

CMPU4021 Distributed Systems Lab Notes - Week 2

Introduction To Network Programming in Java

Ports and Sockets

- Closely associated with the hardware communication links between computers within a network, ports and sockets, but they are not themselves hardware elements.
- Abstract concepts that allow the programmer to make use of those communication links.

Ports (I)

- A port is a *logical* connection to a computer (as opposed to a physical connection) and is identified by a number in the range 1 to 65 535.
 - This number has no correspondence with the number of physical connections to the computer.
- There may be just one physical connection, even though the number of ports used on that machine may be much greater than this.
- Ports are implemented upon all computers attached to a network, but the network programmer will refer explicitly to port numbers only for those machines that have server programs running on them.
- Each port may be dedicated to a particular server/service.

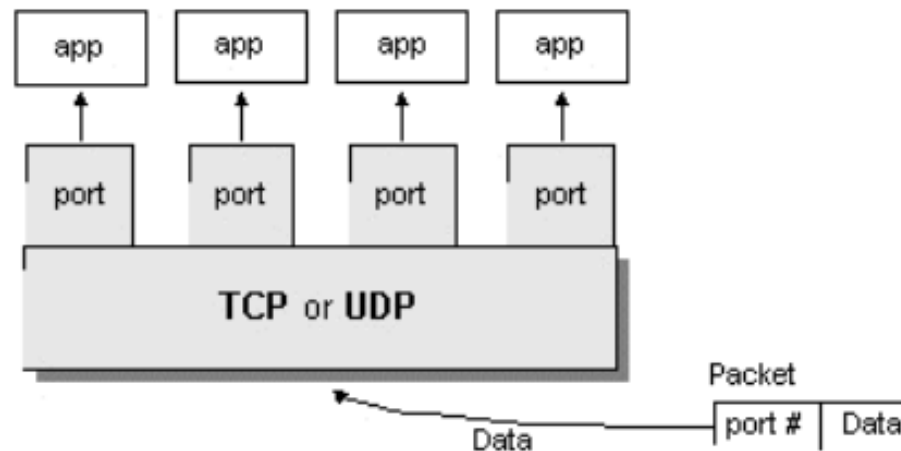
Ports (II)

- The number of available ports will normally greatly exceed the number that is actually used.
- 1 to 1023 port range normally set aside for the use of specified standard services ('well-known' services).
 - E.g. port 80 is normally used by Web servers.
- Application programs wishing to use ports for non-standard services should use a range of 1024 to 65 535.

Protocol Name	Port Number	Nature of Service
Echo	7	The server echoes the data sent to it.
Daytime	13	Provides the ASCII representation of the current date and time on the server.
FTP-data	20	Transferring files
FTP	21	Sending FTP commands like PUT and GET
Telnet	23	Remote login and command line interaction
SMTP	25	E-mail
HTTP	80	The World Wide Web protocol
NNTP	119	Usenet

Ports (III)

- For each port supplying a service, there is a server program waiting for any requests. Server programs run in parallel on the host machine.
- When a client attempts to make connection with a particular server program, it supplies the port number of the associated service.
- The host machine examines the port number and passes the client's transmission to the appropriate server program for processing.



Sockets

- Normally, there are multiple clients wanting the same service at the same time.
 - E.g., multiple browsers wanting Web pages from the same server.
- The server needs some way of distinguishing between clients and keeping their dialogues separate. This is achieved via the use of sockets.
- A *socket* is an abstract concept and *not* an element of computer hardware.
- It is used to indicate one of the two end points of a communication link between two processes.

Sockets (I)

- When a client wishes to make connection to a server, it will create a socket at its end of the communication link.
- Upon receiving the client's initial request (on a particular port number), the server will create a new socket at its end that will be dedicated to communication with that particular client.
- Just as one hardware link to a server may be associated with many ports, so too may one port be associated with many sockets.

The Internet and IP Addresses (I)

- Each computer on the Internet has a unique IP address
- IPv4 - IP version that represents machine address in so called **quad notation** (four eight-bit numbers).
 - E.g. 131.111.111.244
- Shortage of IPv4 addresses
 - IPv4 is being replaced with IPv6
 - IPv6 – uses 128-bit addresses

Decimal representation of Internet addresses

	octet 1	octet 2	octet 3		Range of addresses
	Network ID		Host ID		
Class A:	1 to 127	0 to 255	0 to 255	0 to 255	1.0.0.0 to 127.255.255.255
	Network ID		Host ID		
Class B:	128 to 191	0 to 255	0 to 255	0 to 255	128.0.0.0 to 191.255.255.255
	Network ID		Host ID		
Class C:	192 to 223	0 to 255	0 to 255	1 to 254	192.0.0.0 to 223.255.255.255
	Multicast address				
Class D (multicast):	224 to 239	0 to 255	0 to 255	1 to 254	224.0.0.0 to 239.255.255.255
Class E (reserved):	240 to 255	0 to 255	0 to 255	1 to 254	240.0.0.0 to 255.255.255.255

Internet Services, URLs and DNS

- **Protocol**

- governs the communication that takes place between server and client
- Each end of the dialogue must know what may/must be sent to the other, the format in which it should be sent, the sequence in which it must be sent (if sequence matters) and for 'open-ended' dialogues – how the dialogue is to be terminated.

Uniform Resource Locator (URL)

- Unique identifier for any resource located on the Internet
- Structure:
`<protocol>://<hostname>[:port>]/[<pathname>][/<filename>[<section>]]`
- Items in square brackets are optional.
Example:
`http://www.oracle.com/technetwork/java/index.html`
- Port may be omitted – for a well-known protocol, and the default port will be assumed.
- If the file name is omitted, then the sever sends default file from the directory specified in the path name. This default file is commonly called *index.html* or *default.html*.
- The 'section' part of the URL (not often specified, a name preceded by '#') indicates a named 'anchor' in an HTML document.

Domain Name System (DNS)

- A **domain name**, or **host name**, is the user-friendly equivalent of an IP address.
 - In the previous example, the domain name was www.oracle.com.
- The individual parts of a domain name do not correspond to the individual parts of an IP address. In fact, domain names do not always have four parts (as IPv4 addresses must have).
- The Domain Name provides a mapping between IP addresses and domain names and is held in a distributed database.
- The IP address systems and the DNS are governed by ICANN (Internet Corporation for Assigned Names and Numbers) – non-profit making organization.
- When a URL is submitted to a browser, the DNS automatically converts the domain name part into its numeric IP equivalent.

Networking Classes in the JDK

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

Let's look at a simple example that illustrates how a program can establish a connection to a server program using the Socket class and then, how clients can send data to and receive data from the server through the socket.

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 input = 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();
```

Multicast Networking

Multicast

- One-to-many or many-to-many distribution
- In computer networking
 - Group communication where information is addressed to a group of destination computers simultaneously
- Group communication, either
 - *application layer multicast*
 - *network assisted multicast*
 - makes it possible for the source to efficiently send to the group in a single transmission
- Network assisted multicast
 - May be implemented at the Internet layer using IP multicast

IP Multicast

- Built on top the Internet Protocol
- An implementation of group communication
- IP packets are addressed to computers
- IP multicast allows the sender to transmit a single IP packet to a set of computers that form a multicast group.

IP Multicast

- Multicast addresses may be permanent or temporary.
- Permanent groups exist even when there are no members – their addresses are assigned by IANA
- The remainder of the multicast addresses are available for use by temporary groups, which must be created before use and cease to exist when all the members have left.
- When a temporary group is created, it requires a free multicast address to avoid accidental participation in an existing group.

Multicasting

- Broader than *unicast* (one sender and one receiver, point-to-point communication) but narrower and more targeted than broadcast communication.
- Sends data from one host to many different hosts, but not to everyone;
 - the data only goes to clients that have expressed an interest by joining a particular multicast group.
- Used for ‘public meetings’ on the Internet
 - a multicast socket sends a copy of the data to a location close to the parties that have declared an interest in the data;
 - the data is duplicated only when it reaches the local network serving the interested clients;
 - the data crosses the Internet only once.)

Multicasting (I)

- Most multicast data is audio or video or both (relatively large and robust against data loss).
- Multicast data is sent via UDP (unreliable – but can be as much as three times faster than data sent via connection-oriented TCP).
- A multicast group is specified by a class D IP address and by a standard UDP port number.
- Class D IP addresses are in the range 224.0.0.0 to 239.255.255.255, inclusive.
- The address 224.0.0.0 is reserved and should not be used (the address of the Network Time Protocol distributed service – assigned the name ***ntp.mcast.net***)

Multicasting (II)

- Better than broadcast where everyone receives traffic
 - And broadcasting is limited.
- With multicasting, data duplicated only where necessary
- Routers typically form trees to efficiently distribute information
- The IP multicast protocol is relying on a few routers that support IP multicast routing to determine the routes, but transmitting IP packets through other routers using standard IP addresses.

Multicast Group

- A set of Internet hosts that share a multicast address.
- Any data sent to the multicast address is relayed to all the members of the group. Membership in a multicast group is open; hosts can enter or leave the group at any time.
- Groups can be either permanent or transient:
 - Permanent groups have assigned addresses that remain constant, whether or not there are any members in the group; typically given names
 - Most multicast groups are transient – exist only as long as they have members (in the 255 -- 238 range).

Special Multicast Addresses

- 224.0.0.0: reserved
- 224.0.0.1 (*all-hosts group*): contains all multicast-capable hosts on this subnet
- 224.0.0.2 (*all-routers group*): contains all multicast routers on this subnet
- 224.0.0.3: unassigned

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 MulticastSocket()
```

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 (Third Edition)
- <https://docs.oracle.com/javase/tutorial/networking>