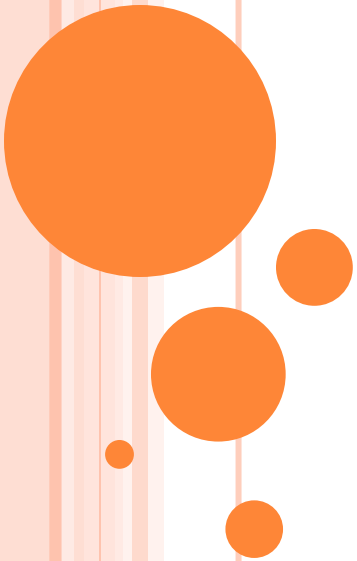


Network Programming: A Java Perspective

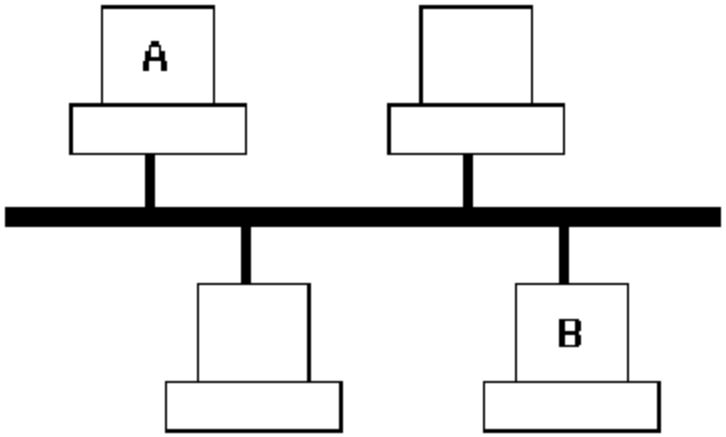
COMP 30220: Distributed Systems

Lecturer: Rem Collier

Email: rem.collier@ucd.ie

Basic Networking: Local Networks

- Ethernet bus:



- IP addresses identify nodes
  - 193.1.132.49
- ARP resolves IP addresses to Media Access Control (MAC) addresses.
  - stored on the ROM on the Network Card Interface (NIC).
  - 08-00-27-00-3c-77 (try `ipconfig -all` or `ifconfig` on Linux/MacOS)
- DNS resolves names to IP addresses
  - www.cs.ucd.ie -> 137.43.93.109

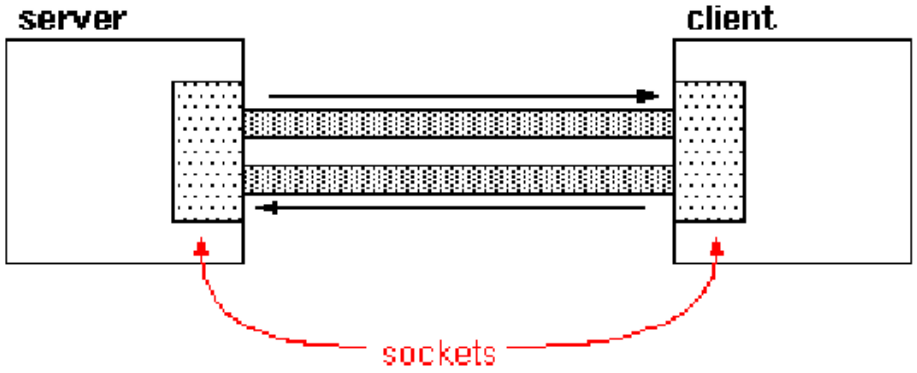
Ports

- All network traffic arrives at the NIC
- Can address data to individual processes
  - via a port number
- Standard Ports:
  - 80 HTTP, 442 HTTPS
  - 22 SSH, 23 TELNET
  - 25 SMTP
  - 53 DNS
- User ports > 1024
- Try
  `telnet www.cs.ucd.ie 80` (use putty on windows)
  - Specifying port 80 means telnet should connect to a web server. Once it connects, try typing the following line:

```
GET /index.html
```

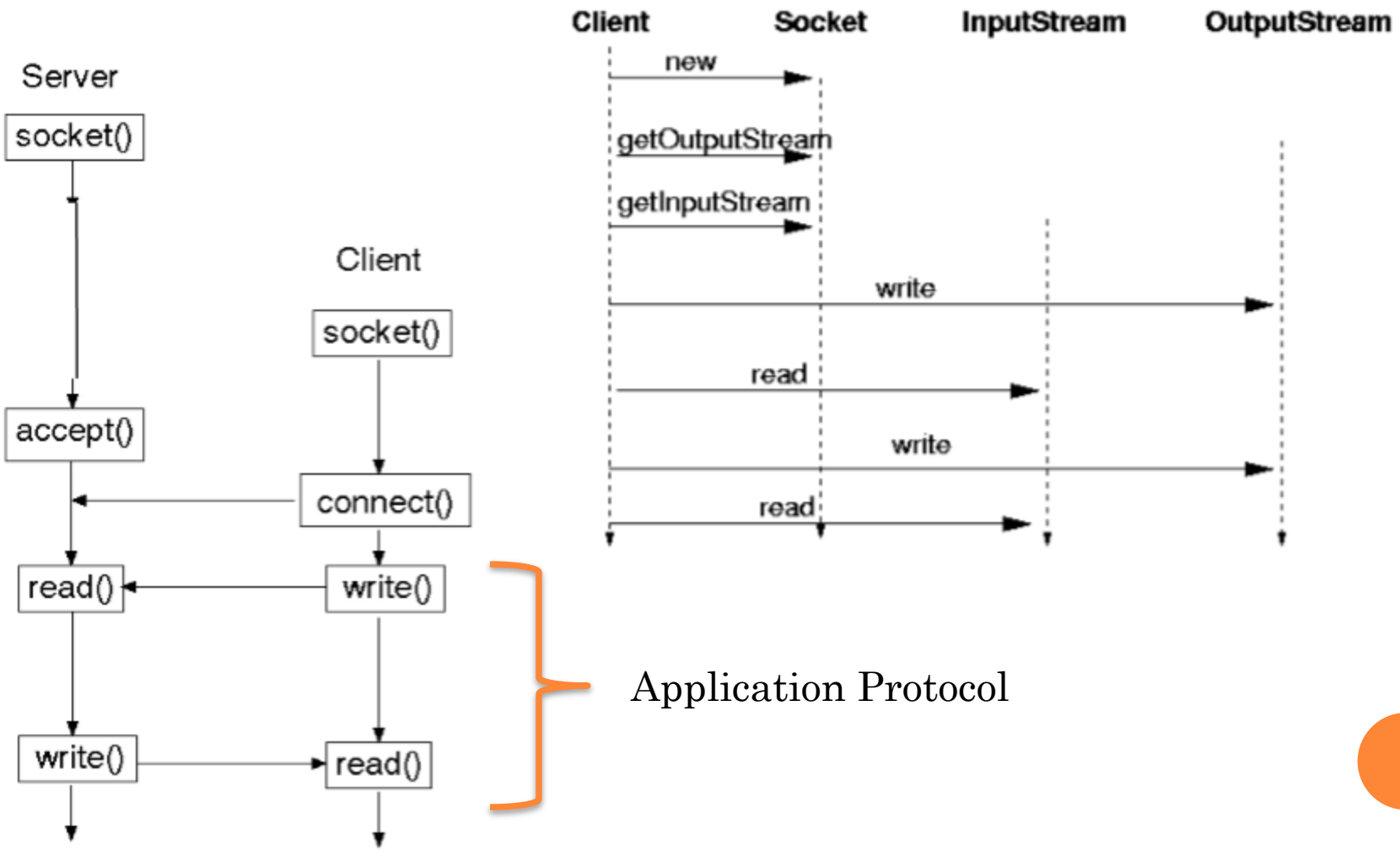
Sockets and Streams

- Client-server architecture
- Server opens a socket on designated port and listens ("I'm opening a socket" – Chloe)



- Client connects and two-way comms becomes possible
  - At finish, socket is torn down
- A **protocol** defines the sequence of interactions between the client and server

Sockets and Streams



Client Sockets in Java

```
try {
    Socket mySocket = new Socket("www.cs.ucd.ie", 80);
    InputStream is = mySocket.getInputStream();
    OutputStream os = mySocket.getOutputStream();

    BufferedReader in =
        new BufferedReader(new InputStreamReader(is));
    PrintWriter out = new PrintWriter(os, true);

    // send a string to the server
    out.println("GET /index.html");

    // Print out the response
    String line = null;
    while ((line = in.readLine()) != null) {
        System.out.println(line);
    }
    in.close(); out.close(); mySocket.close();
} catch (IOException e) { e.printStackTrace(); }
```

Create a new socket connected to www.cs.ucd.ie on port 80

Get input and output streams for read/write

Wrap the streams to make our lives easier true = autoflush

Close everything down – this flushes the streams

Server Sockets in Java: EchoServer

```
try {
    ServerSocket serverSocket = new ServerSocket(7788);

    Socket socket = serverSocket.accept();



    BufferedReader in = new BufferedReader(
        new InputStreamReader(socket.getInputStream()));
    PrintWriter out = new PrintWriter(socket.getOutputStream(), true);



    String message = in.readLine();
    out.println(message);
} catch (IOException e) { e.printStackTrace(); }
```

Open a ServerSocket listening on port 7788

Blocking call that returns a socket when a client connects

Get input and output streams for read/write and wrap them

Receive a message, echo it back as the response and then quit…

EchoClient

```
try {
    Socket mySocket = new Socket("localhost", 7788);
    InputStream is = mySocket.getInputStream();
    OutputStream os = mySocket.getOutputStream();

    BufferedReader in =
        new BufferedReader(new InputStreamReader(is));
    PrintWriter out = new PrintWriter(os, true);

    // send a string to the server
    out.println("Hello World");

    // Print out the response
    String line = in.readLine();
    System.out.println(line);

    in.close(); out.close(); mySocket.close();
} catch (IOException e) { e.printStackTrace(); }
```

Open a socket to the server listening on port 7788 (localhost)

Send a message to the server

Simplified handling of response – only expecting one line back…

EchoServer2: Multiple Connections

```
public class EchoServer2 {
    public static final int BACKLOG = 5;
    public static void main(String[] args) { new EchoServer2(7788).listen(); }
    private ServerSocket serverSocket;
    public EchoServer2(int port) {
        try {
            serverSocket = new ServerSocket(7788, BACKLOG);
        } catch (IOException e) { e.printStackTrace(); }
    }
    public void listen() {
        while (true) {
            try {
                Socket socket = serverSocket.accept();
                BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
                PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
                String message = in.readLine();
                out.println(message);
            } catch (IOException e) { e.printStackTrace(); }
        }
    }
}
```

BACKLOG = number of concurrent client connections

serverSocket is a field

Infinite loop = keep on accepting client connections.

EchoClient2: Multiple Connections

```
public class EchoClient2 {
    public static void main(String[] args) {
        EchoClient2 client = new EchoClient2("localhost", 7788);
        for (int i=0; i<10; i++) client.sendMessage("Hello World");
    }
    private String host; private int port;
    public EchoClient2(String host, int port) { this.host = host; this.port = port; }
    public void sendMessage(String message) {
        try {
            Socket mySocket = new Socket(host, port);
            InputStream is = mySocket.getInputStream();
            OutputStream os = mySocket.getOutputStream();
            BufferedReader in = new BufferedReader(new InputStreamReader(is));
            PrintWriter out = new PrintWriter(os, true);
            out.println(message);
            String line = in.readLine();
            System.out.println(line);
            in.close(); out.close(); mySocket.close();
        } catch (IOException e) { e.printStackTrace(); }
    }
}
```

Insert: Threads

- **Process**: a running instance of a computer program.
  - Each program is executed in a separate memory space.
  - Direct interaction between processes is limited to semaphores.

- **Thread**: an individual control flow within a process.
  - A process can contain main threads that can be executed in parallel.
    - Unlike processes, threads have shared memory.
    - Threads can communicate via this shared space.
  - When two threads try to access the same object at the same time, problems can arise.
    - What happens two threads try to modify and read the same field?
    - Which value is read?
    - Can we guarantee this will happen consistently?
  - **Monitors** solve this by enforcing **mutual exclusion**.
    - In Java, mutual exclusion is achieved through the use of **synchronized** blocks and methods.

Insert: Threads

- When a Java program is launched, a single thread is created (there is a single control flow).
  - Unless you explicitly create a thread all your code runs in that single "main" thread.
  - When you start a new thread:
    - the new thread is initialised and started
    - the existing thread continues to execute in parallel
  - Code that is to be run in a thread should be created by implementing the `Runnable` interface:

```
class MyThread implements Runnable {
    private String name;
    public MyThread(String name) { this.name = name; }
    public void run() {
        for (int i=0; i < 1000; i++) {
            System.out.println(name + "="+i);
        }
    }
}
```

Insert: Threads

- To run the thread:
  - Create an instance of the Thread class, passing an instance of the class containing the code to be run in the thread.
  - Start the thread.
  - Example:

```
Thread t = new Thread(new MyThread("rem"));
t.start();
```

  - Execution of threads is **interleaved**:

```
for (int i=0; i<2; i++) {
    Thread t = new Thread(new MyThread("th-" + i));
    t.start();
}
```

  - WARNING: Interleaving of threads is non-deterministic.

Threading Example

```
public class SharedMemoryThread implements Runnable {
    private static class SharedValue {
        private int value;
        public void updateValue(int value) {
            System.out.println("updating: " + value);
            this.value = value;
            System.out.println("value=" + this.value);
            System.out.println("updated: " + value);
        }
    }
    private SharedValue value;
    public static void main(String[] args) {
        SharedValue value = new SharedValue();
        new Thread(new SharedMemoryThread(value)).start();
        new Thread(new SharedMemoryThread(value)).start();
    }
    public SharedMemoryThread(SharedValue value) { this.value = value; }
    public void run() {
        for (int i=0; i<1000; i++) { value.updateValue(i); }
    }
}
```

ThreadedEchoServer

```
public class ThreadedEchoServer {
    public static final int BACKLOG = 5;
    public static void main(String[] args) {
        new ThreadedEchoServer(7788).listen();
    }

    private ServerSocket serverSocket;
    public ThreadedEchoServer(int port) {
        try {
            serverSocket = new ServerSocket(7788, BACKLOG);
        } catch (IOException e) { e.printStackTrace(); }
    }
    public void listen() {
        while (true) {
            try {
                new Thread(new Connection(serverSocket.accept())).start();
            } catch (IOException e) { e.printStackTrace(); }
        }
    }
}
```

ThreadedEchoServer

```
public class Connection implements Runnable {
    private Socket socket;
    public Connection(Socket socket) { this.socket = socket; }
    public void run() {
        try {
            BufferedReader in = new BufferedReader(
                new InputStreamReader(socket.getInputStream()));
            PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
            String message = in.readLine();
            while (message != null) {
                out.println(message);
                message = in.readLine();
            }
        } catch (IOException e) { e.printStackTrace(); }
    }
}
```

EchoClient3: Reusing Connections

```
public class EchoClient3 {
    private String host; private int port;
    private Socket mySocket;
    private BufferedReader in; private PrintWriter out;
    public EchoClient3(String host, int port) { this.host = host; this.port = port; }
    public void connect() {
        try {
            mySocket = new Socket(host, port);
            in = new BufferedReader(new InputStreamReader(mySocket.getInputStream()));
            out = new PrintWriter(mySocket.getOutputStream(), true);
        } catch (IOException e) { e.printStackTrace(); }
    }
    public void sendMessage(String message) {
        try {
            out.println(message); System.out.println(in.readLine());
        } catch (IOException e) { e.printStackTrace(); }
    }
    public void close() {
        try {
            in.close(); out.close(); mySocket.close();
        } catch (IOException e) { e.printStackTrace(); }
    }
}
```

EchoClient3: Reusing Connections

```
public static void main(String[] args) {
    EchoClient3 client = new EchoClient3("localhost", 7788);

    client.connect();
    for (int i=0; i<10; i++) {
        client.sendMessage("Hello World");
    }

    client.close();
}
```

Binary Data

- Sending Binary Data:

```
DataOutputStream os =
    new DataOutputStream(socket.getOutputStream());
os.writeInt(3);
os.writeFloat(6.99);
os.writeBytes(userInput);
os.writeByte('\n')
```

- Receiving Binary Data:

```
DataInputStream is =
    new DataInputStream(socket.getInputStream());
int n = is.readInt();
Float f = is.readFloat();
String str = is.readLine();
```

Java Socket Hints

- Use the API that fits your protocol and your data:
  - Wrap the streams in `BufferedReaders` and `PrintWriters` for unicode strings
  - Wrap the streams in `DataOutputStreams` and `DataInputStreams` for binary data
- Watch your exception handling
  - Leaving open sockets can break later executions
  - Use `finally` blocks
- Close streams and judicious use of `flush()`
- Spawn threads at server side to deal with incoming requests
  - Use a simple thread pool

Java Socket Hints

- Keep using the same socket if you need to exchange more messages
  - This is typically what we call a **PROTOCOL**.
- Separate code handling sockets from functional code
  - General principle of modularity and separation of concerns (communication vs. business logic)

- Socket Tutorials:
  - https://docs.oracle.com/javase/tutorial/networking/sockets/index.html
  - http://www.oracle.com/technetwork/java/socket-140484.html
- Thread Tutorials:
  - http://docs.oracle.com/javase/tutorial/essential/concurrency/