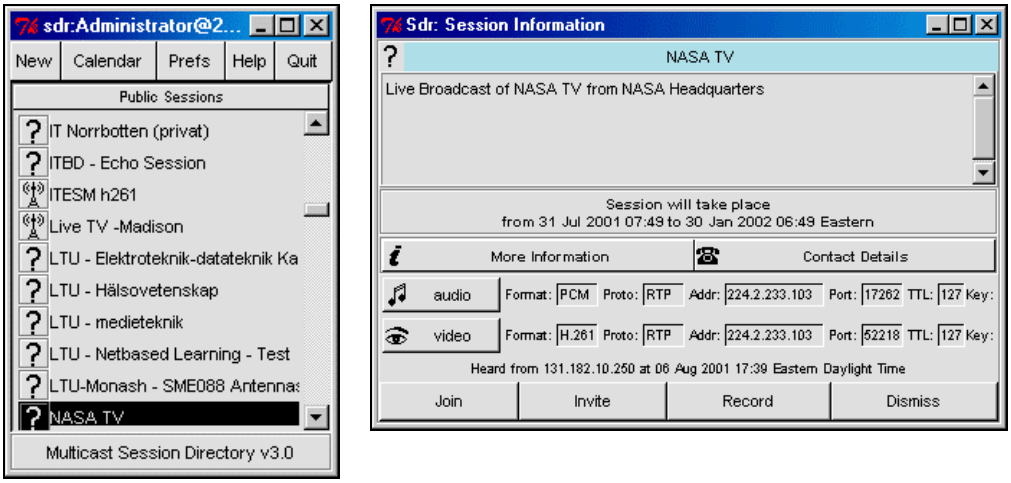
Dynamic Reservation

For short-lived multicast sessions, a dynamic reservation mechanism was devised. Reservation announcements are multicast to a well-known group address and are formatted using the **Session Announcement Protocol** (**SAP**) (experimental IETF RFC 2974). Announcements can be made for immediate or future multicast group use. Before using a multicast address, applications are supposed to listen to a specific multicast channel for group reservation announcements. This wait period is typically at least 10 minutes long. At the end of this period the application picks a multicast group address that has not been reserved and starts sending its own periodic reservation messages. Clearly, this is not a fail-proof mechanism. Group address conflicts may occur if two applications simultaneously pick the same address, or if reservation announcements are lost. Since group address conflicts only have a performance implication, they can be acceptable for a short period until they can be detected and the applications attempt to switch to using another address.

Session Discovery Tools

The reservation multicast group can also be used as a broadcast schedule guide. In addition to the multicast address used, the reservation protocol carries a user description of the session, as well as information about the session protocol and payload encoding. A user tool called SDR can be used to display scheduled multicast sessions to users. The SDR tool is freely available for several platforms (including Windows, and Linux) at http://www-mice.cs.ucl.ac.uk/multimedia/software/sdr/. At the time of writing, the latest release was version 3; two downloads are available, one for IPv4 and another for IPv6 networks.

A sample SDRv3 session listing on an IPv4 MBONE-connected host is shown below. The main window on the left shows a number of advertised sessions by title. The session window on the right shows details about the "NASA TV" session, including the scheduled transmission time, multicast group address and ports (one for audio and another for video), as well as the audio/video encoding of the transmission. Note that if your host is not on an MBONE-connected network, only local session announcements will be received.



IP Multicast Scoping

Some IP multicast applications may desire to limit the scope of their multicast transmissions. Possible reasons include:

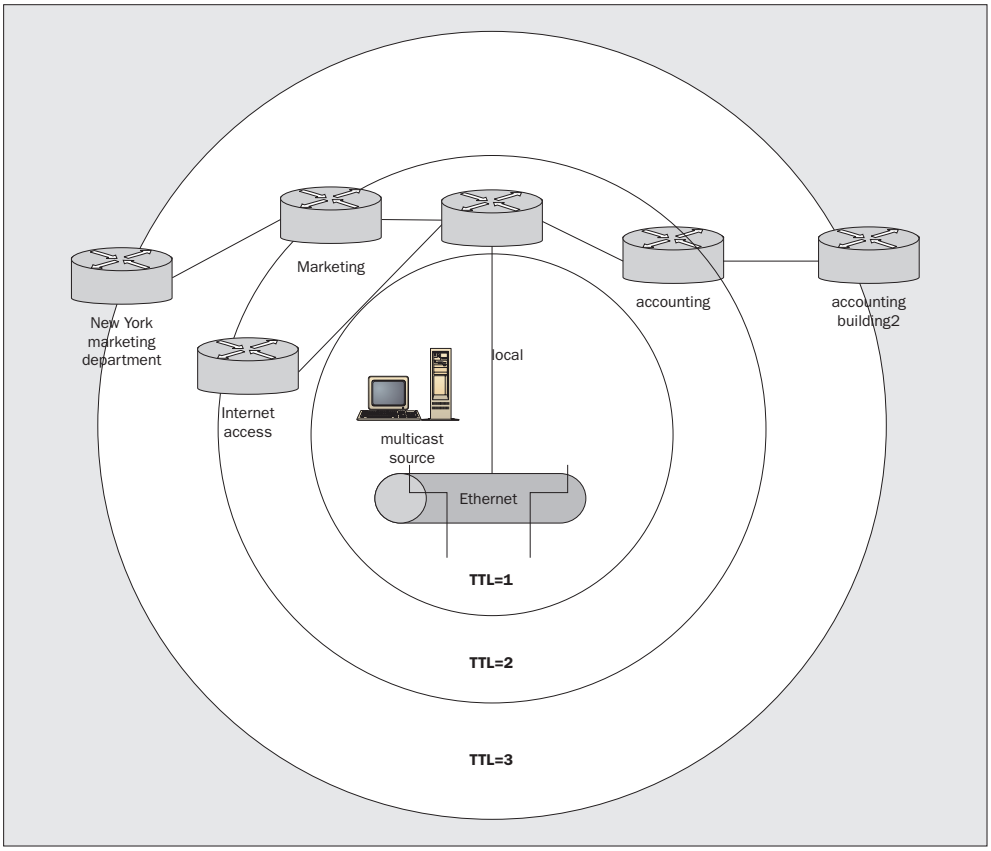
- Security and privacy
- Limiting network resources usage (bandwidth/routing load)
- Avoiding address conflicts

It should be noted that depending on any type of multicast scoping for security or privacy is probably not a very good idea. Currently, there are two mechanisms for scoping multicast transmissions: Time-to-Live scoping (TTL), and administrative scoping.

Time-to-Live Scoping

In TTL scoping the IP time-to-live field is used as the radius of a multicast transmission in terms of the number of hops. Every time an IP packet is forwarded, its TTL value is decreased by one. If the TTL value reaches zero before the destination has been reached, then that packet is discarded. This mechanism was originally introduced to limit the effect of IP routing loops. These occur when packets travel in continuous loops due to redundant links in the routing table.

If you wanted to multicast only on the local area network, then multicast packets could be transmitted with a TTL of one. This value would guarantee that the packet could be picked up by other hosts, but would be discarded by the local router. Using the TTL field to control the extent of transmission propagation can be a difficult task. For example, if one wanted to multicast to all of an organization, then you would need to know the exact topology of the network to determine the TTL. Moreover, unless the server was situated in the middle of the network, then no single TTL would exist that could reach all of the organization's nodes without leaving the network. For example, in the diagram below, a multicast source is connected to a corporate network via a departmental router. The local router is directly connected to the marketing and accounting department routers, as well as the Internet access router. Some departments are further connected to other routers representing internal divisions.



If the multicast source broadcasts a packet with TTL=1, then only the local area network hosts will receive it, that is, it's received by the router, but then discarded because the TTL will have hit zero before getting out "the other side". A packet with TTL=2 will be forwarded to subscribed receivers that are one-hop off. In this case, only routers are one-hop off, so this TTL is not really useful. A TTL=3 will send the packet to all subscribed receivers that are two-hops off. This will include hosts connected to the marketing and accounting departments, but also hosts connected to the Internet access router (oops). In order to reach all organization hosts, in this example, a TTL of 4 is needed, but that will also mean that packets will leak into the Internet.

In order to reduce the guesswork involved in TTL scoping, certain arbitrary values were assigned for specific scopes (called TTL-threshold). The table below shows these assignments:

TTL	Scoping Restriction
= 0	Local host
= 1	Local subnet
<= 32	Site (organization)
<= 64	Region
<= 128	Continent
> 128 (<= 255)	Unrestricted

These assignments are not foolproof as they depend on the correct configuration of the routers involved. For example, if an organization does not configure its Internet access routers to drop multicast packets with TTL less than 32, then these packets will be transmitted on the Internet (up to the specified TTL radius).

Administrative Scoping

The second scoping mechanism is more recent and involves the use of specially assigned IP multicast groups. The range of IP multicast addresses has been further subdivided into blocks. Use of the addresses from each block has implications for the scope of the multicast transmission. This type of scoping is called Administrative Scoped IP Multicast and is defined in IETF RFC 2365 (http://www.ietf.org/rfc/rfc2365.txt). Although the use of administrative scoping may appear to be similar to the use of TTL scoping, it is preferred because TTL scoping interacts negatively with the IP multicast routing protocols. Basically, dropped packets due to TTL expiration interfere with the multicast tree maintenance. To understand why this is the case requires the introduction of several complex topics that are inappropriate here.

IP Range	Scoping Restriction
224.0.0.0 – 224.0.0.255	Link-scope addresses assigned to routing protocols. Addresses in this range are not routed and hence remain local.
224.0.1.0 – 224.0.1.255	Individual addresses assigned to application protocols.
224.0.2.0 – 238.255.255.255	Assigned to companies and applications (global).
239.192.0.0 – 239.251.255.255	Organization-local scope. Not routed outside an organization and therefore can be privately used.
239.255.0.0 –239.255.255.255	Local-scope. Not routed outside the local scope (further limited from organization-local).

IP Multicast Routing

There are two components to IP multicast routing. The first component is the protocol used by the edge hosts to request multicast group join and leave requests from their first-hop routers. The second component is the IP multicast routing protocol used between routers to efficiently route multicast IP datagrams. The first component is standard for all multicast hosts and is known as the Internet Group Management Protocol (IGMP). The second component is non-standard and is chosen by network administrations.

IP hosts report group membership to immediately neighboring multicast routers using the Internet Group Management Protocol (IGMP). IGMP is defined as part of the "Host Extensions for IP Multicasting" IETF RFC 1112 (http://www.ietf.org/rfc/rfc1112.txt). All Internet hosts that support IP multicast must implement the IGMP protocol. IGMP messages are encapsulated in IP datagrams with a specified binary format. Operationally, there are two types of IGMP messages:

- **Host membership query**: Periodically sent by every multicast router to solicit host membership reports (usually no more than once a minute). The query is addressed to the special all-local-hosts multicast group (`224.0.0.1`) with an IP TTL of `1`. Every multicast host is supposed to listen to the `224.0.0.1` group for multicast router membership queries.
- **Host membership report**: Hosts send IGMP membership reports in response to a membership query. A separate report is sent for every group for which a local process has requested a join. Each IGMP report is addressed to the joined group and has an IP TTL of `1`. In order to prevent an "implosion" of reports, hosts pick a random delay in sending their report. If another host has sent a report for a given group during this delay interval, the report is not sent. This is acceptable since the multicast routers do not need to know how many hosts have joined a group, just that there is at least one.

Version 1 of the IGMP protocol did not provide an explicit group leave message. Multicast routers that have not received a membership report for a specific group after a certain number of queries will simply drop the group from their join-list. Version 2 (IETF RFC 2236 – http://www.ietf.org/rfc/rfc2236.txt) of the IGMP protocol added an explicit leave message in order to support faster pruning of the multicast tree.

Multicast Routing Protocols

The second component to multicast routing is the protocol used between multicast routers. There are multiple multicast routing protocols in use today. They can be roughly divided into two categories: **dense mode** and **sparse mode** multicast protocols. Dense mode protocols perform better when used in a topology densely populated with group members. Their main disadvantage is that they maintain state information for each source at every router in the network. Sparse mode protocols provide better scalability. However, sparse protocols depend on a rendezvous point for synchronization, they typically have a single point of failure. Sparse protocols may also generate non-optimal paths in the multicast tree when routing from the source to the rendezvous point and then to the receivers.

The choice of IP multicast routing protocols is not in the user's control and therefore their coverage is an advanced topic not suitable for this book. The original MBONE routing protocol was called DVMRP (Distance Vector Multicast Routing Protocol) and was a dense mode routing protocol. Newer protocols include PIM (Protocol Independent Multicast) which is a protocol supporting a dense mode, as well as a sparse mode and CBT (Core Based Trees) which is a sparse mode protocol.

Multicast Port Addressing

The Internet Protocol does not support any application-level addressing. IP datagrams may only be addressed to a particular IP address representing a location or a multicast group. For this reason, Internet transport layer protocols such as TCP and UDP add an additional application addressing service using 16-bit port identifiers. TCP cannot be used for multicast transmission since the protocol is built on the assumption that there are only two parties to a streaming connection. Moreover, as discussed in Chapter 10, applications typically do not use raw IP for transmission because it does not detect payload corruption, does not provide application-level addressing and many times is restricted by operating systems due to security considerations. For these reasons, Internet multicast applications typically use the UDP protocol to send datagrams to IP multicast group addresses. In many cases, the UDP datagrams sent contain RTP (Real Time Protocol) formatted data. The RTP protocol was presented at the end of Chapter 10 and is used to provide sequencing and encoding information for real-time streaming.

Multicast UDP applications use port numbers in a different way than unicast UDP applications. In traditional UDP unicast transmissions, port numbers are used to de-multiplex incoming packets for delivery to different applications. In multicast transmission this behavior is not as useful, since different applications typically use different multicast group addresses to avoid sharing traffic. Another important difference is that the semantics of multicast allow for multiple applications to send and receive multicast packets using the same group and port number. Therefore, multiple applications may use the same UDP port for sending and receiving multicast traffic on the same host/address! In other words, multicast UDP ports no longer uniquely identify a single application as a recipient, in the same way that IP multicast group addresses no longer identify a single Internet node as a recipient.

> **It is possible to concurrently bind multiple times to the same multicast port, either in the same or in different processes. The behavior of the multicast protocol stack will be to deliver a copy of each received datagram to all bound sockets. Conversely, every socket may be used to send multicast datagrams marked as originating from the shared port.**

If multicast group addresses are typically used by a single session, why should multicast applications need to use port numbers at all? The main reason is that the single session may transmit datagrams using different protocols and/or different encodings. For example, earlier we showed a multicast session announcement for "NASA TV" that advertised multicast transmission using group address `224.2.233.103` with a PCM-encoded audio stream sent using the RTP protocol to port 17262 and an H.261-encoded video stream sent using the RTP protocol to port `52218` (PCM and H.261 are content encoding standards like WAV and MPEG). When a multicast host joins the `224.2.233.103` group, it will receive both the video and the audio traffic. The different port numbers will enable the client to potentially use different programs for receiving the audio and the video stream. In many cases, a source will transmit the audio and video streams on separate group addresses to enable receivers to join to either or both. For example, a host connected via a slow link may chose to only join the audio multicast group.

Java IP Multicast Programming

IP multicast is a network layer service, and as such, can potentially be used with different transport layer protocols. Of the two popular Internet transport layer protocols, however, only the connectionless UDP protocol is applicable, since the TCP protocol is connection-oriented and synchronizes state between two participants. As a result, nearly all multicast traffic today uses UDP as its transport layer encapsulation. Even real-time streaming applications using the Real Time Protocol (RTP) send data as RTP over UDP over multicast-IP.

The Java network libraries support multicast transmission through instances of the `java.net.MulticastSocket` class. The `MulticastSocket` class extends the UDP `java.net.DatagramSocket` class with multicast-specific operations, such as joining and leaving a multicast group. In the subsequent discussion, it will be assumed that readers are familiar with the coverage of the `DatagramPacket`, and `DatagramSocket` classes in the preceding chapter.

java.net.MulticastSocket

A `MulticastSocket` is a UDP socket with extra support for the IGMP multicast group management protocol. As a UDP extension, multicast sockets instances are also associated with UDP port numbers, but with the different semantics discussed earlier. Most importantly, multiple `MulticastSocket` instances may be bound to the same port!

Constructors

Two constructors may be used in creating `MulticastSocket` instances:

```
MulticastSocket() throws IOException

MulticastSocket(int port) throws IOException
```

The first is a no-argument constructor that creates a multicast socket bound to an arbitrary port. After construction, the local port number used may be retrieved by invoking the `getLocalPort()` method. Similarly to the `DatagramSocket` no-argument constructor, the `java.io.IOException` is used to signal an error in allocating resources for the multicast socket. A `SecurityException` will be thrown if the `checkListen() SocketPermission` has not been granted (see Chapter 7). As a runtime exception, the `SecurityException` is not listed in the method's signature but has been added to the text for clarity purposes.

The second constructor is more commonly used and allows invokers to specify the port number used for binding. Although only one port number is typically used per multicast group address, its use must be consistent across all members. Currently, Java does not support arrival notification for all multicast packets destined for a particular group irrespective of the port number. The same exceptions are thrown as in the previous constructor. Unlike the `DatagramSocket(int)` constructor, an `IOException` will not be thrown if this or another local process has already bound to the socket, because the multicast protocol permits such behavior.

> **By default, `MulticastSocket` instances are set to transmit datagrams with a time-to-live value of `1`. This means that multicast transmissions will be constrained to the local area network. The default TTL value can be changed by invoking the `setTimeToLive()` method as described later in this section.**

Methods

The `MulticastSocket` class supports IGMP-based IP multicast membership via two methods. The `joinGroup()` method may be used to request subscription for the specified multicast group from the first-hop (local) router. The method takes an IP address represented as a `java.net.InetAddress` instance as its only argument. The address must represent a multicast address, as defined by the `InetAddress isMulticastAddress()` method, otherwise a `SocketException` is thrown. Due to the nature of the IGMP protocol, the method returns immediately without indication of remote success or failure. Only local failures can be detected, and are signaled by an `IOException` if the IGMP request could not be sent at all, or a `SecurityException` if the "accept, connect" `SocketPermission` permissions have not been granted (as a runtime exception it is not listed in the method signature but shown here for clarity). If the local host does not support IP multicast on any of its interfaces, a `SocketException` may be thrown (platform dependent; discussed later in the multicast host configuration section).

```
void joinGroup(InetAddress mcastaddr) throws IOException
```

After the `joinGroup()` method has been invoked, the `MulticastSocket` will periodically transmit IGMP membership reports, and respond to IGMP membership queries as prescribed by the IGMP protocol. In general, the only way to discover that a multicast router is available is to note the receipt of an IGMP membership query request, however, the Java multicast support does not expose this information. As a result, all Java applications can do is to invoke the `joinGroup()` method and hope that packets start arriving at some point! Note that applications are not required to join a multicast group in order to send multicast datagrams destined for that group. However, in order to receive multicast datagrams addressed to a group the `joinGroup()` method must be invoked.

> **`MulticastSocket` instances differ from `DatagramSocket` instances in that datagrams addressed to multicast group addresses for which the `joinGroup()` method has been invoked are also received by the transmitter. This creates a loop-back effect. The "Host Extensions for IP Multicasting" IETF RFC 1112 requires implementations to provide a method to disable local delivery of multicast datagrams; however Java versions prior to JDK1.4 did not comply with this requirement. Therefore, Multicast Java applications executed on earlier versions (including JDK 1.3) cannot disable the loop-back effect and must be prepared to receive their own multicast datagrams.**

Applications can request to leave a group by invoking the `leaveGroup()` method. If the application has not previously successfully invoked `joinGroup()` for the same IP group address, the `java.net.BindException` will be thrown. If the multicast socket has been closed the method will throw a `SocketException`. The `SocketException` may also be thrown in response to some underlying communications error. A `SecurityException` will be thrown if the "accept, connect" `SocketPermission` permissions have not been granted. This security check is made to prevent an internal denial-of-service-type attack.

```
void leaveGroup(InetAddress mcastaddr) throws IOException
```

As we explained in Chapter 8, a host may be attached to the Internet through multiple network interfaces, each with its own IP address. By default, the `MulticastSocket` constructor binds the multicast socket to all available interfaces. The effect of this behavior is that the decision of which interface to use is left to the operating system. In nearly all cases, this is the most appropriate approach. In some rare cases a Java multicast program may need to have control over interface selection. For example, a Java program running on a host connected to a corporate network as well as the public Internet may want to restrict its multicast operations to the corporate network interface. The `setInterface()` method takes as a single argument the IP address of one of the local host's interfaces. A `SocketException` will be thrown if the address does not belong to a local interface, or an error occurred while configuring the socket. Invoking the method with the special "0.0.0.0" IP address will restore the default behavior of binding to all sockets. As we discussed in Chapter 8, there was no method prior to JDK1.4 for discovering all interfaces in a host. Therefore, this information would need to be supplied externally. For this reason, the `setInterface()` method was rarely invoked.

```
void setInterface(InetAddress inf) throws SocketException;
```

The `getInterface()` method returns the address of the interface set by the last `setInterface()` invocation, or the special address "`0.0.0.0`" marking the fact that the multicast socket is not bound to a particular interface. The method may throw a `SocketException` if the multicast socket's configuration cannot be read.

```
InetAddress getInterface() throws SocketException;
```

The IP time-to-live (TTL) field plays a special role in multicast transmission. As was explained earlier in this chapter, the TTL field was originally used to scope the range of multicast transmissions. For example, a TTL of `1` restricts the multicast packet to the local-area network (defined in terms of the LAN broadcast radius), while a TTL less than `32` should restrict the transmission to the local-organization network (depending on router configuration).

The `MulticastSocket` class adds a method for setting and a method for retrieving the default TTL of a multicast socket. The `setTimeToLive()` method deprecates the old `setTTL()`and takes a TTL value as its only argument. The method will throw an `IOException` if the TTL value is invalid, that is not in the range [`0..255`], or an error occurred while configuring the multicast socket. The `getTimeToLive()` method returns the current TTL value for the socket. The method may also throw an `IOException` if the socket's configuration could not be read.

```
void setTimeToLive(int ttl) throws IOException;
int getTimeToLive() throws IOException;
```

In addition to the `send()` method inherited from `DatagramSocket`, instances of `MulticastSocket` can be asked to transmit a `DatagramPacket` with a specific time-to-live value. This allows applications to temporarily override the current socket TTL for one particular transmission. As in the `DatagramSocket send()` method, a transmission error is signaled by an `IOException` and may be caused by misconfiguration of the `DatagramPacket` (missing address/port), an invalid TTL, or some internal protocol stack error. A `SecurityException` will be thrown if the code-base has not been granted the "accept, connect" permissions. The security requirements are more stringent than those for unicast UDP transmissions, where only the "connect" permission is required. The security requirements also apply for the inherited `send(DatagramPacket)` method.

```
void send(DatagramPacket p, int ttl) throws IOException
```

The `MulticastSocket` class inherits all the methods of the `DatagramSocket` class. Readers should refer to Chapter 10 for coverage of these methods.

Multicast Security Permissions

Multicast operations are security sensitive. In a normal denial of service attack, a single destination is flooded with requests from multiple hosts. Using the multicast protocol, a rogue application can abuse network resources in a much more effective way, flooding a large multicast group from a single host. Using the multiplicative effect of the multicast protocol attackers can cause damage that is disproportionate to their Internet access bandwidth. Even permission to join a group without the right to transmit can be dangerous. An attacking application could simply subscribe to all multicast groups possibly completely overwhelming the local network's Internet access link. Another problem with multicast transmission is that the local security manager cannot determine which hosts will receive the transmission. The IP multicast protocol is distributed and hence no one knows the membership of a group. For these reasons, multicast access should be restricted to trusted applications. That is why Java applets are typically not given multicast access.

The Java `SocketPermission` security model defines four types of operations; their mapping to actual `SecurityManager` methods is shown below:

Operations	Method
Resolving	`checkConnect(host, -1)`
Listening	`checkListen(port)`
Connecting	`checkConnect(host, port)`
Accepting	`checkAccept(host, port)`

Note: there is no `checkResolve(host)` method as you might expect; instead the `checkConnect()` method is used with the invalid port number –1 as an argument (don't ask us why!).

The `MulticastSocket` adds an additional check called `checkMulticast()`:

```
public void checkMulticast(InetAddress maddr)
```

The `checkMulticast()` method is invoked when joining or leaving a group, and when sending a packet to a multicast group address (in addition to `checkConnect()`). You would therefore expect that the `SocketPermission` class would support an explicit "multicast" permission; this is however not the case. Instead, the `checkMulticast(InetAddress)` method is translated into two other calls for `checkAccept(InetAddress)` and `checkConnect(InetAddress)`.

Given the lack of an explicit "multicast" `SocketPermission`, how can we restrict applications from using multicast, but still perform unicast operations? The answer is not to grant permissions to multicast class-D IP addresses (range `224.0.0.0-239.255.255.255`). Unfortunately, the current `SocketPermission` class does not permit use of a network mask in the host argument. Therefore, it is not possible to restrict applications using the default security manager. The answer will be to extend the security manager and override the `checkMulticast()` method to provide the missing functionality.

IP Multicast Host Configuration

Today, most popular computing platforms include support for IP multicast. Multicasting is supported by the 32-bit versions of the Microsoft Windows operating systems family as well as most modern UNIX-based operating systems (Solaris, Linux, MacOS X, AIX). Support however does not imply that multicast will work out of the box. In some cases, some extra configuration may be required. A test program is provided at the end of this section to verify your host's IP multicast configuration.

It will be assumed that the network your host is connected to supports IP multicast. If the network does not support multicast there is not much you can do as a user, except contact your local systems administrators. To find out if your network is connected to the MBONE you can either ask your systems administrators, or download and execute the SDR utility presented earlier in this section.

Configuring Microsoft Windows (32-bit)

The Microsoft Windows 32-bit operating systems support multicast transmission over Ethernet and other popular link-layer protocols by default. No special configuration is required for such "real" interfaces. Unfortunately, Microsoft Windows-based operating systems currently do not support IP multicast transmission over the loopback interface. The loopback interface is a virtual interface used to provide local delivery of IP packets. As a result, standalone Microsoft Windows hosts cannot execute multicast applications! A possible work-around is to install a network interface card and configure it for some specific IP address. Note that it is not sufficient to just install a network interface card; it must be configured as well. Mobile users who use the Dynamic Host Configuration Protocol (DHCP) will need to manually enter an IP address when not connected to the network in order to test multicast applications.

Configuring GNU/Linux

The Linux kernel provides optional support for IP multicast. Nearly all current GNU/Linux distributions ship with IP multicast support. Unfortunately, some distributions, such as RedHat, do not enable multicast support in the IP interface by default, and do not include a default multicast route. You can check the configuration of your interfaces by using the `ifconfig` command as shown below. This example was executed on a host running RedHat Linux, with one Ethernet network interface and a loop-back interface. Users need to look for the MULTICAST flag (shown in bold).

```
$ /sbin/ifconfig -a
eth0   Link encap:Ethernet  HWaddr 00:10:4B:C6:D3:39
       inet addr:128.59.22.27  Bcast:128.59.23.255  Mask:255.255.248.0
       UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
       RX packets:95096460 errors:0 dropped:0 overruns:61 frame:0
       TX packets:18386848 errors:0 dropped:0 overruns:0 carrier:0
       collisions:0 txqueuelen:100
       Interrupt:11 Base address:0x1400

lo     Link encap:Local Loopback
       inet addr:127.0.0.1  Mask:255.0.0.0
       UP LOOPBACK RUNNING  MTU:16436  Metric:1
       RX packets:8340660 errors:0 dropped:0 overruns:0 frame:0
       TX packets:8340660 errors:0 dropped:0 overruns:0 carrier:0
       collisions:0 txqueuelen:0
```

If the MULTICAST flag is missing, pick an interface you want to enable IP multicast transmission on, and then execute the `ifconfig` utility as root:

```
#  /sbin/ifconfig eth0 multicast
```

You must also verify that a route exists to the multicast interface in the host's routing table. Invoke the route utility as shown:

```
$ /sbin/route
Kernel IP routing table
Destination     Gateway         Genmask         Flags  Metric Ref    Use Iface
128.59.16.0     *               255.255.248.0   U      0      0        0 eth0
127.0.0.0       *               255.0.0.0       U      0      0        0 lo
224.0.0.0       *               240.0.0.0       U      0      0        0 eth0
default         vortex-gw.net.c 0.0.0.0         UG     0      0        0 eth0
```

Look for the line listing the 224.0.0.0/240.0.0.0 route (shown in bold). If it is missing you must add it by invoking:

```
# >/sbin/route add –net 224.0.0.0 netmask 240.0.0.0 dev eth0
```

Replace `eth0` with the interface you selected in the previous step.

Testing Your Configuration

The following small program may be used to test your host's multicast configuration. The program creates a `MulticastSocket` bound to a specific port, and joins a multicast group belonging to the link-local administrative scope. We further guarantee that multicast datagrams will not leave the local network by explicitly setting the TTL to `1` (its default value).

```
// TestLocalMulticast.java

import java.net.*;

public class TestLocalMulticast {

  public static void main(String[] args)
          throws UnknownHostException, SocketException,
                 java.io.IOException {

    int port = 5265;
    InetAddress group = InetAddress.getByName("239.255.10.10");

    System.out.println("Binding multicast socket to "
                       + group.getHostAddress() + ":" + port + " ...");
    MulticastSocket msocket = new MulticastSocket(port);
    msocket.setSoTimeout(10000);
    msocket.setTimeToLive(1);   // restrict to local delivery

    System.out.println("Requesting multicast group membership ...");
    msocket.joinGroup(group);
```

A `DatagramPacket` instance is then created, and a message is addressed to the multicast group at the specified port. After sending the datagram to the group, the program tries to receive a datagram. Because the multicast datagrams loop-back to the sender, and we have already joined the group, we would expect to receive the datagram that we just sent. Although the multicast service is unreliable, it is extremely unlikely that a datagram would be lost during loop-back delivery. If the datagram is not received within 10 seconds, the socket timeout will expire and an exception will be thrown. Note that o error handling is performed in the program to keep the example simple.

```
      String outMessage = "Hello multicast world!";
      byte[] data = outMessage.getBytes();
      DatagramPacket packet = new DatagramPacket(data, data.length, group,
                                                 port);

      System.out.println("Sending multicast message: " + outMessage);
      msocket.send(packet);

      packet.setData(new byte[512]);
      packet.setLength(512);   // very important!

      System.out.println("Waiting for multicast datagram ...");
      msocket.receive(packet);

      String inMessage = new String(packet.getData(), 0, packet.getLength());
      System.out.println("Received message: " + inMessage);

      System.out.println("Leaving multicast group ...");
      msocket.leaveGroup(group);

      msocket.close();
    }
  }
```

On a host correctly configured for at least local IP multicast delivery the above program should produce the output shown below.

```
C:\Beg_Java_Networking\Ch11>javac TestLocalMulticast.java

C:\Beg_Java_Networking\Ch11>java -classpath . TestLocalMulticast
Binding multicast socket to 239.255.10.10:5265 ...
Requesting multicast group membership ...
Sending multicast message: Hello multicast world!
Waiting for multicast datagram ...
Received message: Hello multicast world!
Leaving multicast group ...
```

If an exception is thrown during execution, then the host has not been configured correctly for IP multicast. For example, on a Microsoft Windows host that does not have a real IP interface, the `BindException` will be thrown when attempting to join the multicast group. On GNU/Linux platforms that have not been properly configured the binding and joining requests may succeed, but the datagram receipt will time-out. In such cases, readers should refer to the previous platform configuration discussion and the documentation of the operating system.

In-depth Example: A Group Chat Application

We will develop an in-depth example that will help us highlight some valuable protocol and implementation design patterns. Our goal will be to build a simple group chat application. The application will permit multiple users to join in a chat group and exchange simple string messages. Chat participants will assign themselves nicknames to be used for identification during the session. To simplify the application, no attempt will be made to guarantee uniqueness of nicknames.

Protocol Design

We will use the peer-to-peer paradigm, no single entity will be in control of the chat. As in the multicast model, anyone may join or leave a chat session, and members may send as many messages as they like. In fact, due to the design of the IP multicast protocol, users who have not even joined the group may send messages. Message delivery will not be guaranteed either.

> **Providing scalable reliable multicast is a difficult problem. Reliable multicast is a hot research area but as of today no single proposal has been standardized upon. In the meanwhile, it is advised that multicast be used only for applications that can accept datagram loss.**

Our multicast chat protocol will be simple because we have matched it closely to the IP multicast model. Had our specification required the assignment of unique names, authentication and privacy (encryption) the protocol would have been much more complicated. As it stands, the protocol will use the following three simple PDUs (Protocol Data Units), a.k.a. message-types:

- **JOIN PDU**: This is a datagram multicast when a user first joins; includes the user's self-assigned username (string)
- **LEAVE PDU**: This datagram is multicast when a user leaves the group; it includes the user's self-assigned username
- **MESSAGE PDU**: This datagram is multicast when a user finishes typing a message; it includes the self-assigned username, and a string containing the message

Every multicast application must be capable of coexisting with another application using a different protocol on the same multicast group and port. One simple solution is to begin every datagram sent with a so-called magic number. The magic number can be any binary value whose length makes it highly unlikely that the same value will be picked by two protocol authors. In the multicast chat application we will be using a randomly chosen 64-bit value. Following the magic number, every multicast chat datagram will contain a 32-bit integer value identifying the PDU-type (`JOIN`, `LEAVE`, `MESSAGE`). This is likely overkill in terms of space, but that's what programmers probably thought when deciding not to use a 4-byte year representation (which led to the well-known year 2000 problem).

The above pretty much describes the multicast chat application-layer protocol. No synchronization will be provided, so chat messages may arrive at different receivers in different order. You may think that this can simply be addressed by adding a timestamp to the message, but there is a catch. This is that the clocks are unlikely to be closely synchronized and therefore should not be relied upon for ordering. This is a well-known distributed programming problem that has been studied extensively. Interesting enough, many of the algorithms developed are being replaced today by GPS-based clocks whose synchronization can be guaranteed to an extremely small margin of error.

Implementation

Having completed the group chat protocol design, we proceed to design the programmatic and user interface of our application. As in the previous examples, we would like to separate the two in order to support multiple user interfaces. The programmatic interface will consist of a Java class called `MulticastChat` whose instances will encapsulate the participation in a multicast chat. Users may send messages by invoking the `sendMessage()` method of this class. A group chat is a two-way process so we must design a method for receiving incoming messages. For this purpose, we employ the Java AWT event model by defining the `MulticastChatEventListener` interface. Users of `MulticastChat` instances may subscribe to receive notification of incoming multicast chat events (join, leave, message) by registering an object implementing this listener interface. The listener interface is shown below. We have deviated a little from the AWT event model by not defining a `MulticastChatEvent` class in order to keep the example manageable.

```
// MulticastChatEventListener.java

import java.net.InetAddress;

public interface MulticastChatEventListener
  extends java.util.EventListener {

  // Invoked when a multicast chat message has been received
  void chatMessageReceived(String username, InetAddress host, int port,
                           String message);

  // Invoked when a multicast participant has joined
  void chatParticipantJoined(String username, InetAddress host, int port);

  // Invoked when a multicast participant has left
  void chatParticipantLeft(String username, InetAddress host, int port);
}
```

As stated, the `MulticastChat` class provides a simple programmatic interface to the multicast chat protocol. The constructor takes the participant's self-selected chat identity, the address of the IP multicast group, the port number, and a listener object as arguments. At construction time, the multicast socket is initialized and the multicast chat JOIN message is sent. An alternative design would be to require users to invoke a `sendJoin()` method, or to send the JOIN message in response to the first `sendMessage()` invocation. The advantage of creating and initializing the multicast socket in the constructor is that any exception can be propagated to the caller. Before returning, the constructor starts a new thread that will be used to perform blocking reads from the multicast socket.

This design is similar to the one used in Chapter 10. The class attribute state requirements are predictable. The class must store the constructor's arguments and the multicast socket instance created. This class is a programmatic interface to a simple peer-to-peer multicast chat protocol. Instances of this class provide an entry-point into a multicast chat characterized by a multicast group address and a port number. Users of `MulticastChat` instances may send multicast chat messages (JOIN, LEAVE, MESSAGE), and subscribe for notification of chat message receipt through the `MulticastChatEventListener` interface.

```
// MulticastChat.java

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

```
public class MulticastChat extends Thread {

  // Identifies a JOIN multicast chat PDU
  public static final int JOIN = 1;

  // Identifies a LEAVE multicast chat PDU
  public static final int LEAVE = 2;

  // Identifies a MESSAGE multicast chat PDU
  public static final int MESSAGE = 3;

  // Chat protocol magic number (preceeds all requests)
  public static final long CHAT_MAGIC_NUMBER = 4969756929653643804L;

  // Default number of milliseconds between terminations checks
  public static final int DEFAULT_SOCKET_TIMEOUT_MILLIS = 5000;

  // Multicast socket used to send and receive multicast protocol PDUs
  protected MulticastSocket msocket;

  // Chat username
  protected String username;

  // Multicast group used
  protected InetAddress group;

  // Listener for multicast chat events
  protected MulticastChatEventListener listener;

  // Controls receive thread execution
  protected boolean isActive;
```

The MulticastChat constructor expects the following arguments:

- `group`: This is the multicast group used for communications
- `port`: This is the port used to bind the multicast socket
- `ttl`: This defines the time to live value used in multicast transmission and hence determines the multicast radius
- `listener`: This defines the object to receive notification of chat events

The constructor initializes the instance variables of the new object and then creates a multicast socket bound to the port argument number and configured to join the multicast group specified in the group argument. Once the multicast socket has been configured, the constructor starts its thread and sends the multicast JOIN request.

```
  public MulticastChat(String username, InetAddress group, int port,
                       int ttl,
                       MulticastChatEventListener listener) throws IOException {

    this.username = username;
    this.group = group;
    this.listener = listener;
```

```
      isActive = true;

      // create & configure multicast socket
      msocket = new MulticastSocket(port);
      msocket.setSoTimeout(DEFAULT_SOCKET_TIMEOUT_MILLIS);
      msocket.setTimeToLive(ttl);
      msocket.joinGroup(group);

      // start receive thread and send multicast join message
      start();
      sendJoin();
    }
```

We repeat the thread termination pattern used in the TCP and UDP chapter examples, by defining a `terminate()` method. The method changes the `isActive` instance variable so that the next time, the thread read loop is terminated when the socket read timeout expires, or after the next packet is received. It also sends a multicast chat leave message.

```
    public void terminate() throws IOException {
      isActive = false;
      sendLeave();
    }

    // Issues an error message
    protected void error(String message) {
      System.err.println(new java.util.Date() + ": MulticastChat: "
                         + message);
    }
```

Following are methods for sending and receiving multicast chat PDUs. The `sendMessage()` method is public while the `sendJoin()` and `sendLeave()` methods are protected to assure that they will only be called by the constructor and the `terminate()` method. All receive methods are protected because they should only be invoked internally by the class instance thread. The send and receive methods are paired to facilitate protocol verification.

PDUs are written and read using the `DataOutputStream` and `DataInputStream` I/O utilities. Their encoding is well defined and could in principle be replicated by non-Java applications. Every send method begins by writing the multicast chat protocol magic number, and the PDU ID. The receive methods do not reciprocate because the magic number and PDU ID have already been read by the de-multiplexing read loop thread.

```
  // Sends a multicast chat JOIN PDU
    protected void sendJoin() throws IOException {
      ByteArrayOutputStream byteStream = new ByteArrayOutputStream();
      DataOutputStream dataStream = new DataOutputStream(byteStream);

      dataStream.writeLong(CHAT_MAGIC_NUMBER);
      dataStream.writeInt(JOIN);
      dataStream.writeUTF(username);
      dataStream.close();

      byte[] data = byteStream.toByteArray();
      DatagramPacket packet = new DatagramPacket(data, data.length, group,
                                                 msocket.getLocalPort());
      msocket.send(packet);
```

```
    }

    // Processes a multicast chat JOIN PDU and notifies listeners
    protected void processJoin(DataInputStream istream, InetAddress address,
                               int port) throws IOException {
      String name = istream.readUTF();

      try {
        listener.chatParticipantJoined(name, address, port);
      } catch (Throwable e) {}
    }

    // Sends a multicast chat LEAVE PDU
    protected void sendLeave() throws IOException {

      ByteArrayOutputStream byteStream = new ByteArrayOutputStream();
      DataOutputStream dataStream = new DataOutputStream(byteStream);

      dataStream.writeLong(CHAT_MAGIC_NUMBER);
      dataStream.writeInt(LEAVE);
      dataStream.writeUTF(username);
      dataStream.close();

      byte[] data = byteStream.toByteArray();
      DatagramPacket packet = new DatagramPacket(data, data.length, group,
                                                 msocket.getLocalPort());
      msocket.send(packet);
    }

    // Processes a multicast chat LEAVE PDU and notifies listeners
    protected void processLeave(DataInputStream istream, InetAddress address,
                                int port) throws IOException {
      String username = istream.readUTF();

      try {
        listener.chatParticipantLeft(username, address, port);
      } catch (Throwable e) {}
    }

    // Sends a multicast chat MESSAGE PDU
    public void sendMessage(String message) throws IOException {

      ByteArrayOutputStream byteStream = new ByteArrayOutputStream();
      DataOutputStream dataStream = new DataOutputStream(byteStream);

      dataStream.writeLong(CHAT_MAGIC_NUMBER);
      dataStream.writeInt(MESSAGE);
      dataStream.writeUTF(username);
      dataStream.writeUTF(message);
      dataStream.close();

      byte[] data = byteStream.toByteArray();
      DatagramPacket packet = new DatagramPacket(data, data.length, group,
                                                 msocket.getLocalPort());
      msocket.send(packet);
    }

    // Processes a multicast chat MESSAGE PDU and notifies listeners
    protected void processMessage(DataInputStream istream,
                                  InetAddress address,
```

```
                                    int port) throws IOException {
      String username = istream.readUTF();
      String message = istream.readUTF();

      try {
        listener.chatMessageReceived(username, address, port, message);
      } catch (Throwable e) {}
    }
```

After studying the earlier TCP and UDP examples, the main read loop should be familiar now. While a `terminate()` request has not been made, the next datagram packet is read from the multicast socket. The `DataOutputStream` is used to verify the multicast chat protocol magic number and determine the PDU ID. To keep the example simple, any errors are silently ignored.

```
  // Loops receiving and de-multiplexing chat datagrams
    public void run() {
      byte[] buffer = new byte[65508];
      DatagramPacket packet = new DatagramPacket(buffer, buffer.length);

      while (isActive) {
        try {

          // DatagramPacket instance length MUST be reset before EVERY receive
          packet.setLength(buffer.length);
          msocket.receive(packet);

          DataInputStream istream =
            new DataInputStream(new ByteArrayInputStream(packet.getData(),
                  packet.getOffset(), packet.getLength()));

          long magic = istream.readLong();

          if (magic != CHAT_MAGIC_NUMBER) {
            continue;

          }
          int opCode = istream.readInt();
          switch (opCode) {
          case JOIN:
            processJoin(istream, packet.getAddress(), packet.getPort());
            break;
          case LEAVE:
            processLeave(istream, packet.getAddress(), packet.getPort());
            break;
          case MESSAGE:
            processMessage(istream, packet.getAddress(), packet.getPort());
            break;
          default:
            error("Received unexpected operation code " + opCode + " from "
                  + packet.getAddress() + ":" + packet.getPort());
          }

        } catch (InterruptedIOException e) {

        /**
         * No need to do anything since the timeout is only used to
         * force a loop-back and check of the "isActive" value
         */
        } catch (Throwable e) {
```

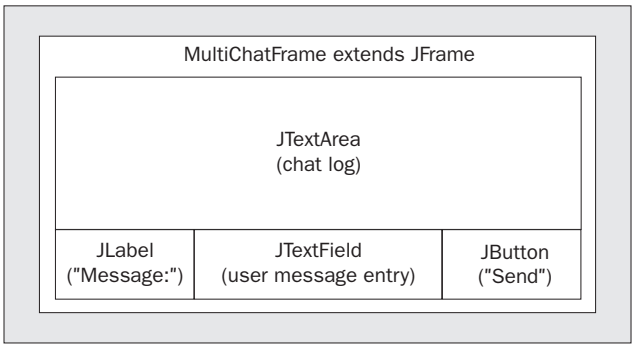
```
          error("Processing error: " + e.getClass().getName() + ": "
                + e.getMessage());
        }
      }

      try {
        msocket.close();
      } catch (Throwable e) {}
    }
  }
```

Our programmatic interface implementation is now complete. The advantage of using the peer-to-peer model is that there are no separate client and server implementations!

The User Interface

The final step will be to develop a user interface for our group chat application. In the previous chapters, we made use of simple command line interfaces. Due to the asynchronous nature of the multicast protocol – messages can arrive while the user is typing – a console-based interface is not really appropriate. Instead, we will develop a simple Swing-based graphical interface. We will assume that readers are familiar with Swing-based GUI development. For those who are not, we recommend looking at *Beginning Java 2 JDK 1.3 Edition, Wrox Press, ISBN 1861003668*. The next section will include complete instructions on how to execute the client.

The user interface will consist of a window containing a large message-log area in the center, and a small message-entry area in the bottom. The Swing `JTextArea` class will be used to display incoming messages, and the `JTextField` will be used to support user message entry. Users may signal message completion by pressing the *ENTER* key, or by clicking on a *Send* button provided. The `MulticastChatFrame` class extends the Swing `JFrame` to encapsulate the graphical interface. The nesting of the Swing components is shown in the diagram below.



At construction time, the Swing components are initialized, but the group chat session is not initialized. This can be achieved by invoking the `join()` method with the username, IP multicast group, and port to be used. The `join()` method then creates a `MulticastChat` instance passing its own parameters and the `MulticastChatFrame` object as a listener. After this initialization, all operations are asynchronous. Either a message arrives, signaled by a `MulticastChatEventListener` method invocation, or the user sends a message. Typically, the interface would provide a menu-based system for parameter entry, but to keep the example small, we read that information from the command line and invoke the `join()` method from the `main()` method.

The Java Swing classes are not thread safe! Once a Swing component has been displayed, all its methods must be invoked from within the Swing event thread. In our application, `MulticastChatEventListener` methods are invoked in the `MulticastChat read()` thread. Therefore, we cannot directly log messages received by invoking methods on the `JTextArea` object. Instead, we use the `SwingUtilities.invokeLater()` method to queue the operation on the Swing event thread. Care must also be taken when performing blocking operations from within the Swing event queue. If the Swing event queue blocks, then the Swing components will "freeze". Because our application uses a non-blocking protocol for sending, we do not have to worry about this problem. Had this not been the case, the *send* `JButton` action would have needed to use a thread to invoke the `MulticastChat.send()` method.

The code for the user interface class is shown below:

```
// MulticastChatFrame.java

import java.io.IOException;
import java.net.InetAddress;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.text.*;

// A swing-based user interface to a MulticastChat session

public class MulticastChatFrame extends JFrame implements ActionListener,
        WindowListener, MulticastChatEventListener {

  // The multicast chat object
  protected MulticastChat chat;

  // Text area used to log chat join, leave and chat messages received
  protected JTextArea textArea;

  // Scroll pane used for the text area (used to auto-scroll)
  protected JScrollPane textAreaScrollPane;

  // The text field used for message entry
  protected JTextField messageField;

  // Button used to transmit messageField data
  protected JButton sendButton;

  // Constructs a new swing multicast chat frame (in unconnected state)
  public MulticastChatFrame() {
    super("MulticastChat (unconnected)");

    // Construct GUI components (before session)
    textArea = new JTextArea();
    textArea.setEditable(false);
    textArea.setBorder(BorderFactory.createLoweredBevelBorder());

    textAreaScrollPane = new JScrollPane(textArea);
    getContentPane().add(textAreaScrollPane, BorderLayout.CENTER);
```

```
      JPanel messagePanel = new JPanel();
      messagePanel.setLayout(new BorderLayout());

      messagePanel.add(new JLabel("Message:"), BorderLayout.WEST);

      messageField = new JTextField();
      messageField.addActionListener(this);
      messagePanel.add(messageField, BorderLayout.CENTER);

      sendButton = new JButton("Send");
      sendButton.addActionListener(this);
      messagePanel.add(sendButton, BorderLayout.EAST);

      getContentPane().add(messagePanel, BorderLayout.SOUTH);

      // detect window closing and terminate multicast chat session
      addWindowListener(this);
    }

    // Configures the multicast chat session for this interface
    public void join(String username, InetAddress group, int port,
                     int ttl) throws IOException {
      setTitle("MulticastChat " + username + "@" + group.getHostAddress()
               + ":" + port + " [TTL=" + ttl + "]");

      // create multicast chat session
      chat = new MulticastChat(username, group, port, ttl, this);
    }
```

The protected log method is used internally to write a message in the chat text area, and scroll the pane to display it in a thread-safe manner (using `invokeLater`). This must be invoked on swing thread since we're invoked from the action listener methods in the context of the `MulticastChat` receive thread.

```
  protected void log(final String message) {
      java.util.Date date = new java.util.Date();

      SwingUtilities.invokeLater(new Runnable() {
        public void run() {
          textArea.append(message + "\n");
          textAreaScrollPane.getVerticalScrollBar()
            .setValue(textAreaScrollPane.getVerticalScrollBar().getMaximum());
        }
      });
    }
```

The `actionPerformed` method is invoked by the Swing event queue in response to asynchronous `ActionListener` events. In our program, it will be invoked when the user presses *ENTER* in the `messageField`, or presses the *Send* button.

```
    public void actionPerformed(ActionEvent e) {

      if ( (e.getSource().equals(messageField)) ||
           (e.getSource().equals(sendButton)) ) {

        String message = messageField.getText();
        messageField.setText("");
```

```
        try {
          chat.sendMessage(message);
        } catch (Throwable ex) {
          JOptionPane.showMessageDialog
            (this, "Error sending message: " + ex.getMessage(),
             "Chat Error", JOptionPane.ERROR_MESSAGE);
        }
      }
    }
```

In the constructor we subscribed to receive notification of window events. Several window-related events may be received. Our application is interested in dealing with two of these: when the window is first opened, and when the user clicks on the window close button. In the `windowOpen` event, we request focus for the user entry `messageField` so that users do not have to explicitly click in the message area to type. In the `windowClosing` event that is triggered when the user clicks on the window close button, we request termination of the chat session.

```
  // Invoked the first time a window is made visible.
    public void windowOpened(WindowEvent e) {
      messageField.requestFocus();
    }

    // On closing, terminate multicast chat
    public void windowClosing(WindowEvent e) {
      try {
        if (chat != null) {
          chat.terminate();
        }
      } catch (Throwable ex) {
        JOptionPane.showMessageDialog(this,
                                      "Error leaving chat: "
                                      + ex.getMessage(), "Chat Error",
                                      JOptionPane.ERROR_MESSAGE);
      }
      dispose();
    }
    public void windowClosed(WindowEvent e) {}
    public void windowIconified(WindowEvent e) {}
    public void windowDeiconified(WindowEvent e) {}
    public void windowActivated(WindowEvent e) {}
    public void windowDeactivated(WindowEvent e) {}
```

In order to display incoming JOIN, LEAVE and MESSAGE events we subscribed with the `MulticastChat` object at construction time. The `MulticastChat` object will invoke one of the three `MulticastChatEventListener` methods listed below. Recall that these methods will be invoked in the `MulticastChat` thread. Therefore, all three implementations log the message to the text area using the protected Swing thread-safe `log()` method listed earlier.

```
    // Invoked by the MulticastChat receive thread when a message has arrived
    public void chatMessageReceived(String username, InetAddress address,
                                    int port, String message) {
      log(username + ": " + message);
    }

    // Invoked by the MulticastChat receive thread when a user has joined
    public void chatParticipantJoined(String username, InetAddress address,
```

```
                                        int port) {
    log("+++ " + username + " has joined from " + address.getHostName()
        + ":" + port);
  }

  // Invoked by the MulticastChat receive thread when a user has left
  public void chatParticipantLeft(String username, InetAddress address,
                                  int port) {
    log("--- " + username + " has left from " + address.getHostName() + ":"
        + port);
  }
```

The `main()` method is responsible for parsing the command-line arguments that include the nickname (username) used in the chat session, the multicast group address and port number, as well as an optional time-to-live (TTL) value. In a production client these values would most likely be queried using a graphical interface.

```
// Command-line invocation expecting three arguments
  public static void main(String[] args) {
    if ((args.length != 3) && (args.length != 4)) {
      System.err.println("Usage: MulticastChatFrame "
                         + "<username> <group> <port> { <ttl> }");
      System.err.println("       - default time-to-live value is 1");
      System.exit(1);
    }

    String username = args[0];
    InetAddress group = null;
    int port = -1;
    int ttl = 1;

    try {
      group = InetAddress.getByName(args[1]);
    } catch (Throwable e) {
      System.err.println("Invalid multicast group address: "
                         + e.getMessage());
      System.exit(1);
    }

    if (!group.isMulticastAddress()) {
      System.err.println("Group argument '" + args[1]
                         + "' is not a multicast address");
      System.exit(1);
    }

    try {
      port = Integer.parseInt(args[2]);
    } catch (NumberFormatException e) {
      System.err.println("Invalid port number argument: " + args[2]);
      System.exit(1);
    }

    if (args.length >= 4) {
      try {
        ttl = Integer.parseInt(args[3]);
      } catch (NumberFormatException e) {
        System.err.println("Invalid TTL number argument: " + args[3]);
        System.exit(1);
```

```
        }
      }

      try {
        MulticastChatFrame frame = new MulticastChatFrame();
        frame.setSize(400, 150);

        frame.addWindowListener(new WindowAdapter() {
          public void windowClosed(WindowEvent e) {
            System.exit(0);
          }
        });

        frame.show();

        frame.join(username, group, port, ttl);
      } catch (Throwable e) {
        System.err.println("Error starting frame: " + e.getClass().getName()
                           + ": " + e.getMessage());
        System.exit(1);
      }
    }
  }
```

Running the Example

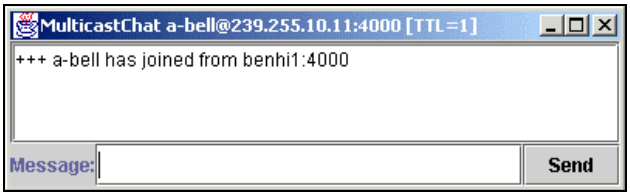
The group chat example code should all be stored in the same directory. The first step will then be to compile the Java source as shown:

C:\Beg_Java_Networking\Ch11>**javac MulticastChatEventListener.java MulticastChat.java MulticastChatFrame.java**

We are going to demonstrate use of the group chat client on a single host. It will be assumed that the host supports multicast as discussed in an earlier section. In our demonstration, we will start two instances of the multicast chat program, one for user A. Bell and another for user T. Watson. An arbitrary local administrative scope IP multicast address (`239.255.10.11`) and UDP port (`4000`) will be used. In this example, `benhi1` is the name of the Microsoft Windows 2000 host used.

We first start the group chat client for user A. Bell (username is "a-bell").

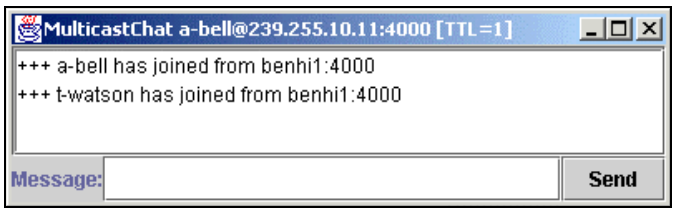
C:\Beg_Java_Networking\Ch11>**java -classpath . MulticastChatFrame a-bell 239.255.10.11 4000**



Notice how the chat group JOIN message sent by A. Bell's client is also received by and echoed on the same client (due to the loop-back effect). Next, we start a group chat client for user T. Watson (username is "t-watson").

C:\Beg_Java_Networking\Ch11>**java -classpath . MulticastChatFrame t-watson 239.255.10.11 4000**

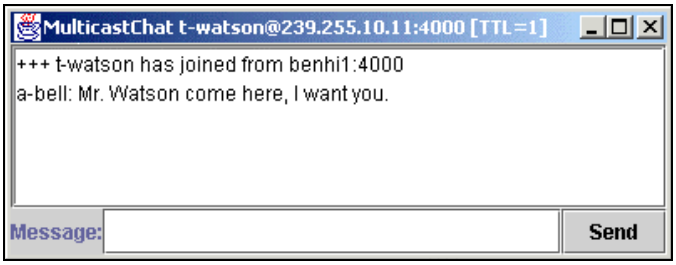
A separate client window is started and the chat group JOIN message from T. Watson's client should then be displayed on A. Bell's client as below:



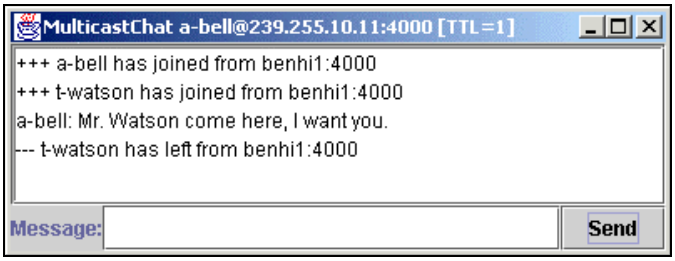
In the next step, we type in a message on A. Bell's client as shown below, and then press the *send* button. The message shown was reportedly the content of the first telephone conversation in history between Alexander Graham Bell and his assistant Thomas Watson.

Mr. Watson come here, I want you.

Once the *Send* button has been clicked (or the *ENTER* key pressed), the message will be sent to the multicast group and will be displayed on both clients.



Finally, we demonstrate use of the leave message by closing T. Watson's client window. The multicast chat leave message should be shown on A. Bell's client window as seen below.



Summary

In this chapter, we have seen that multicasting can offer important reductions in the bandwidth required by a server and backbone networks. Internet applications may effect multicast communications using the IP multicast protocol. Some main points to remember about IP multicasting are:

- In the IP multicast model any host may send a datagram to a group address.
- Groups of Internet hosts are identified by class-D Internet addresses in the range `224.0.0.0 – 239.255.255.255`.
- In order to receive multicast datagrams, hosts must request to join one or more groups. The request to join is sent to the first-hop router using the IGMP protocol.
- IP Multicast transmission is unreliable; multicast datagrams are routed with best effort.
- IP multicast transmissions may be scoped using two methods:
  - Using the IP TTL field to restrict the radius of transmission.
  - Administrative scoping which uses a new subdivision of the class-D address space.

Java supports IP multicast programming through the `java.net.MulticastSocket` class. The `MulticastSocket` class extends the UDP `DatagramSocket` class with support for IGMP join/leave operations, the setting of a datagram's TTL, and the binding to a particular network interface.

The Java IP multicast support is dependent on native platform multicast support. All major Java 2 platforms are IP multicast-capable, but may require appropriate configuration. Moreover, in order to send multicast datagrams to other hosts, the network routers must support IP multicast. The chapter concluded with an in-depth multicast example of a group chat application.