CMPU4021 Distributed Systems Lab Notes - Week 2

Networking in Java

Java Networking APIs

- Through the classes in `java.net`, Java programs can use TCP or UDP to communicate over the Internet

- The `URL, URLConnection, Socket, and ServerSocket` classes all use TCP to communicate over the network

- The `DatagramPacket, DatagramSocket, and MulticastSocket` classes are for use with UDP.

java.net

- Core package *java.net* provides the classes for implementing networking application

- Using these classes, the network programmer can communicate with any server on the Internet or implement his or her own Internet server.

Programmatic Access to Network Parameters

- Systems often run with multiple active network connections, such as wired Ethernet, 802.11 b/g (wireless), and bluetooth.
- Some applications might need to access this information to perform the particular network activity on a specific connection.
- The `java.net.NetworkInterface` class provides access to this information.

Network Interface

- A network interface is the point of interconnection between a computer and a private or public network.

- `NetworkInterface` is useful for a multi-homed system, which is a system with multiple NICs.

- Using `NetworkInterface`, you can specify which NIC to use for a particular network activity.

Class

java.net.InetAddress

- This class represents an Internet Protocol (IP) address. An IP address is either a 32-bit or 128-bit unsigned number used by IP.

- Static method *getByName* of this class uses DNS (Domain Name System) to return the Internet address of a specified host *name* as an *InetAddress* object.

  - `public static InetAddress` **`getByName`**`(String host) throws UnknownHostException`
    - Determines the IP address of a host, given the host's name. The host name can either be a machine name, such as "java.sun.com", or a textual representation of its IP address. If a literal IP address is supplied, only the validity of the address format is checked.

Class *InetAddress*

```
String host;
try {
   InetAddress address = InetAddress.getByName(host);
   System.out.println("IP address: " + address.toString());
}
catch (UnknownHostException e){
   System.out.println("Could not find " + host);
}
```

- For host name entered:
  lugh
  output
  IP address: lugh/147.252.224.73

Working with URLs in Java

- Java programs can use a class called `URL` in the java.net package to represent a URL address

- Creating an *absolute* URL
  - An absolute URL contains all of the information necessary to reach the resource in question.

```
URL myURL = new URL("http://example.com/");
```

Creating a URL Relative to Another

- A relative URL contains only enough information to reach the resource relative to (or in the context of) another URL

- You can create a URL object from a relative URL specification. For example, if you know two URLs at the site *example.com*:

```
http://example.com/pages/page1.html
http://example.com/pages/page2.html
```

- You can create URL objects for these pages relative to their common base URL: http://example.com/pages/ like this:

```
URL myURL = new URL("http://example.com/pages/");
URL page1URL = new URL(myURL, "page1.html");
URL page2URL = new URL(myURL, "page2.html");
```

Creating a URL Relative to Another

The general form of URL constructor is:

`URL(URL` *`baseURL`*`, String` *`relativeURL`*`)`

- The first argument is a URL object that specifies the base of the new URL.
- The second argument is a String that specifies the rest of the resource name relative to the base.
- If `baseURL` is null, then this constructor treats `relativeURL` like an absolute URL specification.
- If `relativeURL` is an absolute URL specification, then the constructor ignores `baseURL`.

Parsing a URL

- The `URL` class provides several methods that let you query URL objects.
- You can get the protocol, authority, host name, port number, path, query, filename, and reference from a URL using accessor methods (all are listed in Java docs)

`getProtocol`

Returns the protocol identifier component of the URL.

`getAuthority`

Returns the authority component of the URL.

`getHost`

Returns the host name component of the URL.

`getPort`

Returns the port number component of the URL. The getPort method returns an integer that is the port number. If the port is not set, getPort returns -1.

`getPath`

Returns the path component of this URL.

Transmission Control Protocol (TCP)

- A communication link created via TCP sockets is a **connection-oriented** link
  - the connection between server and client remains open throughout the duration of the dialogue between the two and is only broken when one end of the dialogue formally terminates the exchanges (via an agreed protocol).

- Since there are two separated types of process involved (client and server), we will examine them separately.

SETTING UP A TCP SERVER AND CLIENT

Setting up a TCP Server and Client TCP Sockets

- A communication link created via TCP sockets is a **connection-oriented** link.
  - The connection between server and client remains open throughout the duration of the dialogue between the two and is only broken when one end of the dialogue formally terminates the exchanges (via an agreed protocol).

- The example program implements a TCP Client, that connects to a TCP Server. The TCP Server receives data from and sends data to its clients.

Setting up a TCP server – Step 1

1. Create a ServerSocket object.
   - The *java.net.ServerSocket* class implements server sockets. A server socket waits for requests to come in over the network. It performs some operation based on that request, and then possibly returns a result to the requester.

```
ServerSocket servSock = new ServerSocket(1234);
```

- Above: the server waits (‘listens for’) a connection from a client on port 1234.

Setting up a TCP server – Step 2

2. Put the server into awaiting state.

   The server waits indefinitely for a client to connect. (Use the *java.net.Socket* class)

```
Socket link = servSock.accept();
```

Setting up a TCP server – Step 3

3. Set up input and output streams.

   Use *getInputStream* and *getOutputStream* of the *java.net.Socket class* to get references to streams associated with the socket set up in step 2.(Use java.io.BufferedReader and java.io.PrintWriter classes).

```
BufferedReader in =
  new BufferedReader(
    new InputStreamReader(link.getInputStream()));

PrintWriter out = new
    PrintWriter(link.getOutputStream(), true());
```

   The second argument (*true*) of the *PrintWriter* constructor causes the output buffer to be flushed for every *println* call.

Setting up a TCP server – Step 4

4. **Send and receive data.** Use the *BufferedReader readLine* method for receiving data and the *PrintWriter println* method for sending data.

   E.g.

```
out.println("Awaiting data…");
String inpuit = in.readLine();
```

Setting up a TCP server – Step 5

5. Close the connection (after completion of the dialogue). Use the class *Socket* method *close*.

   E.g.

   ```
   link.close()
   ```

Setting up a TCP client – Step 1

1. Establish a connection to the server. Use the class *java.net.Socket* following constructor:

   `Socket(InetAddress address, int port),`

   which creates a stream socket and connects it to the specified port number at the specified IP address.

   Note: The port number for server and client programs must be the same.

   ```
   Socket link = new Socket(host, 1234);
   ```

Setting up a TCP client – Step 2

2. Set up input and output streams.

   The same way as for the server.

Setting up a TCP client – Step 3

3. Send and receive data.

   The client's *BufferedReader* object will receive messages sent by the server's *PrintWriter* object.

   The client's *PrintWriter* object will send messages to be received by the *BufferedReader* object at the server end.

Setting up a TCP client – Step 4

4. Close the connection.

   Using the *java.net.Socket close* method.

```
link.close();
```

The Datagram Communication Protocol

- **Connectionless**
  - The connection between client and server is *not* maintained throughout the duration of the dialogue.
  - Instead, each datagram packet is sent as an isolated transmission whenever necessary.

SETTING UP A UDP SERVER AND CLIENT

Setting up a UDP Server and Client Introduction

- Connectionless
- The connection between client and server is not maintained throughout the duration of the dialogue.
- Instead, each datagram packet is sent as an isolated transmission whenever necessary.

Let's look at a simple example that illustrates how a server can continuously receives datagram packets over a datagram socket.

When the server receives a datagram packet, it replies by sending a datagram packet that contains a response back to the client

Setting up a UDP server – Step 1

1. Create a *java.net.DatagramSocket* object. The *DatagramSocket* class represents a socket for sending and receiving datagram packets. A datagram socket is the sending or receiving point for a packet delivery service. Each packet sent or received on a datagram socket is individually addressed and routed. Multiple packets sent from one machine to another may be routed differently, and may arrive in any order.

   E.g., the constructor

   ```
   DatagramSocket dgramSocket = new DatagramSocket(1234);
   ```

   constructs a datagram socket and binds it to the specified port (1234) on the local host machine.

Setting up a UDP server – Step 2

2. Create a buffer for incoming datagrams.

```
byte[] buffer = new byte[256]
```

Setting up a UDP server – Step 3

3. Create a *java.net.DatagramPacket* object for the incoming datagrams.

   E.g., the constructor

```
DatagramPacket inPacket = new
    DatagramPacket(buffer, buffer.length);
```

   constructs a *DatagramPacket* for receiving packets of length *length* in the previously created byte array (*buffer*).

Setting up a UDP server – Step 4

4. Accept an incoming datagram. Use the *receive* method of created *DatagramSocket* object.

```
public void receive(DatagramPacket p)
            throws IOException
```

E.g.

```
dgramSocket.receive(inPacket);
```

When this method returns, the DatagramPacket's buffer (*inPacket*) is filled with the data received. The datagram packet also contains the sender's IP address, and the port number on the sender's machine.

Setting up a UDP server – Step 5

5. Get the sender's address and port from the packet. Use the *getAddress* and *getPort* methods of created *DatagramObject*.

   E.g.,

```
InetAddress clientAddress = inPacket.getAddress();
int clientPort = inPacket.getPort();
```

Setting up a UDP server – Step 6

6. Retrieve the data from the buffer. The data will be retrieved as a *java.lang.String* using the constructor:

```
String(byte[] bytes, int offset, int length)
```

that constructs a new String by decoding the specified subarray of bytes using the platform's default charset.

E.g.

```
String message = new String(inPacket.getData(), 0,
                                    inPacket.getLength());
```

Setting up a UDP server – Step 7

7. **Create the response datagram**. Create a *DatagramPacket* object using the constructor:

```
DatagramPacket(byte[] buf, int length,
               InetAddress address, int port)
```

that constructs a datagram packet for sending packets of length *length* to the specified (client's) port number on the specified (client's) host. The first argument is returned by the *getBytes* method of the String class invoked on the retrieved message string.

E.g.

```
DatagramPacket outPacket = new DatagramPacket(
                                   response.getBytes(),
                                   response.length(),
                                   clientAddress,
                                   clientPort);
```

where the *response* is a *String* variable holding the return message.

Setting up a UDP server – Step 8

8. Send the response datagram. Call the method *send* of the *DatagramSocket object.*

```
public void send(DatagramPacket p)
        throws IOException
```

E.g.

```
dgramSocket.send(outPacket);
```

Setting up a UDP server – Step 9

9. Close the *DatagramSocket.*

Call method *close* of created *DatagramSocket* object.

E.g.

dgramSocket.close();

Setting up a UDP Client – Step 1

1. Create a *DatagramSocket* object.
   Important difference with the server code: the constructor that requires no argument is used, since a default port (at the client end) will be used.

   E.g.

```
DatagramSocket dgramSocket = newDatagramSocket();
```

Setting up a UDP Client – Step 2

2. Create the outgoing datagram. Exactly the same as for step 7 of the server program. E.g.

```
DatagramPacket outPacket = new DatagramPacket(
                                message.getBytes(,
                                message.length(),
                                host, PORT);
```

where, *message* is a *String* variable holding the required message.

Setting up a UDP Client – Step 3

3. Send the datagram message. Call method *send* of the *DatagramSocket* object, supplying the outgoing *DatagramPacket* object as an argument.

   E.g.

```
dgramSocket.send(outPacket);
```

Setting up a UDP Client – Step 4

4. Create a buffer for incoming datagrams.

```
byte[] buffer = new byte[256]
```

Setting up a UDP Client – Step 5

5. Create a *java.net.DatagramPacket* object for the incoming datagrams.

   E.g., the constructor

```
DatagramPacket inPacket = new
    DatagramPacket(buffer, buffer.length);
```

   constructs a *DatagramPacket* for receiving packets of length *length* in the previously created byte array (*buffer*).

Setting up a UDP Client – Step 6

6. Accept an incoming datagram.

   Use the *receive* method of created *DatagramSocket* object.

   E.g.

```
dgramSocket.receive(inPacket);
```

   When this method returns, the DatagramPacket's buffer (*inPacket*) is filled with the data received.

Setting up a UDP Client – Step 7

7. Retrieve the data from the buffer.

   The data will be retrieved as a *java.lang.String* using the constructor:

   E.g.

```
String response = new String(inPacket.getData(), 0,
                             inPacket.getLength());
```

Steps 2-7 may be repeated as many times as required.

Setting up a UDP Client – Step 8

8. Close the DatagramSocket.

```
dgramSocket.close();
```

Java API to IP multicast

Java API to IP multicast

- The Java API provides a datagram interface to IP multicast through the class MulticastSocket, which is a subclass of DatagramSocket with the additional capability of being able to join multicast groups.
- The class MulticastSocket provides two alternative constructors, allowing sockets to be created to use either a specified local port.
- A process can join a multicast group with a given multicast address by invoking the *joinGroup* method of its multicast socket.
- Effectively, the socket joins a multicast group at a given port and it will receive datagrams sent by processes on other computers to that group at that port.
- A process can leave a specified group by invoking the *leaveGroup* method of its multicast socket.

Using Multicast Sockets

- A simple extension to UDP sockets
- Sending application (Server):
  - creates a UDP socket:

```
socket = new DatagramSocket();
group = InetAddress.getByName("230.0.0.1");
```

  - sends the datagram to the multicast group

```
packet = new DatagramPacket(buffer,buffer.length,
                            group, port);
socket.send(packet);
```

  - Don’t really need to join the group to do this

`java.net.MulticastSocket`

- `java.net` includes a class called `MulticastSocket`

- This kind of socket is used on the client-side to listen for packets that the server broadcasts to multiple clients

java.net.MulticastSocket

- A *MulticastSocket* is a (UDP) *DatagramSocket*, with additional capabilities for joining "groups" of other multicast hosts on the internet

`public MultcastSocket()`

Creates socket at anonymous port number

`public MulticastSocket (int port)`

Create a multicast socket and bind it to a specific port.

`public void joinGroup (InetAddress a)`

Joins a multicast group.

`public void leaveGroup (InetAddress a)`

Leave a multicast group

Using `MulticastSocket`(I)

- Receiving application (Client):
  - creates a *MulticastSocket*

```
socket = new MulticastSocket(port);
```

  - binds an address (including a port number) to the socket.

```
address = InetAddress.getByName("230.0.0.1");
```

  - joins the multicast group (performs a *joinGroup*())

```
socket.joinGroup(address);
```

  - then call *receive*()

```
byte[] buffer = new byte[32];
packet = new DatagramPacket(buffer,buffer.length);
socket.receive(packet);
```

  - once done, do a *leaveGroup*() and a *close*()

References

- Chapters 1 and 2, Introduction to Network Programming in Java by Jan Graba

- https://docs.oracle.com/javase/tutorial/networking