COMP2221 Networks

David Head

University of Leeds

Lecture 7

Two lectures ago we looked a IP addressing in Java

- Handled by the `InetAddress` class from `java.net`.
- Accesses the configured DNS server to convert hostname to at least one IP address.

Last lecture we looked at I/O streams:

- Defined in `java.io`.
- How they **abstract** the I/O process from the source/destination
- Typically use **buffers** for efficiency.
- **Chaining** of multiple filter streams.

Today's lecture

Today we are going to start looking at sending data over the network.

- Requires the use of **sockets**.
- For **clients**, we use the Java `Socket` class, defined in `java.net`.
- Give some examples of using `Socket`.
  - `LowPortScanner`, which does not communicate.
  - `DailyAdviceClient`, which only receives.
  - `KnockKnockClient`, which receives and sends.

Next time we will see the other half, *i.e.* the **server** application (which uses Java's `ServerSocket` class).

Network communication recap

Recall that:

- Data is transmitted across the network as a series of **packets**.
- Each packet contains a **header** and a **payload**.
- The header contains the **IP address** and **port** of the destination and the source.
- Packets may arrive out-of-order, or be corrupted/lost and re-transmitted.
  - TCP handles this automatically, potentially with a performance cost.

What is a socket?

A socket is an abstract input-output device.

It may correspond to a display, a keyboard, a printer, or a data communication line.

It is used by a program to input or output a stream of data.

The use of sockets shields us from the low-level details of network communication.

- *i.e.* it is an Application layer concept distinct from the Transport layer.

Networks specifically

A socket is **one end-point** of a two-way communication link between two hosts.

Each end-point is a combination of an IP address and a port number.

A socket is **bound to a port number**, so that the Transport layer can identify the recipient in the Application layer.

- **Immutable** - once the link is made, it cannot be altered without breaking it.
- *i.e.* no public `setPort()`/`setAddress()` methods; only `getPort()`/`getLocalPort()`/`getAddress()`.

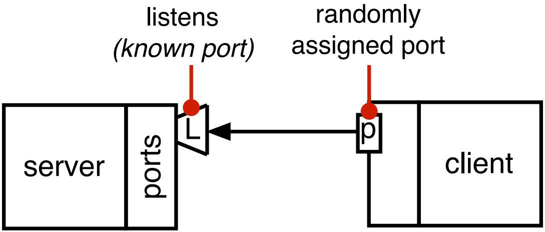
Every TCP connection can be uniquely identified by its endpoints.

Socket connections

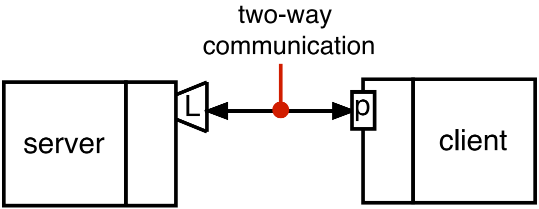
There are 7 basic operations:

1. Connect to a remote machine.
2. Send data.
3. Receive data.
4. Close a connection.
5. **Bind** to a port.
    - Fixed port in an application, *e.g.* 80 for a web server (`http`).
6. **Listen** for new connection requests.
7. **Accept** connections from remote machines in the bound port.

1. Connection



2. Communication



Sockets for clients

For clients, only the first four of these are relevant, *i.e.*

1. Connect to a remote machine.
2. Send data.
3. Receive data.
4. Close a connection.

Only these 4 have methods in the `Socket` class.

The remaining 3 are related to **servers**, and have methods in the `ServerSocket` class.

- We will look at server sockets next lecture.

The Java `Socket` class

Implements the TCP communications protocol.

- There is another class for UDP that we will come to in Lecture 14.

A typical **client** session might look like:

- A new socket is created, using the Java `Socket` constructor.
- The socket attempts to connect to a given remote host at the given port.
- The local machine and the remote machine send and receive data.
  - The meaning of the data sent depends on the applications.
- The connection is two-way; **both** can send **and** receive.
- One or both of the hosts close the connection.

Common constructors

Two constructors are most commonly used. Both perform networking tasks to make the connection.

```
public Socket( String host, int port )
       throws UnknownHostException, IOException
```

- Tries to create an InetAddress object from the hostname.
    - If not possible, throws UnknownHostException.
- IOException thrown for *e.g.* unreachable host, routing problem *etc.*

```
public Socket( InetAddress host, int port ) throws IOException
```

- May throw IOException for same reasons as above.

Simple getters

Accessing information about the remote host:

- `public InetAddress getInetAddress()`
- `public int getPort()`
- These are **immutable**; there are no setters.
- Remote port number known prior to making connection, *e.g.* a recognised reserved port such as 22 for `ssh`.

For the local host:

- `public int getLocalPort()`
- Port number chosen by the OS at runtime.
- Multiple clients connect from same host on different ports.

Stream methods

Each socket has **streams** to receive or send data *via* the socket:

- `public InputStream getInputStream() throws IOException`
- `public OutputStream getOutputStream() throws IOException`

These are **raw** data streams (byte streams).

- Would normally **chain** to make it more usable, and/or more efficient (*e.g.* **buffered**).
- See last lecture on Java I/O streams.

Example 1: Low Port Scanner

Code on Minerva: `LowPortScanner.java`

Alongside the slides for this lecture in Minerva is the code `LowPortScanner.java`:

- Attempts to open a `Socket` to each port in the range 1 to 1023 on `localhost` (*i.e.* the machine running the code).
- Reports when succeeded.
- If an `IOException` is thrown when trying to construct the `Socket`, reports nothing.

Shows any holes in the system, i.e. open ports. All should be in `/etc/services` and identifiable.

```java
public class LowPortScanner
{
  public static void main( String[] args )
  {
    String hostname = "localhost";
    if( args.length > 0 ) hostname = args[0];

    // Try every reserved port number.
    for( int i = 1; i < 1024; i++ ) {
      try {
        Socket s = new Socket(hostname,i);
        System.out.println("There is a server on port "
                                 + i + " of " + hostname);
      }
      catch( UnknownHostException ex ) {
        System.err.println(ex); // Problem with host
        break;
      }
      catch( IOException ex ) {}
    }
  }
}
```

Example output from LowPortScanner

When executed on a School machine[1], LowPortScanner generates output something like the following:

```
There is a server on port 22 of localhost
There is a server on port 25 of localhost
There is a server on port 53 of localhost
There is a server on port 111 of localhost
There is a server on port 631 of localhost
```

You can check in /etc/services to see what the port refers to, or use grep, *e.g.*

```
% grep 22 /etc/services | more
```

[1] You may get no results at all if running on *e.g.* your laptop.

Example 2: Daily Advice Client

Code on Minerva: `DailyAdviceClient.java`

- Connects to `localhost`.
- Tries to connect to port 4242 *(see server code next lecture)*.
- Open an input stream (*i.e.* read only).
- The server sends a single line of advice.
- The client displays the advice, and the connection is closed.

Since it connects to `localhost`, it assumes the server is already running **on the same host**.

- Download the server code from Minerva Lecture 8, and run in a separate shell.
- We will see how the server works next time.

```java
public class DailyAdviceClient
{
  public void connect() {
    try{
      Socket s = new Socket("localhost",4242);
      BufferedReader reader = new BufferedReader(
                                new InputStreamReader(
                                 s.getInputStream()));

      String advice = reader.readLine();
      System.out.println("Thought for the day: " + advice);
      reader.close();
      s.close();
    }
    catch( IOException e ) { ... }
  }

  public static void main(String[] args)
  {
    DailyAdviceClient client = new DailyAdviceClient();
    client.connect();
  }
}
```

Example 3: A Knock-Knock Client

Code on Minerva: `KnockKnockClient.java`

'Knock-knock' is a type of joke that follows the **protocol**:

A: Knock knock.

B: Who's there?

A: . . .

B: . . . who?

A: (punchline)

Unlike the previous example, the client and server must send **and** receive multiple times for **each** connection.

As with `DailyAdviceClient`, this expects to find the server on `localhost`, so you will need to first launch the server `KnockKnockServer` in a separate shell.

Knock-Knock Client Code (1)

```
public class KnockKnockClient
{
  private Socket kkSocket = null;
  private PrintWriter socketOutput = null;
  private BufferedReader socketInput = null;

  public void playKnockKnock() {
    try {
      kkSocket = new Socket("localhost",2323);
      socketOutput = new PrintWriter(
                       kkSocket.getOutputStream(),true);

      socketInput = new BufferedReader(
                      new InputStreamReader(
                        kkSocket.getInputStream()));
    }
    catch( UnknownHostException e ) { ... }
    catch( IOException e ) { ... }
    // Continued ...
```

Knock-Knock Client Code (2)

```
    BufferedReader stdIn = new BufferedReader(
                          new InputStreamReader(System.in));
    String fromServer;
    String fromUser;

    try {
      while( (fromServer=socketInput.readLine())!=null )
      {
        System.out.println("Server: " + fromServer);
        if (fromServer.equals("Bye.")) break;

        fromUser = stdIn.readLine();
        if( fromUser!=null ) {
          System.out.println("Client: " + fromUser);
          socketOutput.println(fromUser);
        }
      }
      // Continued ...
```

Knock-Knock Client Code (3)

```java
      socketOutput.close();
      socketInput.close();
      stdIn.close();
      kkSocket.close();
    }
    catch( IOException e ) {
      System.err.println("I/O exception.\n");
      System.exit(1);
    }
  }

  public static void main(String[] args) {
    KnockKnockClient kkc = new KnockKnockClient();
    kkc.playKnockKnock();
  }
}
```

Notes

The `readLine()` commands are **blocking**:

- They will not return until something has been read.

The client application just sends user input from `System.in` to the server, and echoes the response.

- Terminates when the server sends `"Bye."`.
- A type of **protocol** - understood by both parties that the `"Bye."` message should result in termination.

Generating the 'jokes' and checking for the correct input from the user is the job of the server.

- We will look at the server code next lecture.

Today we have looked at Java client applications:

- The `Socket` class in `java.net`.
- How to construct and extract streams for sending or receiving data.
- Three examples: `LowPortScanner` (no communication), `DailyAdviceClient` (receives only) and `KnockKnockClient` (receives and sends).

Next time we will see how to write a Java server using `ServerSocket`.