COMP2221 Networks

David Head

University of Leeds

Lecture 8

Reminder of the last lecture

Last lecture we saw how to code a network **client** in Java:

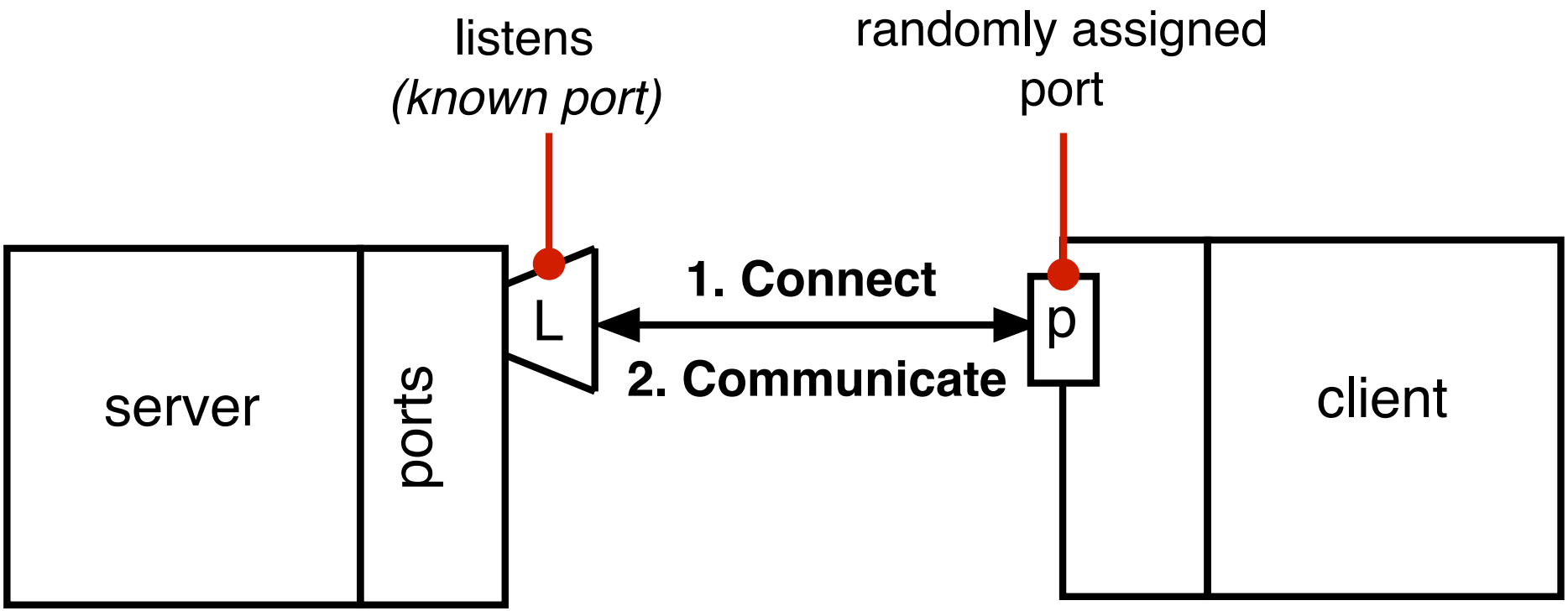
- How applications use **sockets** to interface with the Transport layer.
- How to use the `Socket` class from `java.net`.
- Each socket has an output and input stream, for sending and receiving data.
- Typically **chain** these **raw** data streams to at least include buffering.
- Reading from a socket is a **blocking** operation - it waits until there is something to read.

Today's lecture

Today we will look at the other half of this network communication - the **server**.

- Uses `ServerSocket`, also in `java.net`.
- **Listen** to a **specified** port until a client attempts contact.
- **Accept** that client in the form of an instance of `Socket`.
  - This socket will have the **same port number** as the listening port.
- See the code for `DailyAdviceServer` and `KnockKnockServer`.
- Identify a fundamental **limitation** that we will try to resolve over the next 4 lectures.

Reminder


listens
(known port)
randomly assigned
port
server
ports
L
1. Connect
2. Communicate
p
client

Which is 'client', which is 'server'?

Nothing in the network architecture distinguishes between **clients** and **servers**.

- Both are **hosts** or **end systems** *[cf. Lecture 1]*.

The key difference is how the connection is initially made.

- The **client** makes the connection **request**.
- The **server** accepts the connection (using `ServerSocket`).

Once the communication has started, there may be nothing to say which is the client and which is the server.

- Both use `Socket` objects to communicate.
- Application *may* differentiate between the two, *e.g.* web browser (client) and server.

Clients and Servers

Clients . . .

- assume that the server exists, at a prescribed IP address **and port**;
- start and connect, communicate (following a prescribed **protocol**), then close and exit.

Servers . . .

- run **continuously** on the host;
- **listen** for client connections on the prescribed port;
- handle connections/disconnections to clients;
- implement the connection protocol.

From last lecture

There are 7 basic socket operations:

1. Connect to a remote machine.
2. Send data.
3. Receive data.
4. Close a connection.
5. Bind to a (listening) port.
6. Listen for incoming connection requests.
7. Accept connections from remote machines in the bound port.

Clients **only implement the first 4**.
Servers **require all 7**.

The ServerSocket class

Contains the **only additional functionality** required for servers:

- Opens the given port to external connections.
- Listens for TCP connections on the given port.
- Negotiates the connection between client and server.
- Creates a plain `Socket` object for communication with each client.

The remaining functionality, *i.e.* for communication, is in the `Socket` class.

The Most Common Constructor

```
public ServerSocket( int port ) throws BindException, IOException
```

- `port` is the port number to **listen** for connections.
- The **host** is always[1] the **local** machine, *i.e.* `localhost`.
  - Contrast with `Socket`, where the **remote** machine and port is specified.
- `BindException` arises when the socket could not be created and **bound** to that port.
- `BindException` is a type of `IOException`.

[1] For host machines with multiple IP addresses, there is a constructor which allows you to select which one.

Getters

`public InetAddress getInetAddress()`

- Returns the address the server is using.

`public int getLocalPort()`

- The port that the server is listening on.
- Usually not very useful, as we normally assign a port in the constructor.

There are no (public) `set`-methods for these quantities; they are **immutable**.

- There *are* `set`-methods for buffer size, timeout, *etc*.

Useful Methods

`public Socket accept() throws IOException`

- **Blocks** (*i.e.* waits) until a client connects to the listening port.
- Returns a new `Socket` to communicate with that client.

`public void close() throws IOException`

- Frees up the listening port for another application to use.
- Also closes any connected `Socket`.
- Should be called if the server is **not** in an infinite loop.
- Killing the server application (*i.e.* with Ctrl-C on the command line) should also free up the listening port.

A recipe for a simple server is:

1. **Create** a new `ServerSocket` on a port.
2. **Listen** for a connection using the `accept()` method, which waits until a client connects, when it returns a `Socket` object.
3. Set up `Socket` input and output **streams** for I/O.
4. **Communicate** using the agreed **protocol**.
5. Client, server or both **close** the connection.
6. Return to 2.

The **protocol** is implementation-dependent, and may be **encapsulated** in a separate class.

Example 1: DailyAdviceServer

Code on Minerva: DailyAdviceServer.java

```
1  try {
2    ServerSocket serverSock = new ServerSocket(4242);
3    while( true ) {
4      Socket sock = serverSock.accept();
5      InetAddress inet = sock.getInetAddress();
6      ... // Log date and client address.
7
8      // Protocol: Send a line of text.
9      PrintWriter writer = new PrintWriter(
10                                sock.getOutputStream());
11     String advice = getAdvice();
12     writer.println(advice);       // To client
13     writer.close();
14     System.out.println(advice);  // Local server echo
15     sock.close();
16   }
17 } catch (IOException ex) { ... }
```

Example 2: KnockKnockServer

Code on Minerva: KnockKnockServer.java, KnockKnockProtocol.java

Use a separate **protocol** class to handle the communication with the client:

```
1 public class KnockKnockProtocol;
```

Use instance variables for server socket and the protocol:

```
1 private ServerSocket serverSocket = null;
2 private KnockKnockProtocol kkp = null;
```

The use of a protocol separates connection handling from communication.

- Also useful for **multi-client servers**, which we will look at over the next few lectures.

Initialise the port to listen to in the constructor:

```
try {
  serverSocket = new ServerSocket(2323);
}
catch (IOException e) {
  System.err.println("Cannot listen on port 2323.");
  System.exit(1);
}
```

Also initialise the protocol:

```
kkp = new KnockKnockProtocol();
```

Note could use *e.g.* `private int listeningPort=2323` to allow the port number to be more easily changed (although the clients need to know this number!)

```
while( true ) {
  try {
    clientSocket = serverSocket.accept();
  } catch (IOException e) { ... }

  try {
    PrintWriter out = new PrintWriter(
                        clientSocket.getOutputStream(),true);
    BufferedReader in = new BufferedReader(
                          new InputStreamReader(
                            clientSocket.getInputStream()));

    String outputLine = kkp.processInput(null);
    while( (inputLine=in.readLine())!=null ) {
      outputLine = kkp.processInput(inputLine);
      out.println(outputLine);
      if (outputLine.equals("Bye.")) break;
    }
    out.close();
    in.close();
    clientSocket.close();
  } catch (IOException e) { ... }
}
```

Notes on KnockKnockServer

- The `PrintWriter` had **auto-flush** set to `true`.
  - Calls to `println()`, `printf()` and `format()` will flush the output buffer.
- The `InputStreamReader` is buffered using **chaining** of I/O **streams**.
- The **protocol** is largely handled by a separate class `KnockKnockProtocol` (see next slides).
- `KnockKnockProtocol` also tests we **conform** to the protocol, *i.e.* respond in the expected way.
- `KnockKnockServer` is **sequential** - one input from the client gets one response from the server.

KnockKnockProtocol

Code on Minerva: KnockKnockProtocol.java

The **protocol** class has a **state variable** and some readable state names:

```
1 private static final int WAITING = 0;
2 private static final int SENTKNOCKKNOCK = 1;
3 private static final int SENTCLUE = 2;
4 private static final int ANOTHER = 3;
5
6 private int state = WAITING;
```

Could also use an enumerated type for the possible states:

```
1 private enum jokeState = { ... }
```

```
1  public String processInput(String theInput) {
2    String theOutput = null;
3
4    if (state == WAITING) {
5      theOutput = "Knock! Knock!";
6      state = SENTKNOCKKNOCK;
7    } else if (state == SENTKNOCKKNOCK) {
8      if (theInput.equalsIgnoreCase("Who's there?")) {
9        theOutput = clues[currentJoke];
10       state = SENTCLUE;
11     } else {
12       theOutput="You're supposed to say \"Who's there?\"!"+
13         " Try again. Knock! Knock!";
14     }
15   } else if( state==SENTCLUE )
16   {
17     // See code for the remainder.
18   }
19   return theOutput;
20 }
```

KnockKnockProtocol

Notes:

- Uses an internal `state` variable to keep track of the stage of the joke we have reached.
  - *i.e.* the **protocol stage**.
  - Takes different values for different stages, *i.e.* `WAITING`, `SENTKNOCKKNOCK` *etc.*
- The code could be improved, *i.e.*:
  - **Enumerated** types for the state.
  - `switch` statement rather than the multiple `if-then-elses`.
  - ...

Limitations

For both examples, we are limited to **one** client connection.

- Also only one instance of the protocol.

This is far from ideal:

- While the current client protocol is active, we cannot `accept()` another client.
- Therefore, there can only be at most **one client at any given moment**.
- Imagine a web server that could only deal with one client at a time!

You might think to launch multiple servers simultaneously, but ...

- They cannot both use the same port to listen.
- The second and subsequent servers will fail with a `BindException` (a type of `IOException`).

A better solution would be to `accept()` the client, and handle communications as a separate **thread**.

- The server thread is then available for more client connections.

This **multi-threaded** approach will be described in the next few lectures.

Further Study

The client and server codes for both examples are on Minerva.

Examine the codes and understand how they conform to the model presented here.

- To run locally (*i.e.* on your laptop), make sure the destination host in the client code is `localhost`.
- Also make sure the client's destination port matches the server's listening port.

For reference, the Harold book *Java Network Programming* covers these topics.

- In the 4^{th} edition, client sockets are covered in Chapter 8, and server sockets in Chapter 9.

Summary and next time

We have seen how to implement a simple server using `ServerSocket`:

- **Listens** on a prescribed port for connection requests from clients using `accept()`.
- Opens a connection for each client, which will be on a different port.
- Can only handle one client at a time.

Over the next three lectures we will see how using **threads** can allow one server to communication with multiple clients simultaneously.