Chapter 6. Group Communication

In previous chapters, we have presented interprocess communications as the exchange of information between two processes. In this chapter, we will look at IPC among a group of processes, or group communication.

1. Unicasting versus Multicasting
In the IPC we have presented so far, data is sent from a source process, the sender, to one destination process, the receiver. This form of IPC can be called unicast, the sending of information to a single receiver, as opposed to multicast, the sending of information to multiple receivers. Unicast provides one-to-one IPC; multicast supports one-to-many IPC. See figure 1.

Whereas the majority of network services and network applications use unicast for IPC, multicast is useful for applications such as instant messages, groupware, online conferences, interactive distance learning, and can be used for applications such as real-time online auction. It can also be used in replication\(^1\) of services for fault tolerance\(^2\).

In an application or network service which makes use of multicasting, a set of processes form a group, called a multicast group. Each process in a group can send and receive message. A message sent by any process in the group can be received by each participating process in the group. A process may also choose to leave a multicast group.

In an application such as online conferencing, a group of processes interoperate using multicast to exchange audio, video, and/or text data. As before, our discussion will concentrate on the service layer, specially the IPC mechanism, for the application.

---
\(^1\) Replication of a service refers to maintaining duplicates of that service. A common technique for enhancing the availability of a service in the face of failures is to duplicate the data and the supporting system for that service.
\(^2\) Fault tolerance refers to the ability for an application to tolerate failures to some extent.
2 An Archetypal Multicast API

Let us first look at the facilities provided by any such mechanism before we proceed to study a specific API for multicast.

An application program interface which provides the functionalities of multicast must provide the following primitive operations:

- **Join** – This operation allows a process to join a specific multicast group. A process that has joined a multicast group is a member of the group and is entitled to receive all multicast addressed to the group. A process should be able to be a member of multiple multicast groups at any one time. Note that for this and other multicast operations, a naming scheme is needed to uniquely identify a multicast group.

- **Leave** – This operation allows a process to stop participating in a multicast group. A process that has left a multicast group is no longer a member of the group and is thereafter not entitled to receive any multicast addressed to the group, although the process may remain a member of other multicast groups.

- **Send** – This operation allows a process to send a message to all processes currently participating in a multicast group.

- **Receive** – This operation allows a member process to receive messages sent to a multicast group.

In Section 6.5 we will look at Java’s multicast API and sample programs using the API, at which time you will see how these primitive operations are provided using Java syntax. Before we do that, however, let us explore some of the interesting issues specific to multicast. These issues arise from the one-to-many nature of multicast.

3 Connection-oriented versus Connection-Oriented Multicast

A basic multicast mechanism is connectionless. The reason is obvious if you consider the one-to-many nature of multicast. In a group of \( n \) processes, if connection is to be established between the sender and every other process in the group, a total of \( n \) connections will be needed. Moreover, each of the \( n \) processes may potentially be a sender, so that each process must maintain a connection with every other process, resulting in a total of \( n \times n \) or \( n^2 \) connections. If \( n \) is large, the sheer number of such connections will become prohibitively expensive.

Moreover, connectionless IPC is appropriate for a very typical class of multicast applications: applications where audio or video data is transmitted among processes in real time\(^3\). When audio or video data is transmitted, the reduction in latency\(^4\) provided by connectionless communication outweighs the advantages offered by connection-oriented communication. When data for animation is sent, for example, it is more acceptable for a user to experience a distortion in the image of an occasional frame (a likely occurrence with connectionless communication) than a frequent, perceptible delay between consecutive frames (a likely occurrence with connection-oriented communication).

---

3 By real time it is meant that the latency (see below) between sending and receiving should be close to zero.

4 Latency refers to the delay in data transmission.
4 Reliable Multicast vs. Unreliable Multicast

When a multicast message is sent by a process, the runtime support of the multicast mechanism is responsible for delivering the message to each process currently in the multicast group. As each participating process may reside on a separate host, the delivery of these messages requires the cooperation of mechanisms running independently on those systems. Due to factors such as failures of network links and/or network hosts, routing delays, and differences in software and hardware, the time between when a unicast message is sent and when it is received may vary among the recipient processes. While the differences in message delivery to individual hosts may be insignificant if the machines are localized geographically, this may not be the case if the hosts are dispersed over a wide-area network.

Moreover, a message may never be received by one or more of the processes at all, due to errors and/or failures in the network, the machines, or the runtime support. Whereas some applications, such as video conferencing, can tolerate an occasional miss or misordering of messages, there are applications – such as database applications – for which such anomalies are unacceptable. Therefore, when employing a multicasting mechanism for an application, it is important that you choose one with the characteristics appropriate for your application. Otherwise, measures will need to be provided in the coding of the application in order to handle the anomalies which may occur in message delivery.

For that reason, it is useful to classify multicasting mechanisms in terms of their characteristics of message delivery.

4.1 Unreliable multicast: At its most basic, a multicast system will make a good-faith attempt to deliver messages to each participating process, but the delivery of the correct message to each process is not guaranteed. Thus, any message sent by a process may be received by zero or more processes. In the best case, the message, in its correct form, is received by all processes. In the worst case, the message may be received by no process correctly. In other cases, the message may be received by some but not all, or the messages may be received by some processes in a corrupted form. Such a system is said to provide unreliable multicast.

4.2 Reliable multicast: A multicast system which guarantees that each message is eventually delivered correctly to each process in the group is said to provide reliable multicast. In such a system, each message sent by a process can be assumed to be delivered in a non-corrupted form to all processes in the group eventually. The definition of reliable multicast requires that each participating process receives exactly one copy of each message sent. However, the definition places no restriction on the order that the messages are delivered to each process: each process may receive the messages in any permutation of those messages. For applications where the order of message delivery is significant, it is helpful to further classify reliable multicast systems as described below.
Unordered
An unordered reliable multicast system guarantees the safe delivery of each message, but it provides no guarantee on the delivery order of the messages. For example, suppose three processes $P_1$, $P_2$, and $P_3$ have formed a multicast group. Further suppose that three messages, $m_1$, $m_2$, $m_3$ have been sent to the group. Then an unordered reliable multicast system may deliver the messages to each of the three processes in any of the $3! = 6$ permutations ($m_1$-$m_2$-$m_3$, $m_1$-$m_3$-$m_2$, $m_2$-$m_1$-$m_3$, $m_2$-$m_3$-$m_1$, $m_3$-$m_1$-$m_2$, $m_3$-$m_2$-$m_1$). Note that it is possible for each participant to receive the messages in an order different from the orders of messages delivered to other participants. In our example, it is possible for $P_1$ to be delivered the messages in the order of $m_1$-$m_2$-$m_3$, $P_2$ to be delivered the messages in the order of $m_2$-$m_1$-$m_3$, and $P_3$ to be delivered the messages in the order of $m_1$-$m_3$-$m_2$. Of course it is also possible for $P_1$, $P_2$, and $P_3$ to each be delivered the messages in the same order, say $m_1$-$m_2$-$m_3$, but an application cannot make that assumption if it employs an unordered multicast mechanism.

FIFO multicast
A system which guarantees that the delivery of the messages adhere to the following condition is said to provide FIFO (first-in-first-out) or send-order multicast:

If process $P$ sent messages $m_i$ and $m_j$ in that order, then each process in the multicast group will be delivered the messages $m_i$ and $m_j$ in that order.

To illustrate this definition, let us look at an example. Suppose $P_1$ sends messages $m_1$, $m_2$, and $m_3$, in that order. Then, with FIFO multicast, each process in the group is guaranteed to have those messages delivered in that same order: $m_1$, $m_2$, then $m_3$. Note that this definition places no restriction on the delivery order among messages sent by different processes. To illustrate the point, let us use a simplified example of a multicast group of two processes: $P_1$ and $P_2$. Suppose $P_1$ sends messages $m_{11}$ then $m_{12}$, while $P_2$ sends messages $m_{21}$ then $m_{22}$. Then a FIFO multicast system can deliver the messages to each of the two processes in any of the following orders:

$m_{11}$-$m_{12}$-$m_{21}$-$m_{22}$, $m_{11}$-$m_{21}$-$m_{12}$-$m_{22}$, $m_{11}$-$m_{21}$-$m_{22}$-$m_{12}$,
$m_{21}$-$m_{11}$-$m_{12}$-$m_{22}$, $m_{21}$-$m_{11}$-$m_{22}$-$m_{12}$, $m_{21}$-$m_{22}$-$m_{11}$-$m_{12}$.

Note that while the messages sent by $P_1$ must be delivered to each process in the order of the sequence $m_{11}$-$m_{12}$, and the messages sent by $P_2$ must be delivered in the order of the sequence $m_{21}$-$m_{22}$, the two sequences can interleave in any manner.

Causal Order Multicast
A multicast system is said to provide causal multicast if its message delivery satisfies the following criterion:

If message $m_j$ causes (results in) the occurrence of message $m_j$, then $m_i$ will be delivered to each process prior to $m_j$. Messages $m_i$ and $m_j$ are said

---

5 Please note that the word “causal” is different from the word “casual”.
to have a causal or happen-before relationship, denoted \( m_i \rightarrow m_j \). The happen-before relationship is transitory: if \( m_i \rightarrow m_j \) and \( m_j \rightarrow m_k \), then \( m_i \rightarrow m_j \rightarrow m_k \). In this case, a causal-order multicast system guarantees that these three messages will be delivered to each process in the order of \( m_i, m_j, \) then \( m_k \).

As an illustration, suppose three processes \( P_1, P_2, \) and \( P_3 \) are in a multicast group. \( P_1 \) sends a message \( m_1 \), to which \( P_2 \) replies with a multicast message \( m_2 \). Since \( m_2 \) is triggered by \( m_1 \), the two messages share a causal relationship of \( m_1 \rightarrow m_2 \). Suppose the receiving of \( m_2 \) in turn triggers a multicast message \( m_3 \) sent by \( P_3 \), that is, \( m_2 \rightarrow m_3 \). The three messages then share the causal relationship of \( m_1 \rightarrow m_2 \rightarrow m_3 \). A causal-order multicast message system ensures that these three messages will be delivered to each of the three processes in the order of \( m_1\rightarrow m_2, m_3 \). Note that in this case there is no restriction on the order of message delivery if the multicast system were FIFO instead of causal.

As a variation of the above example, suppose \( P_1 \) multicasts message \( m_1 \), to which \( P_2 \) replies with a multicast message \( m_2 \), and independently \( P_3 \) replies to \( m_1 \) with a multicast message \( m_3 \). The three messages now share these causal relationships: \( m_1 \rightarrow m_2 \) and \( m_1 \rightarrow m_3 \). A causal-order multicast system can delivery these message to the participating processes in either of the following orders:

\[ m_1\rightarrow m_2\rightarrow m_3 \]
\[ m_1\rightarrow m_3\rightarrow m_2 \]

since the causal relations are preserved in either of the two sequences. In such a system, it is not possible for the messages to be delivered to any of the processes in any other permutation of the three messages, such as \( m_2 \rightarrow m_1 \rightarrow m_3 \) or \( m_3 \rightarrow m_1 \rightarrow m_2 \), the first of these violates the causal relationship \( m_1 \rightarrow m_2 \), while the second permutation violates the causal relationship \( m_1 \rightarrow m_3 \).

**Atomic order multicast**

In an atomic-order multicast system, all messages are guaranteed to be delivered to each participant in the exact same order. Note that the delivery order does not have to be FIFO or causal, but must be identical for each process.

Example:

- \( P_1 \) sends \( m_1 \), \( P_2 \) sends \( m_2 \), and \( P_3 \) sends \( m_3 \).
- An atomic system will guarantee that the messages will be delivered to each process in only one of the six orders: \( m_1\rightarrow m_2\rightarrow m_3, m_1\rightarrow m_3\rightarrow m_2, m_2\rightarrow m_1 \rightarrow m_3, m_2\rightarrow m_3 \rightarrow m_1, m_3\rightarrow m_1 \rightarrow m_2, m_3\rightarrow m_2\rightarrow m_1 \).

Example:

- \( P_1 \) sends \( m_1 \) then \( m_2 \).
- \( P_2 \) replies to \( m_1 \) by sending \( m_3 \).
- \( P_3 \) replies to \( m_3 \) by sending \( m_4 \).

Although atomic multicast imposes no ordering on these messages, the sequence of the events dictates that \( P_1 \) must be delivered \( m_1 \) before sending \( m_2 \). Likewise,

---

6 The causal relationship is transitive: if \( e_i \rightarrow e_j \), and \( e_j \rightarrow e_k \), then \( e_i \rightarrow e_j \rightarrow e_k \).
$P_2$ must receive $m_1$ then $m_3$, while $P_3$ must receive $m_3$ before $m_4$. Hence any atomic delivery order must preserve the order $m_1, m_3, m_4$. The remaining message $m_2$ can, however, be interleaved with these messages in any manner. Thus an atomic multicast will result in the messages being delivered to each of the processes in one of the following orders: $m_1, m_2, m_3, m_4, m_1, m_3, m_2, m_4$, or $m_1, m_3, m_2, m_4, m_2$. For example, each process may be delivered the messages in this order $m_1, m_3, m_2, m_4$.

5. The Java Basic Multicast API

We are now ready to study a programming tool for using multicast in an application. The tool is the basic Java multicast API [1].

At the transport layer, the basic multicast supported by Java is an extension of UDP (the User Datagram Protocol), which, as you recall, is connectionless and unreliable. At the network layer of the network architecture, multicast packets are transmitted across networks using Internet multicast routing [5], supported by routers (known as mrouters) capable of multicast routing in addition to unicast routing. Multicast on a local network (one interconnected without a router) is carried out using multicast supported by the local-area-network protocol (such as Ethernet multicast). The routing and delivery of multicast messages are topics beyond the scope of this book. Fortunately, such matters are transparent to an application programmer using a multicast API.

For the basic multicast API, Java provides a set of classes which are closely related to the datagram socket API classes that we looked at in Chapter 3. There are three major classes in the API, the first three of which we have already seen in the context of datagram sockets.

1. **InetAddress**: In the datagram socket API, this class represents the IP address of the sender or receiver. In multicasting, this class can be used to identify a multicast group (see next section).
2. **DatagramPacket**: As with datagram sockets, an object of this class represents an actual datagram; in multicast, a DatagramPacket object represents a packet of data sent to all participants or received by each participant in a multicast group.
3. **MulticastSocket**: The **MulticastSocket** class is derived from the **DatagramSocket** class, with additional capabilities for joining and leaving a multicast group. An object of the multicast datagram socket class can be used for sending and receiving IP multicast packets.

5.1 IP Multicast addresses

Recall that in the Java **unicast** socket API, a sender identifies a receiver by specifying the host name of the receiving process, as well as the protocol port to which the receiving process is bound.

Consider for a moment what a **multicast** sender need to address. Instead of a single process, a multicast datagram is meant to be received by all the processes that are currently members of a specific multicast group. Hence each multicast datagram needs to be addressed to a multicast group instead of an individual process.
The Java multicast API uses the Internet Protocol (IP) multicast addresses for identifying multicast groups.

In IPv4\(^7\), a multicast group is specified by (i) a class D IP address combined with (ii) a standard UDP port number. Recall from Chapter 1 that Class D IP addresses are those with the prefix bit string of 1110, and hence these addresses are in the range of 224.0.0.0 to 239.255.255.255, inclusive. Excluding the four prefix bits, there are 32-4=28 remaining bits, resulting in an address space size of \(2^{28}\); that is, approximate 268 million class D addresses are available, although the address 224.0.0.0 is reserved and should not be used by any application. IPv4 multicast addresses are managed and assigned by the Internet Assigned Numbers Authority (IANA) [3].

An application which uses the Java multicast API must specify at least one multicast address for the application. To select a multicast address for an application, there are the following options:

1. Obtain a permanently assigned static multicast address from IANA: Permanent addresses are limited to global, well-known Internet applications, and their allocations are highly restricted. A list of the currently assigned addresses can be found in [2]. Following is a sample of some of the most interesting of the assigned addresses:

   224.0.0.1 All Systems on this Subnet  
   224.0.0.11 Mobile-Agents  
   224.0.1.16 MUSIC-SERVICE  
   224.0.1.17 SEANET-TELEMETRY  
   224.0.1.18 SEANET-IMAGE  
   224.0.1.23 XINGTV  
   224.0.1.41 gatekeeper  
   224.0.1.84 jini-announcement  
   224.0.1.85 jini-request  
   224.0.1.115 Simple Multicast  
   224.0.6.000-224.0.6.127 Cornell ISIS Project  
   224.0.7.000-224.0.7.255 Where-Are-You  
   224.0.8.000-224.0.8.255 INTV  
   224.0.9.000-224.0.9.255 Invisible Worlds  
   224.0.11.000-224.0.11.125 NCC.NET Audio  
   224.0.12.000-224.0.12.063 Microsoft and MSNBC  
   224.0.16.000-224.0.16.255 XingNet  
   224.0.17.000-224.0.17.031 Mercantile & Commodity Exchange  
   224.0.17.064-224.0.17.127 ODN-DTV  
   224.0.18.000-224.0.18.255 Dow Jones  
   224.0.19.000-224.0.19.063 Walt Disney Company  
   224.0.22.000-224.0.22.255 WORLD MCAST  
   224.2.0.0-224.2.127.253 Multimedia Conference Calls

2. Choose an arbitrary address, assuming that the combination of the random address and

\(^7\) Multicast addressing in IPv6 is significantly different; see [4] for details.
3. Obtain a transient multicast address at runtime; such an address can be received by an application through the Session Announcement Protocol[6].

Option 3 is beyond the scope of this chapter. For our examples and exercises, we will make use of the static address 224.0.0.1, with an equivalent domain name ALL-SYSTEMS.MCAST.NET, for processes running on all machines on the local area network, such as those in your laboratory; alternatively, we may use an arbitrary address that has not been assigned, such as a number in the range of 239.*.*.* (for example, 239.1.2.3).

In the Java API, a MulticastSocket object is bound to a port address such as 3456, and methods of the object allows for the joining and leaving of a multicast address such as 239.1.2.3.

5.2 Joining a multicast group

To join a multicast group at IP address $m$ and UDP port $p$, a MulticastSocket object must be instantiated with $p$, then the object’s joinGroup method can be invoked specifying the address $m$:

```java
// join a Multicast group at IP address 239.1.2.3 and port 3456
InetAddress group = InetAddress.getByName("239.1.2.3");
MulticastSocket s = new MulticastSocket(3456);
s.joinGroup(group);
```

5.3 Sending to a multicast group

A multicast message can be sent using syntax similar to that for the datagram socket API. Specifically, a datagram packet must be created with the specification of a reference to a byte array containing the data, the length of the array, the multicast address, and a port number. The send method of the MulticastSocket object (inherited from the DatagramSocket class) can then be invoked to send the data.

It is not necessary for a process to join a multicast group in order to send messages to it, although it must do so in order to receiver the messages. When a message is sent to a multicast group, all processes that have joined the multicast group, which may include a sender, can be expected (but not guaranteed) to receive the message.

The following code segment illustrates the syntax for sending to a multicast group.

```java
String msg = "This is a multicast message."
InetAddress group = InetAddress.getByName("239.1.2.3");
MulticastSocket s = new MulticastSocket(3456);
s.joinGroup(group);  // optional
DatagramPacket hi = new DatagramPacket(msg.getBytes( ),
                                          msg.length( ),group, 3456);
s.send(hi);
```

Receiving messages sent to a multicast group
A process that has joined a multicast group may receive messages sent to the group using syntax similar to receiving data using a datagram socket API. The following code segment illustrates the syntax for receiving messages sent to a multicast group.

```java
byte[] buf = new byte[1000];
InetAddress group = InetAddress.getByName("239.1.2.3");
MulticastSocket s = new MulticastSocket(3456);
s.joinGroup(group);
DatagramPacket recv = new DatagramPacket(buf, buf.length);
s.receive(recv);
```

**Leaving a multicast group**

A process may leave a multicast group by invoking the `leaveGroup` method of a `MulticastSocket` object, specifying the multicast address of the group.

```java
s.leaveGroup(group);
```

**Setting the “time-to-live”**

The runtime support for a multicast API often employs a technique known as message propagation, whereby a packet is propagated from a host to a neighboring host in an algorithm which, when executed properly, will eventually deliver the message to all the participants. Under some anomalous circumstances, however, it is possible that the algorithm which controls the propagation does not terminate properly, resulting in a packet circulating in the network indefinitely. This phenomenon is undesirable, as it causes unnecessary overhead on the systems and the network. To avoid this occurrence, it is recommended that a “time to live” parameter be set with each multicast datagram. The time-to-live (ttl) parameter, when set, limits the count of network links or hops that the packet will be forwarded on the network indefinitely. In the Java API, this parameter can be set by invoking the `setTimeToLive` method of the sender’s `MulticastSocket`, as follows:

```java
String msg = "Hello everyone!";
InetAddress group = InetAddress.getByName("224.0.0.1");
MulticastSocket s = new MulticastSocket(3456);
s.setTimeToLive(1); // set time-to-live to 1 hop – a count appropriate for
                    // multicasting to local hosts
DatagramPacket hi = new DatagramPacket(msg.getBytes(),
                                         msg.length(),group, 3456);
s.send(hi);
```

The value specified for the `ttl` must be in the range $0 \leq ttl \leq 255$; an `IllegalArgumentException` will be thrown otherwise.

The recommended `ttl` settings are:

- 0 if the multicast is restricted to processes on the same host
- 1 if the multicast is restricted to processes on the same subnet
- 32 if the multicast is restricted to processes on the same site
- 64 if the multicast is restricted to is processes on the same region
128 is if the multicast is restricted to processes on the same continent
255 is the multicast is unrestricted

**Example 1**

Figures 2a and 2b show the coding for a simple example multicast application, presented here primarily to illustrate the syntax of the API. When run, each receiver process (Figure 2b) subscribes to the multicast group 239.1.2.3 at port 1234 and listens for a message. The sender process (Figure 2a), on the other hand, is not a member of the multicast group (although it can be); it sends a single message to the multicast group 239.1.2.3 at port 1234 before closing its multicast socket.

01. `import java.io.*;`
02. `import java.net.*;`
03. `/* *
04. * This example illustrates the basic syntax for basic multicast.
05. * @author M. L. Liu
06. */
07. public class Example1Sender {
08.   // An application which uses a multicast socket to send
09.   // a single message to a multicast group.
10.   // The message is specified as a command-line argument
11.   public static void main(String[] args) {
12.      MulticastSocket s;
13.      InetSocketAddress group;
14.      if (args.length != 1)
15.         System.out.println
16.            ("This program requires a command line argument");
17.      else {
18.         try {
19.            // create the multicast socket
20.            group = InetSocketAddress.getByName("239.1.2.3");
21.            s = new MulticastSocket(3456);
22.            s.setTimeToLive(32); // restrict multicast to processes
23.            // running on hosts at the same site.
24.            String msg = args[0];
25.            DatagramPacket packet =
26.              new DatagramPacket(msg.getBytes(), msg.length(),
27.                  group, 3456);
28.            s.send(packet);
29.            s.close();
30.        } 
31.        catch (Exception ex) { // here if an error has occurred
32.            ex.printStackTrace( );
33.        }
34.     } // end else
35. } // end main
36. } // end class

**Figure 2a. Example1Sender.java**

01. `import java.io.*;`
Distributed Computing, M. Liu

```java
import java.net.*;

/* *
 * This example illustrates the basic syntax for basic multicast.
 * @author M. L. Liu
 */
public class Example1Receiver {

  // An application which joins a multicast group and
  // receives a single message sent to the group.

  public static void main(String[] args) {
    MulticastSocket s;
    InetAddress group;
    try {
      // join a Multicast group and send the group salutations
      group = InetAddress.getByName("239.1.2.3");
      s = new MulticastSocket(3456);
      s.joinGroup(group);
      byte[] buf = new byte[100];
      DatagramPacket recv = new DatagramPacket(buf, buf.length);
      s.receive(recv);
      System.out.println(new String(buf));
      s.close();
    }
    catch (Exception ex) { // here if an error has occurred
      ex.printStackTrace();
    }
  }
  }// end main
  }// end class
```

Figure 2b. Example1Receiver.java

Example 2

As another illustration of the Java multicast API, we present an example where each process in a multicast group sends a message, and, independently, each process also displays all of the messages it receives as a member of the multicast group.

`Example2SenderReceiver.java` (Figure 3a) is the code for the application. For each process to receive all of the messages, each process must be ready to receive prior to the sending of any message. Hence in the main thread the process is kept waiting until it is time for the process to send its message. Meanwhile, a separate thread, of the `ReadThread` class (Figure 3b), is spawned to receive and echo the messages.
10. public class Example2SenderReceiver{
11.  
12.  // An application which uses a multicast socket to send
13.  // a single message to a multicast group, and a separate
14.  // thread which uses a separate multicast socket to receive
15.  // messages sent to the same group.
16.  // Three command-line arguments are expected:
17.  // <multicast IP address>,<multicast port>,<message>
18. 
19.  public static void main(String[ ] args) {
20.    
21.     InetAddress group = null;
22.     int port = 0;
23.     MulticastSocket socket = null;
24.     String characters;
25.     byte[ ] data = null;
26.     
27.     if (args.length !=3)
28.        System.out.println("Three command-line arguments are expected.")
29.     else {
30.            try {
31.                group = InetAddress.getByName(args[0]);
32.                port = Integer.parseInt(args[1]);
33.                characters = args[2];
34.                data = characters.getBytes();
35.                DatagramPacket packet =
36.                    new DatagramPacket(data, data.length, group, port);
37.                Thread theThread =
38.                    new Thread(new ReadThread(group, port));
39.                theThread.start();
40.                System.out.println("Hit return when ready to send.");
41.                InputStreamReader is = new InputStreamReader(System.in);
42.                BufferedReader br = new BufferedReader(is);
43.                br.readLine();
44.                socket = new MulticastSocket(port);
45.                socket.send(packet);
46.                socket.close( );
47.            } 
48.            catch (Exception se) {
49.                se.printStackTrace( );
50.            } 
51.        } //end else 
52.    } // end main 
53. 
54. } // end class
09. */
10. class ReadThread implements Runnable {
11.
12.    static final int MAX_LEN = 30;
13.    private InetAddress group;
14.    private int port;
15.
16.    public ReadThread(InetAddress group, int port) {
17.        this.group = group;
18.        this.port = port;
19.    }
20.
21.    public void run() {
22.        try {
23.            MulticastSocket socket = new MulticastSocket(port);
24.            socket.joinGroup(group);
25.            while (true) {
26.                byte[ ] data = new byte[MAX_LEN];
27.                DatagramPacket packet =
28.                    new DatagramPacket(data, data.length, group, port);
29.                socket.receive(packet);
30.                String s = new String(packet.getData());
31.                System.out.println(s);
32.            } // end while
33.        } // end try
34.        catch (Exception exception) {
35.            exception.printStackTrace();
36.        } // end catch
37.    } // end run
38.}
39.} // end class

Figure 3c. ReadThread.java

The Java basic multicast API and similar mechanisms can be employed to provide support for an application’s service logic. Note that an application may use a combination of unicast and multicast for its IPC; the details of the service logic application should be transparent to the application logic and presentation logic. An application that makes use of multicast is sometimes called a multicast-aware application.

For those who are interested in a sample multicast chatroom application, please see reference [7].

6. Reliable Multicast API
The Java multicast API that we have explored in this chapter is an extension of the datagram socket API. As a result, it shares a key characteristic of datagrams: unreliable
delivery. In particular, messages are not guaranteed to be delivered to any of the receiving processes. Hence, the API provides **unreliable multicast**.

As mentioned, there are applications for which unreliable multicast is unacceptable. For such applications, there exist a number of packages available that provide reliable multicast API, some of which are listed below:

- **The Java Reliable Multicast Service (JRM Service)** [8, 9] is a package which enhances the Java basic multicast API by providing the capabilities for a receiver to repair multicast data that are lost or damaged, as well as security measures to protect data privacy.
- The **Totem** system [10], developed by the University of California, Santa Barbara, “provides reliable totally ordered delivery of messages to processes within process groups on a single local-area network, or over multiple local-area networks interconnected by gateways.”
- TASC’s **Reliable Multicast Framework (RMF)**[11] provides reliable and send-ordered (FIFO) multicast.

The use of these packages is beyond the scope of this book. Interested readers are encouraged to consult the references for further details.

**Summary**

This chapter provides an introduction to the use of **group communication** in distributed computing. Topics covered include:

- **Unicast vs. multicast**: unicast is one-to-one communication, while multicast is one-to-many communication.
- An archetypal multicast API must provide operations for **joining** a multicast group, **leaving** a multicast group, **sending** to a group, and **receiving** multicast messages sent to a group.
- **Basic multicast** is **connectionless** and **unreliable**; in an unreliable multicast system, messages are not guaranteed to be safely delivered to each participant.
  - A **reliable** multicast system ensures that each message sent to a multicast group is delivered correctly to each participant. Reliable multicasts can be further categorized by the order of message delivery they support:
    - **Unordered** multicast may deliver the messages to each participant in any order.
    - **FIFO** multicast preserves the order of messages sent by each host.
    - **Causal** multicast preserves causal relationships among the messages.
    - **Atomic** multicast delivers the messages to each participant in the same order.
- **IP multicast addressing** uses a combination of a Class D address and a UDP port number. Class D IP addresses are managed and assigned by IANA. A multicast application may use a **static** Class D address, a **transient** address obtained at run time, or an **arbitrary unassigned** address.

---

8 In your exercises, if you run your processes on one host or on hosts on one subnet, you are not likely to observe any loss of messages or scrambling in the delivery order of the messages. These anomalies are more likely when the participating hosts are remotely connected.
The **Java basic multicast API** provides **unreliable multicast**. A **MulticastSocket** is created with the specification of a port number. The **joinGroup** and **leaveGroup** methods of the **MulticastSocket** class, a subclass of **DatagramSocket**, can be invoked to join or leave a specific multicast group; and the **send** and **receive** methods can be invoked to send and receive a multicast **datagram**. The **DatagramPacket** class is also needed to create the datagrams.

- There are existing packages that provide **reliable multicast**, including the Java Reliable Multicast Service (JRM Service).

### Exercises

1. Suppose a multicast group currently is participated by two processes: \( P_1 \) and \( P_2 \). Suppose \( P_1 \) multicasts \( m_{11} \) then \( m_{12} \), \( P_2 \) multicasts \( m_{21} \) then \( m_{22} \). Further assume that no message is lost in delivery.
   
   a) Theoretically, how many different orders can all four messages be delivered to each process if the messages are unrelated?
   
   b) Theoretically ow many different orders can all the messages be delivered to each process if the messages are casually related as \( m_{11} -> m_{21} -> m_{12} -> m_{22} \)?
   
   c) What are the possible orders of message delivery of each process if the messages are unrelated and the multicast is (i) FIFO, (ii) causal, and (iii) atomic?
   
   d) What are the possible orders of message delivery of each process if the messages are causally related as \( m_{11} -> m_{21} -> m_{12} -> m_{22} \) and the multicast is (i) FIFO, (ii) causal, and (iii) atomic?

2. Suppose the following events take place in chronological order, in a multicast group participated by three processes \( P_1 \), \( P_2 \), and \( P_3 \):
   
   - \( P_1 \) multicasts \( m_1 \).
   - \( P_2 \) responds to \( m_1 \) by multicasting \( m_2 \).
   - \( P_3 \) multicasts \( m_3 \) spontaneously.
   - \( P_1 \) responds to \( m_3 \) by multicasting \( m_4 \).
   - \( P_3 \) responds to \( m_2 \) by multicasting \( m_5 \).
   - \( P_2 \) multicasts \( m_6 \) spontaneously.

   For each of the following scenarios, state in the corresponding entry in the table below whether it is permitted or not by that mode of multicast.

   - All processes are delivered \( m_1, m_2, m_3, m_4, m_5, m_6 \), in that order
   - \( P_1 \) and \( P_2 \) are each delivered \( m_1, m_2, m_3, m_4, m_5, m_6 \).
   - \( P_3 \) is delivered \( m_2, m_3, m_1, m_4, m_5, m_6 \).
   - \( P_1 \) is delivered \( m_1, m_2, m_5, m_3, m_4, m_6 \).
   - \( P_2 \) is delivered \( m_1, m_3, m_5, m_4, m_2, m_6 \).
   - \( P_3 \) is delivered \( m_3, m_1, m_4, m_2, m_5, m_6 \).
   - \( P_1 \) is delivered \( m_1, m_2, m_3, m_4, m_5, m_6 \).
   - \( P_2 \) is delivered \( m_1, m_4, m_2, m_3, m_6, m_5 \).
P3 is delivered \( m_1, m_3, m_6, m_4, m_2, m_5 \).
e. P1 is delivered \( m_1, m_2, m_3, m_4, m_5, m_6 \)
P2 is delivered \( m_1, m_3, m_2, m_5, m_4, m_6 \)
P3 is delivered \( m_1, m_2, m_6, m_5, m_3, m_4 \).
f. \( P_1 \) is delivered \( m_2, m_1, m_6 \)
\( P_2 \) is delivered \( m_1, m_2, m_6 \)
\( P_3 \) is delivered \( m_6, m_2, m_1 \)
g. No message is delivered to any of the processes.

<table>
<thead>
<tr>
<th>scenario</th>
<th>Reliable multicast</th>
<th>FIFO multicast</th>
<th>Causal multicast</th>
<th>Atomic multicast</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3. This exercise is based on Example1 presented in this chapter.
   a) Compile the Example1*.java programs, then execute them in each of the following sequences, describe and explain the outcome of each:
      i. Start two or more Receiver processes first, then a Sender process with a message of your choice.
      ii. Start a Sender process with a message of your choice first, then two or more receiver processes.
   b) Based on Example1Receiver.java, create a program
      Example1aReceiver.java which joins a multicast group of a different IP address (e.g., 239.1.2.4) but the same port. Compile
      Example1bReceiver.java. Start two or more Example1Receiver processes first, then a Example1a Receiver process, and then a Sender process with a message of your choice. Does the Example1bReceiver process receive the message? Describe and explain the outcome.
   c) Based on Example1Receiver.java, create a program
      Example1bReceiver.java which joins a multicast group of the same IP address but a different port. Compile Example1bReceiver.java. Start two or more Example1Receiver processes first, then a Example1bReceiver process, and then a Sender process with a message of your choice. Does the Example1bReceiver process receive the message? Describe and explain the outcome.
   d) Based on Example1Sender.java, create a program
      Example1SenderReceiver.java which joins the multicast group, sends a message, then listens for (receives) a multicast message before closing the multicast socket and exiting. Compile the program, then start two or more Receiver processes before starting the SenderReceiver process. Describe the outcome. Turn in the listing of SenderReceiver.java.
e) Based on *Example1Sender.java*, create a program *Example1bSender.java* which sends a message to the multicast address of the program *Example1bReceiver.java*. Compile the program, then start an *Example1Receiver* process, an *Example1bReceiver* process, an *Example1Sender* process, then an *Example1bSender* process. Describe and explain the message(s) received by each process.

f) Based on *Example1Receiver.java* and *Example1bReceiver.java*, create a program *Example1cReceiver.java* which uses two threads (including the main thread). Each thread joins one of the two multicast groups and receive then display one message before leaving the group. You may find the sample *ReadThread.java* useful. Compile and run *Example1cReceiver.java*, then start an *Example1Sender* process, followed by an *Example1bSender* process. Does the receiver process display both messages? Turn in the program listings of *Example1cReceiver.java* and its thread class.

4. This exercise is based on Example2 presented in this chapter.
   a) Compile *Example2SenderReceiver.java*, then start two or more processes of the program, specifying with each a unique message. Example commands are as follows:

   ```
   java Example2SenderReceiver 239.1.2.3 1234 msg1
   java Example2SenderReceiver 239.1.2.3 1234 msg2
   java Example2SenderReceiver 239.1.2.3 1234 msg3
   ```

   In this example, each of the three processes should display on screen the messages msg1, msg2, and msg3.

   Be sure to start all processes before allowing each one to send its message.

   Describe the run outcomes.

   b) Modify *Example2SenderReceiver.java* so that each process sends out its message 10 times. Compile and run. Describe the run outcomes and hand in the program listings.

5. Write your own multicast application.
   Write an application such that multiple processes use group communication to carry out an election. There are two candidates: Jones and Smith. Each process multicasts its vote in a message that identifies itself and its vote. Each process keeps track of the vote count for each candidate, including its own. At the end of the election (when everyone in the group has voted), each process tallies the votes independently and display the outcome on its screen (e.g., Jones 10, Smith 5).

   Hand in the listings of your application and answer these questions:
   a. How does your design allow the participants to join a multicast group?
   b. How does your design synchronize the onset of the election so that every process is ready to receive any vote cast by a member in the group?
   c. In your run, do the independent tallies agree with each other? Can you assume that the tallies will always agree with each other? Explain.
References

2. IANA multicast-addresses, http://www.iana.org/assignments/multicast-addresses
9. Hans-Peter Bischof, JRMS Tutorial, Department of Computer Science, Rochester Institute of Technology.