Multicasting in Java

1. Introduction

This article deals primarily with the subject of multicast communication in Java.

2. Sending multicast datagrams

In order to send any kind of datagram in Java, one needs a `java.net.DatagramSocket`:

```
DatagramSocket socket = new DatagramSocket();
```

This sample code creates the socket and a datagram to send and then simply sends the same datagram every second:

```
DatagramSocket socket = new DatagramSocket();

byte[] b = new byte[DGRAM_LENGTH];
DatagramPacket dgram;

dgram = new DatagramPacket(b, b.length,
  InetAddress.getByName(MCAST_ADDR), DEST_PORT);

System.err.println("Sending " + b.length + " bytes to " +
  dgram.getAddress() + ':' + dgram.getPort());
while(true) {
  System.err.print(".");
  socket.send(dgram);
  Thread.sleep(1000);
}
```

Valid values for the constants are:

- `DGRAM_LENGTH`: anything from 0 to 65507 (see section 5),
- `MCAST_ADDR`: any class D address (see appendix D), from 224.0.0.0 to 235.1.1.1,
- `DEST_PORT`: an unsigned 16-bit integer, eg. 6789 or 7777.

3. Receiving multicast datagrams

One can use a normal DatagramSocket to send and receive multicast datagrams as seen in the section 2. In order to receive multicast datagrams, however, one needs a MulticastSocket. The reason for this is simple, additional work needs to be done to control and receive multicast traffic by all the protocol layers below UDP.

The example given below, opens a multicast socket, binds it to a specific port and joins a specific multicast group:

```
byte[] b = new byte[BUFFER_LENGTH];
DatagramPacket dgram = new DatagramPacket(b, b.length);
```

```
MulticastSocket socket =
  new MulticastSocket(DEST_PORT); // must bind receive side
socket.joinGroup(InetAddress.getByName(MCAST_ADDR));

while(true) {
  socket.receive(dgram); // blocks until a datagram is received
  System.err.println("Received " + dgram.getLength() +
    " bytes from " + dgram.getAddress());
  dgram.setLength(b.length); // must reset length field!
}
```

Values for `DEST_PORT` and `MCAST_ADDR` must match those in the sending code for the listener to receive the datagrams sent there. `BUFFER_LENGTH` should be at least as long as the data we intend to receive. If `BUFFER_LENGTH` is shorter, the data will be **truncated silently** and `dgram.getLength()` will return `b.length`.

The `MulticastSocket.joinGroup()` method causes the lower protocol layers to be informed that we are interested in multicast traffic to a particular group address. One may execute `joinGroup()` many times to subscribe to different groups. If multiple MulticastSockets bind to the same port and join the same multicast group, they will all receive copies of multicast traffic sent to that group/port.

As with the sending side, one can re-use ones DatagramPacket and byte-array instances. The `receive()` method sets length to the amount of data received, so remember to reset the length field in the DatagramPacket before subsequent receives, otherwise you will be silently truncating all your incoming data to the length of the shortest datagram previously received.

One can set a timeout on the `receive()` operation using `socket.setSoTimeout(timeoutInMilliseconds)`. If the timeout is reached before a datagram is received, the `receive()` throws a `java.io.InterruptedIOException`. The socket is still valid and usable for sending and receiving if this happens.