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();
    }
}
```

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

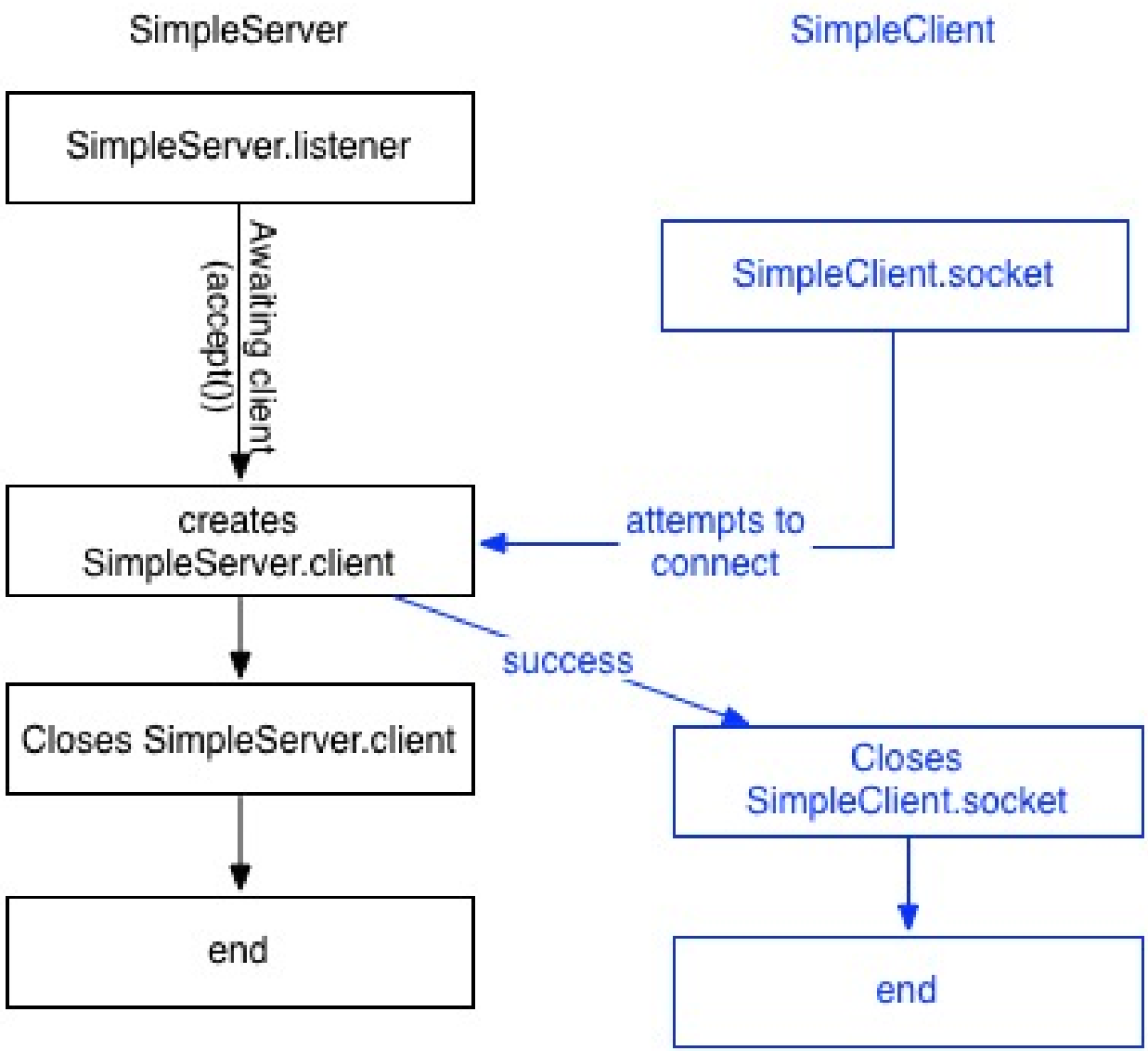

Figure 1:

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 (normally) bytes or chars
  - In Java, objects that are `Serializable` can be transformed into bytes/chars and sent down a stream
  - They can then be read (and cast) at the other end
  - E.g. we will use the fact that `String` is `Serializable`
- We will use the following Stream classes:
  - PrintWriter
  - BufferedReader

Sending a message from the Server to the Client

- In the Server we 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`

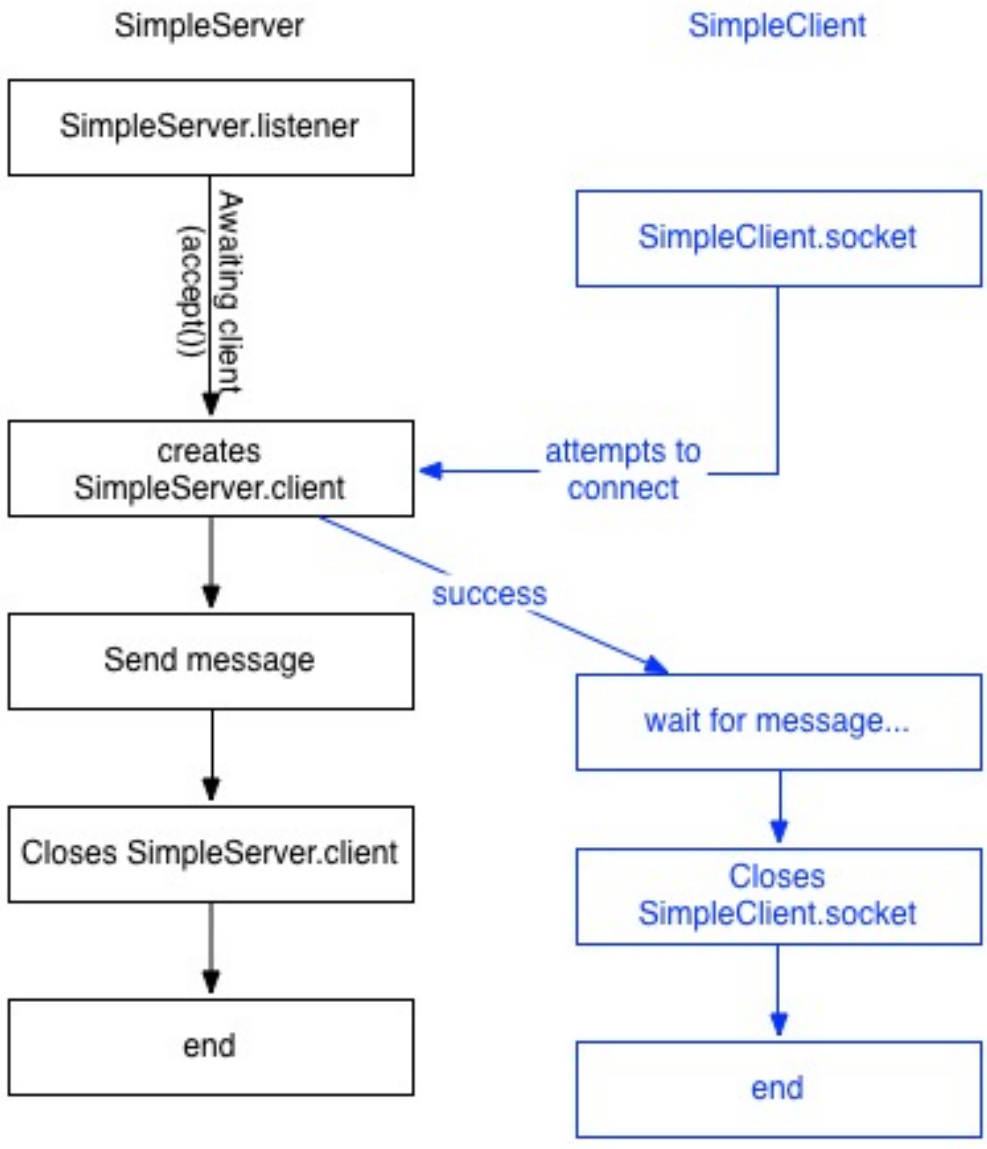
SimpleServer
SimpleClient
SimpleServer.listener
Awaiting client
(accept())
SimpleClient.socket
creates
SimpleServer.client
attempts to
connect
success
Send message
wait for message...
Closes SimpleServer.client
Closes
SimpleClient.socket
end
end

Figure 2:

- 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
- Note the exception handling – it can get quite complicated!
- To allow for multiple connections, we put the server work (sending the date) into an object that extends `Thread`
- `DateServer2`
- When a new connection is accepted, the socket is passed to a new thread object that is then started
  - Do we need to change `DateClient`?

Knowing when a client/server has stopped?

- Often, it will be useful to know when a client/server has left
- The easiest way is by periodically trying to read a line
- If `readLine()` returns `null`, then the other party has gone
- `DateServer3`, `DateClient3`

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`

Design exercise - Chatroom Application