APIT - Distributed Systems

Dr. Simon Rogers

19/02/2018

Overview

- Previously saw how we could run multiple processes on one machine - threads
- What about processes communicating across machines?
  - Examples?
- These are *distributed* systems

Servers and sockets

- We will build *Servers* and *Clients*
- Java has inbuilt objects to do this
- `ServerSocket` – server
- `Socket` – client

Building a server

- Servers can be created via ServerSocket objects (SimpleServer):

```
import java.net.*;
import java.io.*;
public class SimpleServer {
    private static int PORT = 8765;
    public static void main(String[] args) throws IOExcepti
        // Make a server object
        ServerSocket listener = new ServerSocket(PORT);
        // Wait for a connection and create a client
        Socket client = listener.accept();
        // Close the connection
        client.close();
    }
}
```

`ServerSocket.accept()`

- The `accept()` method of the `ServerSocket` object waits indefinitely for a connection.
- Once a client has arrived, it created the `Socket` object
- Once the `Socket` is made, the Server moves onto the next instruction

Aside: IP addresses and ports

- The Internet Protocol (IP) is a set of rules used for connecting devices in a network
- All devices on the network are assigned an IP address.
- e.g. 192.168.1.122
  - Each portion goes from 0 to 255
  - There are rules - have a look online
- Some special addresses:
  - 127.0.0.1 - use this to access your own machine *from* your own machine (localhost)
- Finding your address:
  - `ipconfig`, `ifconfig`

- A particular machine my be involved in several client-server communications
- These are subdivided through the use of `ports`
  - Ports are an abstract thing – they are produced in software
- When we create a server, we choose a (currently unused) port
- Clients need to know which port to access the server through
- In the previous example, we used the port 8765
- Commonly used ports:
  - 20,21: FTP
  - 22: SSH
  - 80: HTTP

Building a client

- Clients are created via `Socket` objects (`SimpleClient`):

```
import java.io.*;
import java.net.*;
public class SimpleClient {
    private static int PORT = 8765;
    private static String server = "127.0.0.1";
    public static void main(String[] args) throws IOExcepti
        // Make a socket and try and connect
        Socket socket = new Socket(server,PORT);
        // Close the socket
        socket.close();
    }
}
```

What's happening

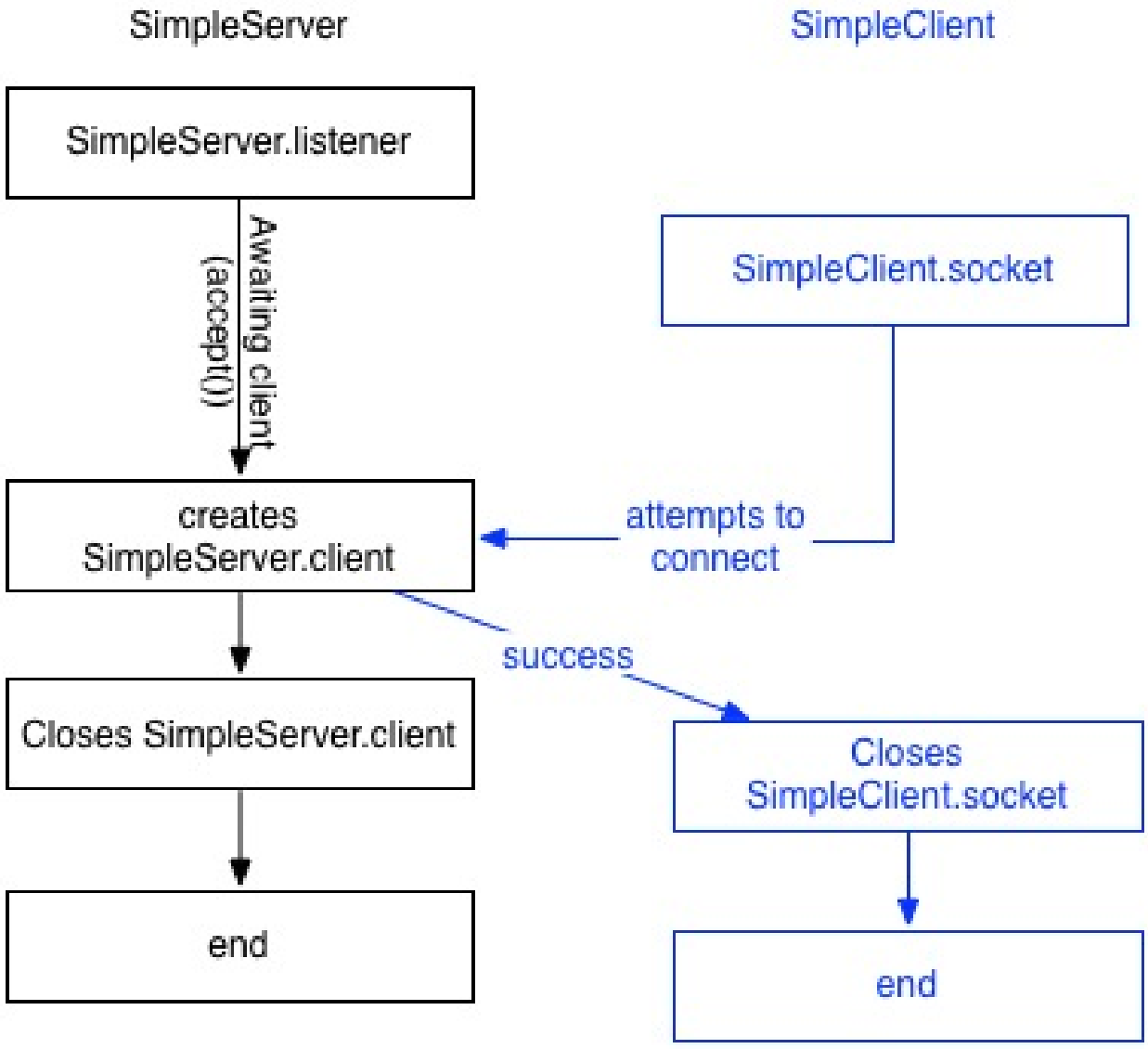

Figure 1:

Client-server communication

- Communication can be performed through input and output `streams`
  - Streams are *continuous flows of data*. i.e. data can be put in at one end, and read at the other end
  - Streams transmit bytes
- `InputstreamReader` and `OutputStreamWriter` provide the functionality to convert individual characters to bytes that can be sent down a `Stream`.

Sending and receiving a single character

- To send a character from the Server:
  - Create an `OutputStreamReader` from the `Sockets` output stream
  - Use `write` to write a single character
  - Use `flush` to force it to send
- To receive a character:
  - Create an `InputStreamReader` from the `Sockets` input stream (in the client)
  - Use `read` to read a single character (as an int)
  - Cast to `char`
  - If -1 is read, the `Stream` has been disconnected
- See `OneCharServer.java` and `OneCharClient.java`

Sending longer messages

- This process is repeated to send longer messages
- When the Server is closed, the client will read -1
- Note that `OutputStreamWriter` can also take a `String`

Useful abstractions

- Dealing with individual `chars` is a pain, especially reading
- Various useful classes exist to make things easier
- `Scanner` can be used with an `InputStream`
- `Server.java` and `Client.java`

Alternatives: `PrintWriter` and `BufferedReader`

- In the Server we can also create a `PrintWriter`:

```
PrintWriter writer = new PrintWriter(
    client.getOutputStream(),true);
```

- Note the `true` - this makes the stream automatically flush
  - It's a buffered stream: things only get sent when the buffer is full, or is flushed.
- In the client we create a `BufferedReader`:

```
BufferedReader reader = new BufferedReader(
    new InputStreamReader(socket.getInputStream()));
```

- `println` and `readLine` perform the necessary reading and writing actions
- `SimpleServer2`, `SimpleClient2`

- What happens if you remove the `true` from the PrintWriter constructor?
- What happens if you add the following to the Server, before the `println`?

```
try {
        Thread.sleep(2000);
    }catch(InterruptedException e) {
    }
```

Allowing multiple connections

- `DateServer` and `DateClient` implement a client-server system where a server periodically sends the data and time to a single client
- Once the connection has been made, the Server enters an infinite loop where it sends a String representation of the Date to the client every 500ms
- The client prints the data every time it is received

- To allow for multiple connections, we need multiple threads in the server (one per client)
- We put the server work (sending the date) into an object that extends `Thread`
- Every time a new connection is accepted, a new Thread is created
- see `DateServer2.java`
- Do we need to change `DateClient`?

Two-way communication

- If we want to be able to send and receive messages from both the client and the server we need more threads.
- On each side, a Thread to read and a Thread to write
- `Reader.java` and `Writer.java` are simple classes that implement `Runnable` for reading and writing

- `WalkyTalkyServer.java` is a server that creates a `Reader` and `Writer` when a client connects
- `WalkyTalkyClient.java` is the same, for a client

Sending other objects - Serializable

- We are not restricted to just sending characters
- Any `Serializable` object can be sent down a `Stream`
- A `Serializable` object is one that implements the `Serializable` interface
  - Normally no methods have to be overwritten
- `ObjectOutputStream` and `ObjectInputStream` allow us to send and receive `Serializable` objects
- `MessageServer.java`, `Message.java` and `MessageClient.java`

Serialiazable

- Any object within the object we are Searalizing must also be Serializable
- If there are attributes that you don't want to encode, use the decorator `transient`
- As well as being transmitted, objects can also be written to files:

```
FileOutputStream fileOutputStream =
    new FileOutputStream("yourfile2.txt");
ObjectOutputStream objectOutputStream =
    new ObjectOutputStream(fileOutputStream);
objectOutputStream.writeObject(e);
objectOutputStream.flush();
objectOutputStream.close();
```

Working in Swing

- Recall that intensive jobs should all be placed in `SwingWorker` objects
- `reader.readLine()` waits until a line can be read
  - this could take a long time
- All client and server operations should be placed within `SwingWorker` objects
- Example: `QuestionServer` and `QuestionClient`

Swing Chat Client

- `ChatServer.java` is a multi-threaded Server than can handle multiple clients
- When it receives a message from one client, it transmits it to all
- `MessageClient` is the client we used in a previous example. It works from the console.
- `SwingChatClient` is a Swing based client that can interact with the same server
- It has a class that extends `SwingWorker` for reading messages