Networking

COMP 112 2018

School of Engineering and Computer Science

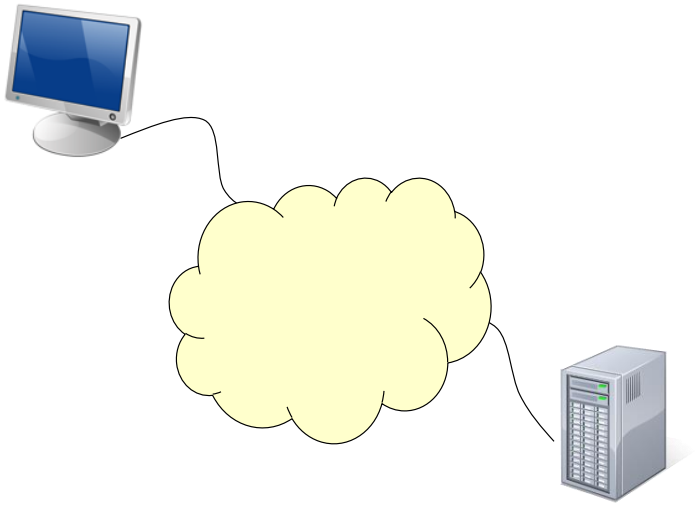
Victoria University of Wellington

Networking

- Computers are (mostly) networked
  ie, connected to other computers:
  - desktops,
  - laptops
  - embedded controllers
  - servers
  - phones
  - sensors
- Users expect to be able communicate with the world
- Computers may require networking to operate
- How does it work?
- How can we write programs that use the network?

What's required to communicate?

http://www.geonet.org.nz/quakes/region/wellington/2014p240655



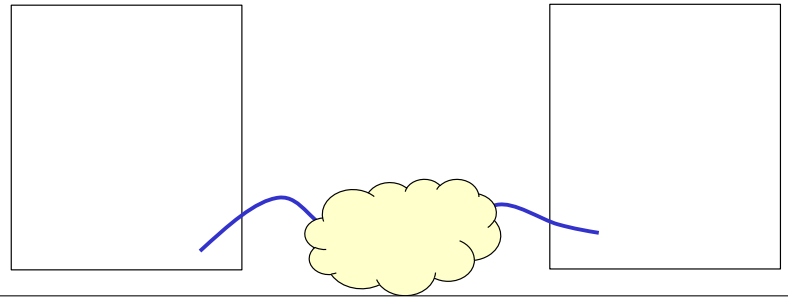
Levels of communication

- transmitting a character over the wire, fibre, or wireless
- chunking a message into "packets"
- ensuring that a packet is not corrupted or lost
- dealing with multiple packets from multiple computers on the same wire at the same time
- specifying where a packet should go to
- working out how to get a packet to its destination
- ensuring that a collection of packets making a message all arrive in order.
- specifying a message in a way that the recipient(s) will understand
- managing a "conversation" with a distant server.
- keeping track of multiple connections with multiple computers

TCP/IP Stack

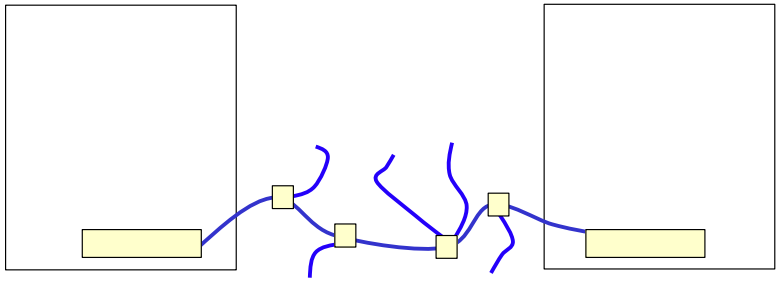
- Layers of programs that build on each other.
  - Application Layer
  - Transport Layer
  - Internet Layer
  - Network Access Layer

Each layer treats the layer below it as a black box which provides an abstraction that hides the lower level details.



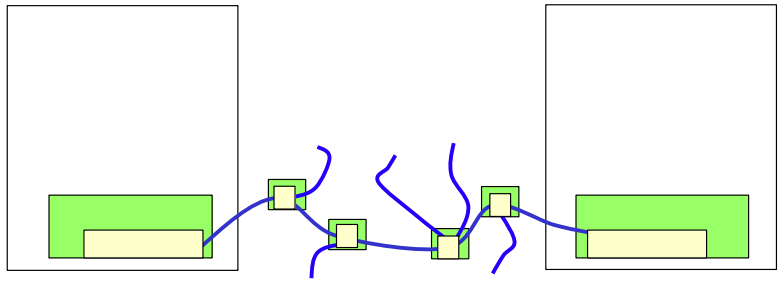
TCP/IP Stack

- Network Access Layer (eg, Ethernet protocol)
  - Part of the operating system
  - Encoding a packet into signals to go over a wire/fibre/wireless to the computer at the other end of the wire, and decoding it.
  - Dealing with collisions



TCP/IP Stack

- Internet Layer (eg, IP protocol)
  - Part of operating system
  - Encoding information into a packet with address information
  - Dealing with addresses and routing a packet to the target computer at the other side of the network.
- Network Access Layer

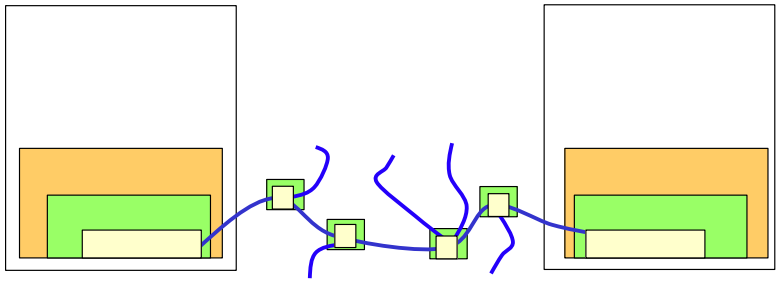


TCP/IP Stack

- Transport Layer (eg TCP protocol)
  - Part of operating system
  - Ensuring sequence of packets in a message all arrive in order
  - Protocol to acknowledge, recover, resend, reorder.

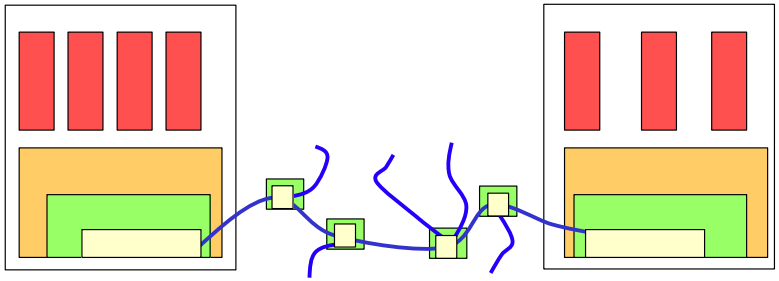
There are other protocols that work differently

- Internet Layer
- Network Access Layer



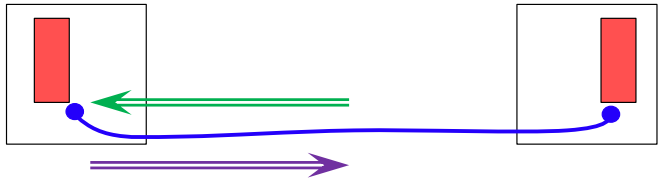
TCP/IP Stack

- Application Layer
  - Programs that need to communicate
  - Clients and Servers
- Transport Layer
- Internet Layer
- Network Access Layer



Application Layer view

- Application wants a communication channel with a program running on a target computer over the network.



- TCP (Transport Layer) provides “Sockets”:
  - A socket is one end of a TCP connection
  - Has two streams associated with it:
    - Input stream (from the other end of the connection) Application can read from it
    - Output stream (to the other end of the connection) Application can write/print to it

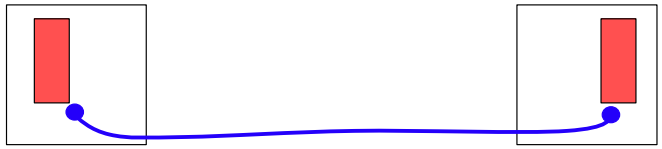
Using sockets in Java

- Easy!
  - Streams are sequences of characters
  - Like files, or UI text pane, or System.in/System.out

```
try {
    Socket  socket =  get a socket somehow ;
    Scanner inputFromTarget = new Scanner(socket.getInputStream());
    PrintStream outputToTarget = new PrintStream(socket.getOutputStream());
    while (inputFromTarget.hasNext()){
        String line = inputFromTarget.nextLine();
        String response =  work out a response ;
        outputToTarget.println(response);
        outputToTarget.flush();
    }
} catch  (IOException e){System.out.println(“connection failure ”+e); }
```


Getting a Socket

- Connection requires each computer to set up a socket connected to the other



- How do you specify where you want the connection to go to?
  - Which program running on which computer is going to serve you a web page?
- Which program goes first?
  - Need a connection at the other end in order to set up the one at this end!

Addressing a connection

- Two parts to the address:
  - host computer: IP address, or Host name
  - "port" (an integer)
- Each computer on the network must have an IP address.
  130.195.5.9 or "debretts.ecs.vuw.ac.nz"
- Operating system provides a set of "ports" that connections can be associated with.
  - Many programs have a standard port number.

- Application can specify a remote host and a port.
  - Can hope the program at the other end will listen

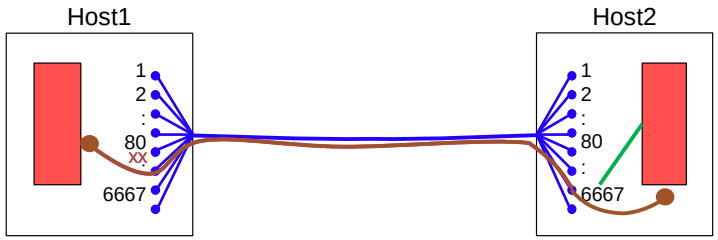
Who goes first?

- If application requests connection to a port on a remote host, but there is no application on that port at the remote host that will respond to the request, connection will fail.
- Need to have an application on remote host listening to the port:

Client – Server model:

- If host provides some service:
  - Server application must be running on remote host
  - Server application must be listening on a well-known port
  - Client application can request a connection to host : port
  - Server application will hear, and can accept the request
  - Operating systems will set up a new socket on each end with a connection between them.

Addressing a connection



Request connection to port 6667 on Host2 via port *xx*

OS creates socket connected to port 6667 on Host2

Listening on port 6667

Application accepts connection

OS creates socket connected to port *xx* on Host1

Application is still listening!

Setting up the sockets

Server

```
try {
    ServerSocket serverSocket  = new ServerSocket(PORT);

    while (true) {
        Socket socket = serverSocket.accept();

        … set up Scanner and PrintStream and communicate via socket

    }
} catch (IOException e){System.out.println("Failed connection "+ e); }
```

Client

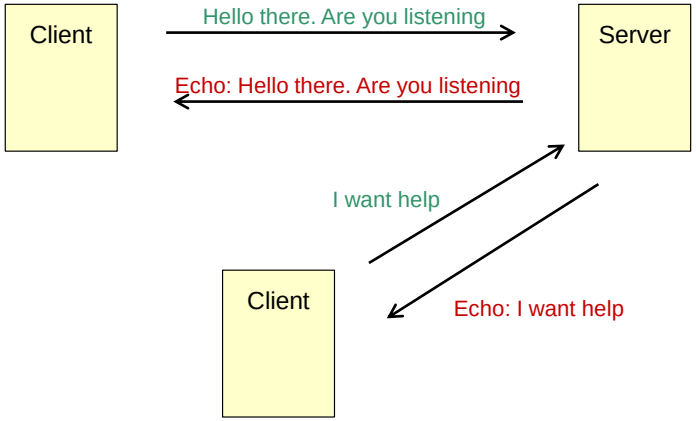
```
try {
    Socket socket = new Socket("irc.ecs.vuw.ac.nz",  PORT);

    …set up Scanner and PrintStream and communicate via socket

} catch (IOException e){System.out.println("Failed connection "+ e); }
```

Echo Service

- Echo server will echo back any message sent to it:



Echo Client.

```
public static final String SERVER = "localhost";
public static final int PORT = 6667;

public static void main(String[ ] args) {
 try {
    Socket socket = new Socket(SERVER, PORT);
    Scanner input = new Scanner(socket.getInputStream());
    PrintStream output = new PrintStream(socket.getOutputStream(), true);
    while (true) {
        String toSend = UI.askString(">");
        if (toSend.equals("QUIT")){ return; }
        output.println(toSend);
        String rcvd = input.nextLine();
        UI.println(rcvd);
    }
 }catch(IOException e){UI.println("IO failure "+ e);}
}
```

Autoflush

Your IRC client

- Will be larger than the echo client
  - Will be more than just a main method
  - Needs a richer interface (buttons, etc)
  - Needs to deal with messages better.
  - Needs a more complex logging in.
- BUT
  - The basic ideas are mostly present in the echo client
  - EXCEPT for asynchronous send and receive.

Echo Server

- Server has to handle lots of clients
- Needs a process in a separate thread for each client

  ⇒ Server requires threads and concurrency!
- Design:
  - Listen to the port.
  - When there is a request,
    - create a new socket for that client connection
    - start a thread to process that client
  - Processing the client
    - loop
      - listen for input from client
      - echo it back
      - quit if the message was QUIT

Echo Server: Listening for clients

```
public class EchoServer{
    public static final int PORT = 6667;    // The port the server will listen on

    public static void main(String[] args) {
        try{
            ServerSocket serverSocket  = new ServerSocket(PORT);
            System.out.println("Waiting for clients to connect...");
            while (true) {
                Socket socket = serverSocket.accept();
                System.out.println("Found client");
                EchoService echoService= new EchoService(socket);
                new Thread(echoService).start();
            }
        }catch(IOException e){System.out.println("Failed to connect" + e);}
    }
}
```

waits for a client

new thread

Echo Server: per client

```
class EchoService implements Runnable {
    private Socket socket;
    private Scanner clientIn;
    private PrintStream clientOut;

    public EchoService(Socket socket) {
        this.socket = socket;
        try {
            clientIn = new Scanner(socket.getInputStream());
            clientOut = new PrintStream(socket.getOutputStream());
        }
        catch (IOException e) {System.out.println("Connection failed." + e);}
    }
```


Echo Server: per client

```
    public void run() {
        while (clientIn.hasNext()) {
            String message = clientIn.nextLine();
            System.out.println("Received: "+ message);
            if (message.equals("QUIT")) { break;}
            clientOut.println("ECHO: "+ message);
            clientOut.flush();
        }
        try{ socket.close();}
        catch (IOException e) {System.out.println("Error socket close" + e);}
        System.out.println("Client disconnected.");
    }


}
```


Synchronous or Asynchronous?

- Echo client:
  - get message from user
  - send message to server
  - get reply from server
  - display reply

Synchronous:
single thread of send and receive.

- IRC client:
  - messages may come from the server unprompted by user
  - messages may come from the user unprompted by server

  ⇒ client must be listening to both user and server

  ⇒ client must have two threads!

Asynchronous:
two threads of send and receive.

Asynchronous client

```
public class AsynchClient  {
    private static final String SERVER = "localhost";
    private static final int PORT = 6667;

    private Socket socket;
    private Scanner input;
    private PrintStream output;

    public static void main(String[] args) {  new AsynchClient();  }

    public AsynchClient(){
        try {
            socket = new Socket(SERVER, PORT);
            new Thread(new Runnable(){ public void run(){
                listenToServer();
            }}).start();
            sendToServer();
        }catch(IOException e){UI.println("IO failure "+ e);}
    }
```


Asynchronous client

```
public void sendToServer(){
    PrintStream output = new PrintStream(socket.getOutputStream());
    while (true) {
    String toSend = UI.askString(">");
    output.println(toSend);
    output.flush();
    }
}
```

```
public void listenToServer(){
    Scanner input = new Scanner(socket.getInputStream());
    while (input.hasNext()){
        String line = input.nextLine();
        UI.println("SERVER: "+ line);
    }
}
```


Threads

- What is a thread?
  - Like a separate CPU running a program (or part of a program).
  - independent of all the other threads (could be faster or slower).
- Java threads all have access to the same memory
  - Two threads accessing the same location can cause conflict and error
- Safe Programming with threads is HARD.
  - Harder to debug.
  - No problems if the different threads don't share any resources
  - You should expect some odd things to happen if the threads do share resources (eg, the same window!)

How do you get a Thread?

- The main method is called with one thread
  - anything it calls is executed in that thread:
    main → constructor → sendToServer
- The GUI events are executed in a separate thread
  - repainting, responding to mouse, buttons etc.
- Create a new Thread object and call run() on it.
- ecs100 library:
  - Each call to buttonPerformed is done in a new thread
  - Mouse events are processed by a mouse event thread.

IRC login

Login sequence:

→ NICK pondy

→ USER pondy 0 unused :Peter Andreae

←…*lots of messages… including grumbling about looking up hostname*

← :irc.ecs.vuw.ac.nz.net 001 pondy …….

← :irc.ecs.vuw.ac.nz.net 002 pondy …….

← :irc.ecs.vuw.ac.nz.net 003 pondy …….

← :irc.ecs.vuw.ac.nz.net 004 pondy irc.ecs.vuw.ac.nz hybrid-7.2.3 ….

or

←:irc.ecs.vuw.ac.nz.net 433 * pondy :Nickname is already in use