How to Multicast Using Java Sockets

The Java Socket APIs enable network communication between remote hosts in the client-server paradigm. The communication can be established in three ways: one-to-one communication (client-server), one-to-all communication (broadcast), and one-to-many communication (multicast). This article elaborates on the overall concept of sockets in general and multicasting in particular, and shows how it can be implemented by using Java sockets.

An Overview: Socket and IP Address

A *socket* essentially means a designated virtual endpoint between machines in a network for the purpose of sending and receiving data. A machine in a network is uniquely identified by an IP Address. There is a specific format and IP classes that defines the type of network it is assigned to. With the evolution of the Internet, the demand for accommodating more devices into a network has increased. As a result, the IPv4 which used 32-bit addressing and can supported about 4.3 billion devices suddenly seems less accommodating. So, an improved version, called IPv6, is called for; it uses 64-bit addressing and can support about $3.4e10^4$ devices! However, IPv4 is still is in use and thriving; therefore, we'll restrict our discussion only to IPv4.

IPv4 address format example: 192.168.17.14 (Class C)

	1st Octet Range	2nd Octet Range	3rd Octet range	4th Octet Range
Range	(0-255)	(0-255)	(0-255)	(0-255)

Now, IPv4 supports five classes of IP ranges: Class A, Class B, Class C, Class D, and Class E. Among these, only A, B, and C are commonly used. Others have special uses. The reason for the classifications is that we can restrict our search for a machine in the haystack of about 4.3 billion machines (worst case scenario) in the network to a manageable filtered section of machines defined by the specific address class.

Class	First Octet Range	Network (N), Host (H)	Subnet Mask	No. of Networks	Hosts per Network
A	1-126	N.H.H.H.	255.0.0.0	126	16777214
B	128-191	N.N.H.H.	255.255.0.0	16382	65534
C	192-223	N.N.N.H.	255.255.255.0	2097150	254

D	224-239	Used for Multicasting.
E	240-254	Experimental; reserved for research purposes.

IP Address 127.x.x.x is reserved as a loopback address, termed *localhost*. IP Address 255.255.255.255 is used to broadcast to all hosts in the local area network. And, there are a few private IP addresses (not relevant at the point of our discussion).

Client and Server Sockets

Imagine a server is running in a machine. *Running* essentially means that the server goes into listening mode; in other words, cyclically checking if any data has arrived. Now, data arrives through a network channel, right? But, the type of data is defined by the protocol it adheres to. *Protocol* defines the norm followed in data packing. Some popular protocols are HTTP, FTP, SMTP, and so forth. (It is like a wrapper that determines its citizenship and how the data is to be treated by the server.) Recall that the server cyclically listens to the ***socket*** (remember, it is a virtual thing). The server may listen to many sockets. Each socket is uniquely identified by the IP address that helps in isolating the right machine in the network and the port number that determines the socket or right endpoint of the machine that server specifically listens to.

Understand the Difference

- **Unicast:** Message send between two machines in a network.
- **Broadcast:** Message send to all the machines in the network.
- **Multicast:** Message send to one or more of the machines in the network.

The TCP and UDP Protocols

There are two types of socket: a *connection-oriented socket* and a *connectionless socket*. A connection-oriented socket is also called a **stream socket.** In a stream socket, before data transmission a virtual one-to-one connection is established by a technique called *handshaking* and stream of data is sent uninterrupted throughout the virtual channel. In a *connectionless socket*, data is sent one packet at a time and no dedicated connection is established. It is also called a *datagram socket*. *Transmission Control Protocol (TCP)* is a transport layer protocol and is the most widely used protocols for connection-oriented sockets. *User Datagram Protocol (UDP)* is also a transport layer protocol but widely used for connectionless sockets.

In short, stream sockets are:

- A point-to-point dedicated channel between two hosts in the network
- Highly reliable communication

- Packets sent and received in a similar order
- Channels remains occupied even between pauses of transmission
- Long recovery time for data lost in transit
- Uses TCP protocol
  The datagram socket is:

- No dedicated channel
- Uses UDP Protocol
- May not be 100% reliable
- Data sent and received order is not the same
- Rapid recovery time for data lost in transit
  Now, if you have been wondering… **can we broadcast or multicast with TCP**? The answer, obviously, is No, because TCP is a protocol for establishing connection between two endpoints. The virtual channel created is dedicated until one of the hosts closes the connection. Thus, TCP establishes a reliable transport with a costly connection. When a data packet is send, it expects an acknowledgement. The acknowledgement determines that the message has been received at the other end properly; otherwise, the packet is re-transmitted. The channel remains occupied one-to-one all throughout this process. In the case of broadcast and multicast, there is no concept of reply and response. It creates a one-way traffic. Therefore, TCP's reliability of data transmission cannot be implemented on top of UDP and by the way, logically it is pointless. So, we use UDP for multicasting and broadcasting.

A Quick Example

The *MulticastSocket* class defined in the java.net package represents a multicast socket. Once a *MulticastSocket* object is created, the *joinGroup()* method is invoked to make it one of the members to receive a multicast message. Note that a multicast IP address is defined in the range of 224.0.0.0 to 239.255.255.255. Following is the IP multicast address ranges and uses.

Start Address	***End Address***	***Uses***
224.0.0.0	224.0.0.255	Reserved for special "well known" multicast addresses
224.0.1.0	238.255.255.255	Globally scoped (Internet-wide) multicast address
239.0.0.0	239.255.255.255	Administratively scoped (local) multicast addresses

```
import java.io.IOException;

import java.net.DatagramPacket;
```

```
import java.net.InetAddress;
import java.net.MulticastSocket;


public class UDPMulticastClient implements Runnable {


   public static void main(String[] args) {
      Thread t=new Thread(new UDPMulticastClient());
      t.start();
   }


   public void receiveUDPMessage(String ip, int port) throws
         IOException {
      byte[] buffer=new byte[1024];
      MulticastSocket socket=new MulticastSocket(4321);
      InetAddress group=InetAddress.getByName("230.0.0.0");
      socket.joinGroup(group);
      while(true){
         System.out.println("Waiting for multicast message...");
         DatagramPacket packet=new DatagramPacket(buffer,
            buffer.length);
         socket.receive(packet);
         String msg=new String(packet.getData(),
         packet.getOffset(),packet.getLength());
```

```
         System.out.println("[Multicast UDP message received]
            >> "+msg);
         if("OK".equals(msg)) {
            System.out.println("No more message. Exiting : "+msg);
            break;
         }
      }
      socket.leaveGroup(group);
      socket.close();
   }

   @Override
   public void run(){
   try {
      receiveUDPMessage("230.0.0.0", 4321);
   }catch(IOException ex){
      ex.printStackTrace();
   }
}

import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
```

```
import java.net.InetAddress;


public class UDPMulticastServer {


   public static void sendUDPMessage(String message,
   String ipAddress, int port) throws IOException {
      DatagramSocket socket = new DatagramSocket();
      InetAddress group = InetAddress.getByName(ipAddress);
      byte[] msg = message.getBytes();
      DatagramPacket packet = new DatagramPacket(msg, msg.length,
         group, port);
      socket.send(packet);
      socket.close();
   }


   public static void main(String[] args) throws IOException {
      sendUDPMessage("This is a multicast messge", "230.0.0.0",
         4321);
      sendUDPMessage("This is the second multicast messge",
         "230.0.0.0", 4321);
      sendUDPMessage("This is the third multicast messge",
         "230.0.0.0", 4321);
      sendUDPMessage("OK", "230.0.0.0", 4321);
```

```
    }

}
```

Multicasting Through Datagram Channels

Java supports multicasting through datagram channels through the class called *DatagramChannel* defined in the *java.nio.channels* package. A datagram channel that wants to receive multicast messages is joined to a multicast group. In this way, it becomes a member of the group to receive multicast messages. Once the connection is established, the datagram channel remains connected until it is disconnected or closed. Therefore, we can check the status of a datagram channel by invoking the *isConnected()* method, which return a Boolean *true* if the connection is open and *false* otherwise.

Here is a quick example to illustrate how to use datagram channels in multicasting.

A Quick Example

Using datagram channels has some additional advantages, such as we can opt to receive multicast datagrams only from selected sources and block others. For example, we can use the *block(InterAddress)* method from the *MembershipKey* class to block a multicast message source. There is a method, called *unblock(InetAddress)*, that unlocks the same. Here, we'll implement the simple multicast message sending and receiving scenario. Refer to the Java API documentation for additional API information.

```
package org.mano.example;

import java.io.IOException;

import java.net.*;

import java.nio.ByteBuffer;

import java.nio.channels.DatagramChannel;

import java.nio.channels.MembershipKey;



public class MulticastReceiver {

   private static final String MULTICAST_INTERFACE = "eth0";

   private static final int MULTICAST_PORT = 4321;
```

```
private static final String MULTICAST_IP = "230.0.0.0";

private String receiveMessage(String ip, String iface, int port)
      throws IOException {
   DatagramChannel datagramChannel = DatagramChannel
      .open(StandardProtocolFamily.INET);
   NetworkInterface networkInterface = NetworkInterface
      .getByName(iface);
   datagramChannel.setOption(StandardSocketOptions
      .SO_REUSEADDR, true);
   datagramChannel.bind(new InetSocketAddress(port));
   datagramChannel.setOption(StandardSocketOptions
      .IP_MULTICAST_IF, networkInterface);
   InetAddress inetAddress = InetAddress.getByName(ip);
   MembershipKey membershipKey = datagramChannel.join
      (inetAddress, networkInterface);
   System.out.println("Waiting for the message...");
   ByteBuffer byteBuffer = ByteBuffer.allocate(1024);
   datagramChannel.receive(byteBuffer);
   byteBuffer.flip();
   byte[] bytes = new byte[byteBuffer.limit()];
   byteBuffer.get(bytes, 0, byteBuffer.limit());
   membershipKey.drop();
   return new String(bytes);
```

```
   }
   public static void main(String[] args) throws IOException {
      MulticastReceiver mr = new MulticastReceiver();
      System.out.println("Message received : "
         + mr.receiveMessage(MULTICAST_IP,
         MULTICAST_INTERFACE, MULTICAST_PORT));
   }
}
```

```
package org.mano.example;
import java.io.IOException;
import java.net.InetSocketAddress;
import java.net.NetworkInterface;
import java.net.StandardSocketOptions;
import java.nio.ByteBuffer;
import java.nio.channels.DatagramChannel;
public class MulticastPublisher {
   private static final String MULTICAST_INTERFACE = "eth0";
   private static final int MULTICAST_PORT = 4321;
   private static final String MULTICAST_IP = "230.0.0.0";
   public void sendMessage(String ip, String iface, int port,
         String message) throws IOException {
```

```
      DatagramChannel datagramChannel=DatagramChannel.open();
      datagramChannel.bind(null);
      NetworkInterface networkInterface=NetworkInterface
         .getByName(iface);
      datagramChannel.setOption(StandardSocketOptions
         .IP_MULTICAST_IF,networkInterface);
      ByteBuffer byteBuffer=ByteBuffer.wrap
         (message.getBytes());
      InetSocketAddress inetSocketAddress=new
         InetSocketAddress(ip,port);
      datagramChannel.send(byteBuffer,inetSocketAddress);
   }
   public static void main(String[] args) throws IOException {
      MulticastPublisher mp=new MulticastPublisher();
      mp.sendMessage(MULTICAST_IP,MULTICAST_INTERFACE,
         MULTICAST_PORT,"Hi there!");
   }
}
```

Conclusion

Multicasting through Java is a part of the Java network programming paradigm. The communication between two remote hosts actually goes through several layers as defined the reference model or the TCP/IP model. The underlying principle of communication is complex. But, from the point of view of Java programming, it is made simple with the APIs supplied by the network library. Although UDP is termed as a not so reliable protocol, in practice it is not that unreliable, either. And, TCP is resource

hungry. Therefore, in most cases, UDP seems to be a better alternative. TCP can do almost everything that UDP can do—except that multicasting and broadcasting is possible only with UDP.