# Networking

Mārtiņš Leitass

lattelecom

# Networking

This section describes networking concepts.

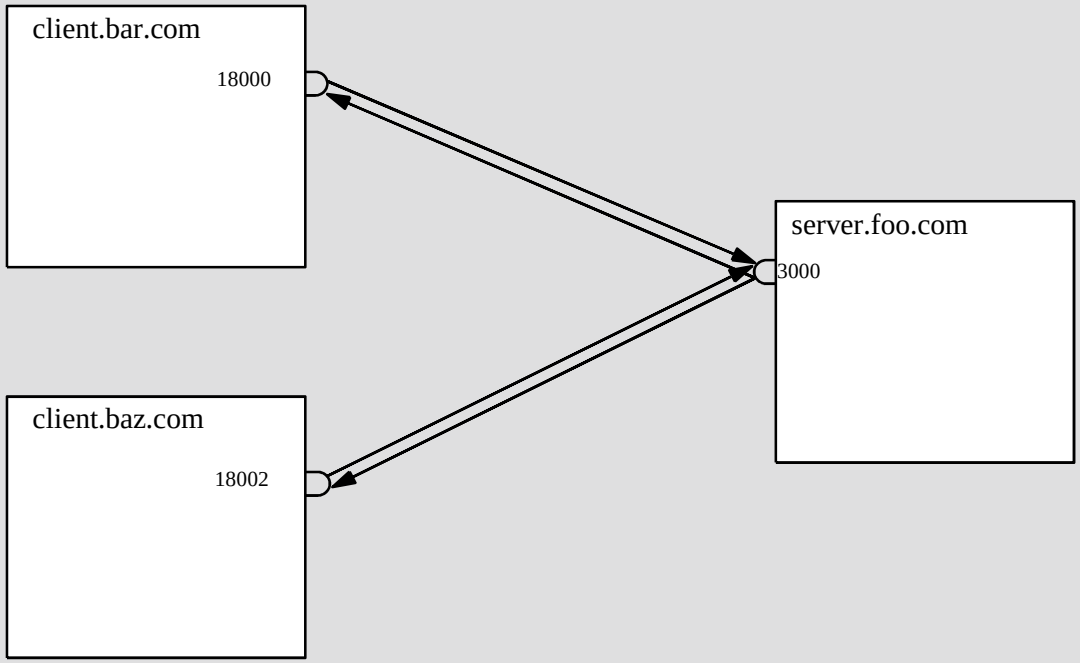
## Sockets

- Sockets hold two streams: an input stream and an output stream.
- Each end of the socket has a pair of streams.

## Setting Up the Connection

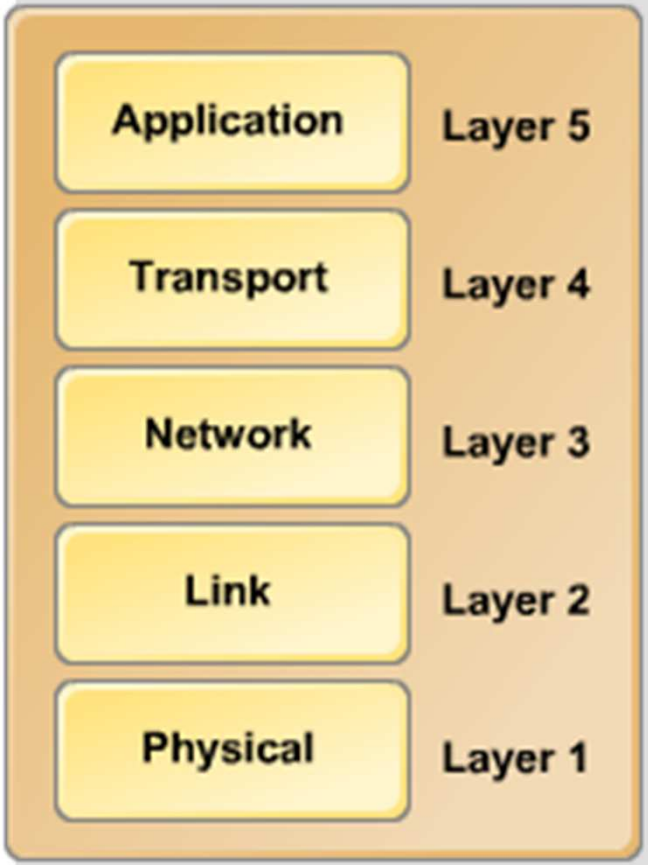
Set up of a network connection is similar to a telephone system: One end must *dial* the other end, which must be *listening*.

# Networking



# The Internet Protocol Stack

- Internet Protocol stack is a series of networking layers




| Layer | Responsibilities |
|---|---|
| Application | This layer is responsible for processing the information sent between two computers (based on protocols such as HTTP or FTP |
| Transport | Responsible for maintaining a reliable connection between the two computers (compensates lost info by re-sending or fixing) |
| Network | Responsible for determining the entire route of a network message. |
| Link | Responsible for coordinating the transfer of bits from one computer to other by keeping track of bits sent and recieved |
| Physical | Responsible for moving individual bits from one computer to other. Physical connection such as wire or wireless is using at this layer |

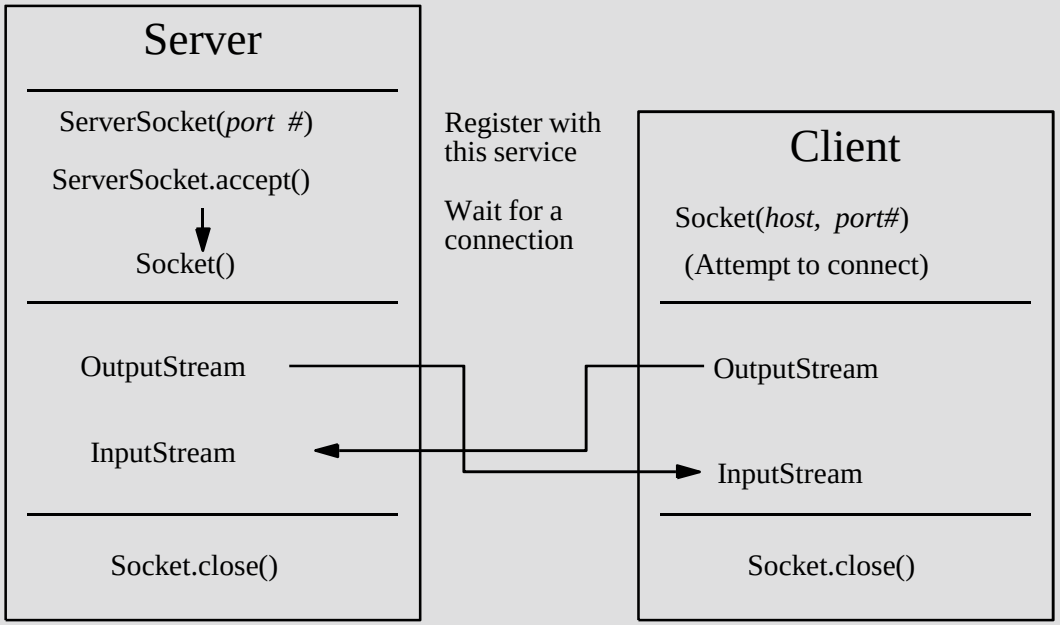
# Transport Protocols

| | UDP | TCP | DCCP | SCTP |
|---|---|---|---|---|
| Packet header size | 8 Bytes | 20 Bytes | Varies | 8 Bytes + Variable Chunk Header |
| Transport layer packet entity | Datagram | Segment | Datagram | Datagram |
| Port numbering | Yes | Yes | Yes | Yes |
| Error detection | Optional | Yes | Yes | Yes |
| Reliability: Error recovery by automatic repeat request (ARQ) | No | Yes | No | Yes |
| Flow control | No | Yes | Yes | Yes |
| Multiple streams | No | No | No | Yes |

# Java Network Basic

- Stream Socket
  - Connection Service
  - Use Transmission Control Protocol (TCP)
  - Need Connection Time Before Communication
- Datagram Socket
  - Connectionless Service
  - Use User Datagram Protocol (UDP)
  - Transmit Each Packet Independently
  - Don’t Need Connection Setup
  - Need Complement of Reliability
- Server and Client

# Java Networking Model



# java.net Package

- provides convenient access to application layer, transport layer, and internet layer
- The most relevant classes and methods:
  - URL
  - URLConnection (interface) / HttpURLConnection (subclass)
  - URLEncoder / URLDecoder
  - InetAddress - IP addresses (DNS lookup, etc.)
  - Socket,ServerSocket - TCP sockets (as in example client and server)
  - DatagramSocket – UDP socket
  - DatagramPacket – UDP datagram

# URL Class

- URL has two main components:
  - Protocol identifier (http/udp...)
  - Resource name
- Class URL represents a Uniform Resource Locator, a pointer to a "resource" on the World Wide Web.
  - A resource can be something as simple as a file or a directory, or it can be a reference to a more complicated object, such as a query to a database or to a search engine
  - Name of resource consists of host name, file name, port number, and reference
- URL Objects
  - URL u-aizu = new URL("http://www.u-aizu.ac.jp/");
  - URL location = new URL(u-aizu, "location/welcome.html");
  - URL software = new URL(u-aizu, "welcome/software.html");
  - URL hardware = new URL(u-aizu, "welcome/hardware.html");

# URL class methods

- **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.
- **getQuery -** Returns the query component of this URL.
- **getFile -** Returns the filename component of the URL. The getFile method returns the same as getPath, plus the concatenation of the value of getQuery, if any.
- **getRef -** Returns the reference component of the URL.

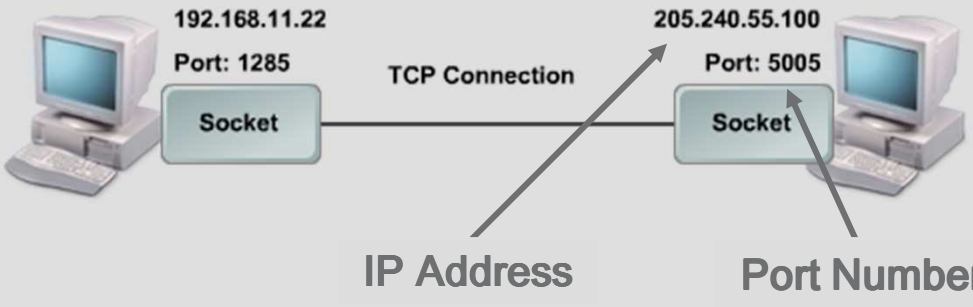
# URL example

http://java.sun.com:80/tutor/index.html?name=networking#DOWNLOADING

| Field | Value |
|---|---|
| protocol | http |
| authority | java.sun.com:80 |
| host | java.sun.com |
| port | 80 |
| path | /tutor/index.html |
| query | name=networking |
| filename | /tutor/index.html?name=networking |
| ref | DOWNLOADING |

# Sockets

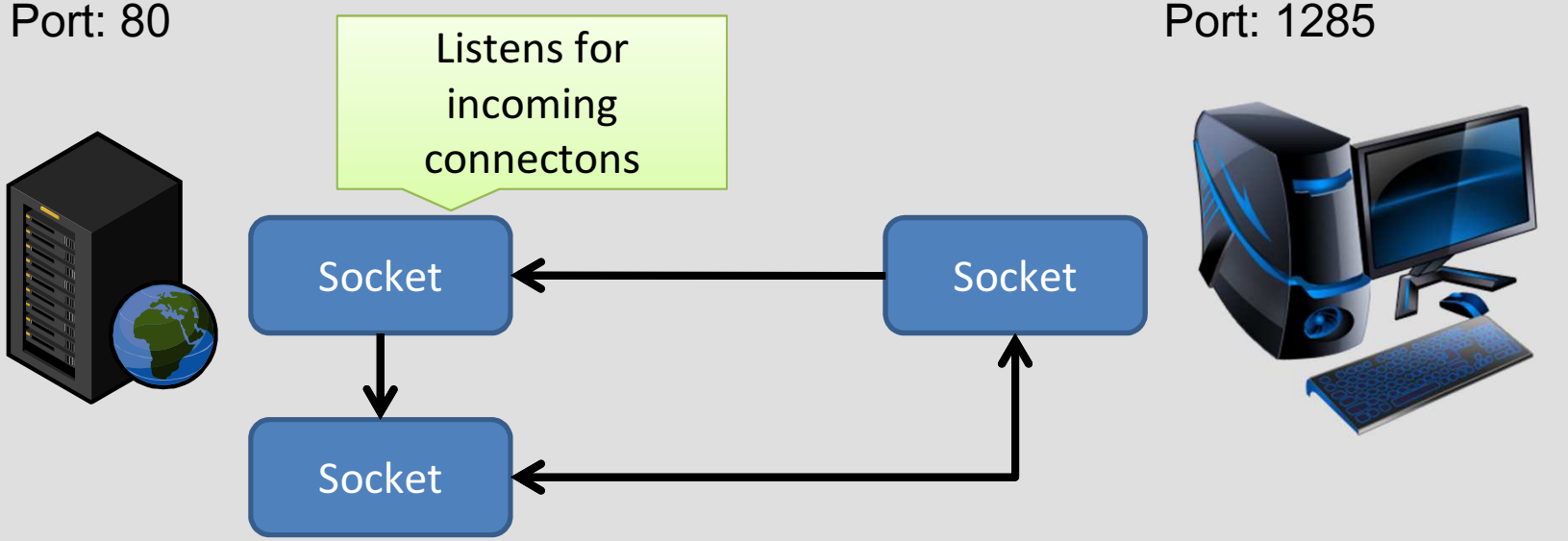
- A socket is a connection between two hosts. It can perform seven basic operations:
  - Connect to a remote machine
  - Send data
  - Receive data
  - Close a connection
  - Bind to a port
  - Listen for incoming data
  - Accept connections from remote machines on the bound port
- Each socket is identifiable by its IP address and port number



# Establishing a TCP Connection

Server
Host: 205.240.55.100
Port: 80

Client
Host: 192.168.11.22
Port: 1285



# The Socket Class

- The java.net.Socket class is Java's fundamental class for performing client-side TCP operations
- Constructors
  - Socket(String host, int port)
  - Socket(InetAddress host, int port)
  - Socket(String host, int port, InetAddress interface, int localPort)
- Methods
  - getInetAddress( )
  - getPort( )
  - getLocalAddress( )
  - getLocalPort( )
  - getInputStream( )
  - getOutputStream( )
  - close( )

# The ServerSocket Class

- The ServerSocket class contains everything needed to write servers in Java
- Constructors:
  - public ServerSocket(int port) throws BindException, IOException
  - public ServerSocket(int port, int queueLength) throws BindException, IOException
  - public ServerSocket( ) throws IOException
- Additional methods:
  - accept()
  - public void setPerformancePreferences(int connectionTime, int latency, int bandwidth)

# ServerSocket class

- In Java, the basic life cycle of a server program is:
  1. A new ServerSocket is created on a particular port using a ServerSocket( ) constructor.
  2. The ServerSocket listens for incoming connection attempts on that port using its accept( ) method. accept( ) blocks until a client attempts to make a connection, at which point accept( ) returns a Socket object connecting the client and the server.
  3. Depending on the type of server, either the Socket's getInputStream() method, getOutputStream( ) method, or both are called to get input and output streams that communicate with the client.
  4. The server and the client interact according to an agreed-upon protocol until it is time to close the connection.
  5. The server, the client, or both close the connection.
  6. The server returns to step 2 and waits for the next connection.

# Minimal TCP/IP Server

```
1   import java.net.*;
2   import java.io.*;
3
4   public class SimpleServer {
5      public static void main(String args[]) {
6         ServerSocket s = null;
7
8         // Register your service on port 5432
9         try {
10           s = new  ServerSocket(5432);
11        } catch (IOException e) {
12           e.printStackTrace();
13        }
```

# Minimal TCP/IP Server

```
14
15          // Run the listen/accept loop forever
16          while (true) {
17              try {
18                  // Wait here and listen for a connection
19                  Socket s1 = s.accept();
20
21                  // Get output stream associated with the socket
22                  OutputStream s1out = s1.getOutputStream();
23                  BufferedWriter bw = new BufferedWriter(
24                      new OutputStreamWriter(s1out));
25
26                  // Send your string!
27                  bw.write("Hello Net World!\n");
```

# Minimal TCP/IP Server

```
28
29              // Close the connection, but not the server socket
30              bw.close();
31              s1.close();
32
33            } catch (IOException e) {
34               e.printStackTrace();
35            } // END of try-catch
36
37         } // END of while(true)
38
39      } // END of main method
40
41   } // END of SimpleServer program
```

# Minimal TCP/IP Client

```
1   import java.net.*;
2   import java.io.*;
3
4   public class SimpleClient {
5
6       public static void main(String args[]) {
7
8           try {
9               // Open your connection to a server, at port 5432
10              // localhost used here
11              Socket s1 = new Socket("127.0.0.1", 5432);
12
13              // Get an input stream from the socket
14              InputStream is = s1.getInputStream();
15              // Decorate it with a "data" input stream
16              DataInputStream dis = new DataInputStream(is);
```

# Minimal TCP/IP Client

```
17
18              // Read the input and print it to the screen
19              System.out.println(dis.readUTF());
20
21              // When done, just close the steam and connection
22              dis.close();
23              s1.close();
24
25          } catch (ConnectException connExc) {
26              System.err.println("Could not connect.");
27
28          } catch (IOException e) {
29              // ignore
30          } // END of try-catch
31
32      } // END of main method
33
34  } // END of SimpleClient program
```

# What are datagrams?

- Datagrams are discrete packets of data
- Each is like a parcel that can be addressed and sent to an recipient anywhere on the Internet
- This is abstracted as the User Datagram Protocol (UDP) in RFC768 (August 1980)
- Most networks cannot guarantee reliable delivery of datagrams

# Why use datagrams?

- Good for sending data that can naturally be divided into small chunks
- Poor for (lossless) stream based communications
- Makes economical use of network bandwidth (up to 3 times the efficiency of TCP/IP for small messages)
- Datagrams can be locally broadcast or multicast (one-to-many communication)

# java.net.DatagramPacket (1)

- DatagramPackets normally used as short lived *envelopes* for datagram messages:
  - Used to assemble messages before they are dispatched onto the network,
  - or dismantle messages after they have been received
- Has the following attributes:
  - Destination/source address
  - Destination/source port number
  - Data bytes constituting the message
  - Length of message data bytes

# java.net.DatagramPacket (2)

- Construction:
  - `DatagramPacket(byte[] data, int length)`
- Some useful methods:
  - `void setAddress(InetAddress addr)`
  - `InetAddress getAddress()`
  - `void setPort(int port)`
  - `int getPort()`
- DatagramPackets are *not* immutable so, in principle you can reuse then, but . .
- Experience has shown that they often misbehave when you do -- create a new one, use it once, throw it away!

# java.net.DatagramSocket (1)

- Used to represent a socket associated with a specific port on the local host
- Used to send *or* receive datagrams
- Note: there is no counterpart to `java.net.ServerSocket`! Just use a DatagramSocket with a *agreed* port number so others know which address and port to send their datagrams to

# java.net.DatagramSocket (2)

- Construction:
  - `DatagramSocket(int port)`
    - Uses a specified port (used for receiving datagrams)
  - `DatagramSocket()`
    - Allocate any available port number (for sending)
- Some useful methods:
  - `void send(DatagramPacket fullPacket)`
    - Sends the full datagram out onto the network
  - `void receive(DatagramPacket emptyPacket)`
    - Waits until a datagram and fills in emptyPacket with the message
- `. . .` and a few more in the Javadoc

# sea.datagram.DatagramSender

- This example sends datagrams to a specific host (anywhere on the Internet)
- The steps are as follows:
  - Create a new DatagramPacket
  - Put some data which constitutes your message in the new DatagramPacket
  - Set a destination address and port so that the network knows where to deliver the datagram
  - Create a socket with a *dynamically allocated* port number (if you are just sending from it)
  - Send the packet through the socket onto the network

# sea.datagram.DatagramSender

```
byte[] data = “This is the message”.getBytes();
DatagramPacket packet =
      new DatagramPacket(data, data.length);

// Create an address
InetAddress destAddress =
      InetAddress.getByName(“fred.domain.com”);
packet.setAddress(destAddress);
packet.setPort(9876);

DatagramSocket socket = new DatagramSocket();
socket.send(packet);
```

# sea.datagram.DatagramReceiver

- The steps are the reserve of sending:
  - Create an empty DatagramPacket (and allocate a buffer for the incoming data)
  - Create a DatagramSocket on an *agreed* socket number to provide access to arrivals
  - Use the socket to receive the datagram (the thread will block until a new datagram arrrives)
  - Extract the data bytes which make up the message

# sea.datagram.DatagramReceiver

```
// Create an empty packet with some buffer space
byte[] data = new byte[1500];
DatagramPacket packet =
      new DatagramPacket(data, data.length);

DatagramSocket socket = new DatagramSocket(9876);

// This call will block until a datagram arrives
socket.receive(packet);

// Convert the bytes back into a String and print
String message =
   new String(packet.getData(), 0, packet.getLength());
System.out.println("message is " + message);
System.out.println("from " + packet.getAddress());
```

# IP Addresses and Java

- Java has a class `java.net.InetAddress` which abstracts network addresses
- Serves three main purposes:
  - Encapsulates an address
  - Performs name lookup (converting a host name into an IP address)
  - Performs reverse lookup (converting the address into a host name)

# java.net.InetAddress (1)

- Abstraction of a network address
- Currently uses IPv4 (a 32 bit address)
- Will support other address formats in future
- Allows an address to be obtained from a host name and vice versa
- Is immutable (is a *read-only* object)
  - Create an InetAddress object with the address you need and throw it away when you have finished

# java.net.InetAddress (2)

- Static construction using a factory method
  - `InetAddress getByName(String hostName)`
    - `hostName` can be "host.domain.com.au", or
    - `hostName` can be "130.95.72.134"
  - `InetAddress getLocalHost()`
- Some useful methods:
  - `String getHostName()`
    - Gives you the host name (for example "www.sun.com")
  - `String getHostAddress()`
    - Gives you the address (for example "192.18.97.241")
  - `InetAddress getLocalHost()`
  - `InetAddress[] getAllByName(String hostName)`