Additional information on socket programming

Dr F. Belqasmi,
Industrial Research Post Doctoral Fellow, Ericsson Canada
Agenda

- Multithreading
- Blocking and Timeouts
- Multiple recipients
Multithreading

• Iterative servers handle clients sequentially, finishing with one client before servicing the next.
  – Work best for applications where the processing time of each client is small
  – Waiting time for subsequent clients may be unacceptable

• Multithreading allows a server to handle clients in parallel
Multithreading

- **Thread example**

```java
public class OneClientHandler implements Runnable {
    private Socket clntSock; // Socket connect to client
    public OneClientHandler(Socket clntSock, Logger logger) {
        this.clntSock = clntSock;
    }
    public static void handleClient(Socket clntSock) {
        try {
            // Get the input and output I/O streams from socket
            InputStream in = clntSock.getInputStream();
            OutputStream out = clntSock.getOutputStream();
            ........
            clntSock.close();
        } catch (IOException e) {
        }
    }
    public void run() {
        handleClient(clntSock);
    }
}
```
Multithreading

- **Multithreading Server example**

  ```java
  public class MultithreadingServer {
      public static void main(String[] args) throws IOException {
          ...
          int echoServPort = Integer.parseInt(args[0]); // Server port
          ServerSocket servSock = new ServerSocket(echoServPort);

          // Run forever, accepting and spawning a new client thread for each connection
          while (true) {
              Socket clntSock = servSock.accept(); // Spawn thread to handle new connection
              Thread thread = new Thread(new OneClientHandler(clntSock));
              thread.start();
          }
      }
  }
  ```
Blocking and Timeouts

• Socket calls may block
  – accept() method of ServerSocket() blocks until a connection is established
  – Socket constructor blocks until a connection is established
  – read() and receive() block if data is not available
  – write() blocks if no sufficient space in the output buffer

• A blocked method call makes the thread that is running it useless
  – E.g. waiting for lost datagrams
Blocking and Timeouts

• How to get around blocking calls
  – Set an upper-bound on the maximum time to block
    • Works for accept(), read() and receive()

```java
try{
    sock.setSoTimeout(timeBoundMillis);
    //serverSocket.setSoTimeout(timeBoundMillis);
    //datagramSocket.setSoTimeout(timeBoundMillis);
}
```

  – Use the available() method
    • Check for available data before calling read()

```java
InputStream in = clntSock.getInputStream();
if (in.available()){
    in.read(...);
}
```
Blocking and Timeouts

• Connecting a socket

```java
Try{
    InetAddress addr = InetAddress.getByName("java.sun.com");
    int port = 80;
    SocketAddress sockaddr = new InetSocketAddress(addr, port); //Create an unbound socket
    Socket sock = new Socket();
    int timeoutMillis = 2000; // 2 seconds
    sock.connect(sockaddr, timeoutMillis);
}catch (SocketTimeoutException ex){....}
```

• Writing to a socket
  – The amount of time that a write() may block is controlled by the receiving application
  – Currently, Java does not provide any way to cause a write() call to time out
Multiple recipients

- The information provided by the server may be of interest to multiple recipients
  - Unicast a copy of the data to each recipient
    - Inefficient (wastes bandwidth)
    - E.g,
      - The server sends 1Mbps streams
      - The network connection is 3Mbps
      - Only three simultaneous users can be supported
  - Networks provide a way to use bandwidth more efficiently
    - Packets are duplicated by the network (and not by the application) only when appropriate
    - 2 ways:
      - Broadcast
      - Multicast
Multiple recipients

• Broadcasting
  – Broadcasting UDP datagrams is similar to unicasting datagrams, except that a broadcast address is used instead of a regular (unicast) IP address
    • IPv4: 255.255.255.255
    • IPv6: FF02::1
  – All of the hosts on the same (local) broadcast network receive a copy of the message.
Multiple recipients

• Multicasting
  – A multicast address identifies a set of receivers
    • IPv4: addresses between 224.0.0.0 and 239.255.255.255
    • IPv6: any address starting with FF
Multiple recipients

• Multicasting example

```java
import java.net.MulticastSocket;
public class MulticastSender {
    public void sendMulticastMessage(String msg) {
        try{
            try{
                MulticastSocket mSocket = new MulticastSocket();
                mSocket.setTimeToLive(TTL); // Set TTL for all datagrams
            ....

                DatagramPacket message = new DatagramPacket(msg, msg.length,
                            multicastDestAddr, destPort);

                mSocket.send(message);
                mSocket.close();
            }catch (IOException ex){....}
        }
    }
}
```
Multiple recipients

- Multicasting example

```java
Try{
    MulticastSocket mSock = new MulticastSocket(port); // for receiving
    mSock.joinGroup(multicastAddress); // Join the multicast group

    // Receive a datagram
    DatagramPacket packet = new DatagramPacket(new byte[MAX_MSG_LENGTH], VoteMsgTextCoder.MAX_WIRE_LENGTH);
    sock.receive(packet);

    sock.close();
}
```
• References
  – “All About Sockets”
    http://java.sun.com/docs/books/tutorial/networking/sockets/