APIT - Distributed Systems

Dr. Simon Rogers

19/02/2018

Contents

Overview

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

Introduction

In previous lectures we saw how to create multiple processes (threads) in a single program, on a single machine. We will now turn our attention to systems that can communicate across machines - distributed systems. These typically take the form of a *server* program running on a machine, and *client* programs running on other machines (or the same machine). Clients make connections with the server and can then send and receive messages according to a pre-defined set of rules (a *protocol*). Some obvious examples are:

- The internet: client programs send requests to web servers and the servers respond by returning files that can be, say, rendered in a browser.
- Email: messages are sent between servers and retrieved by client programs.

As with threads, Java has a lot of built in functionality (`java.net`, `java.io`) that makes programming such systems easier.

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 IOException {
        // 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();
    }
}
```

Note that I have lazily got main to `throw IOException` so that the code will fit onto one slide!

`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 IOException {
        // Make a socket and try and connect
        Socket socket = new Socket(server,PORT);
        // Close the socket
        socket.close();
    }
}
```

What's happening

In this example, `handler` waits for an incoming connection. As soon as one arrives, it creates a `Socket` object. As soon as it creates the socket, it closes it and terminates. The client program creates a socket with the server (note it needs the IP address and port. In this case, as both programs are running on the same machine, we use the IP address 127.0.0.1). Once it has connected, it closes the socket and terminates. i.e. when you run the server program it sits waiting until the client program is run and then terminates.

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

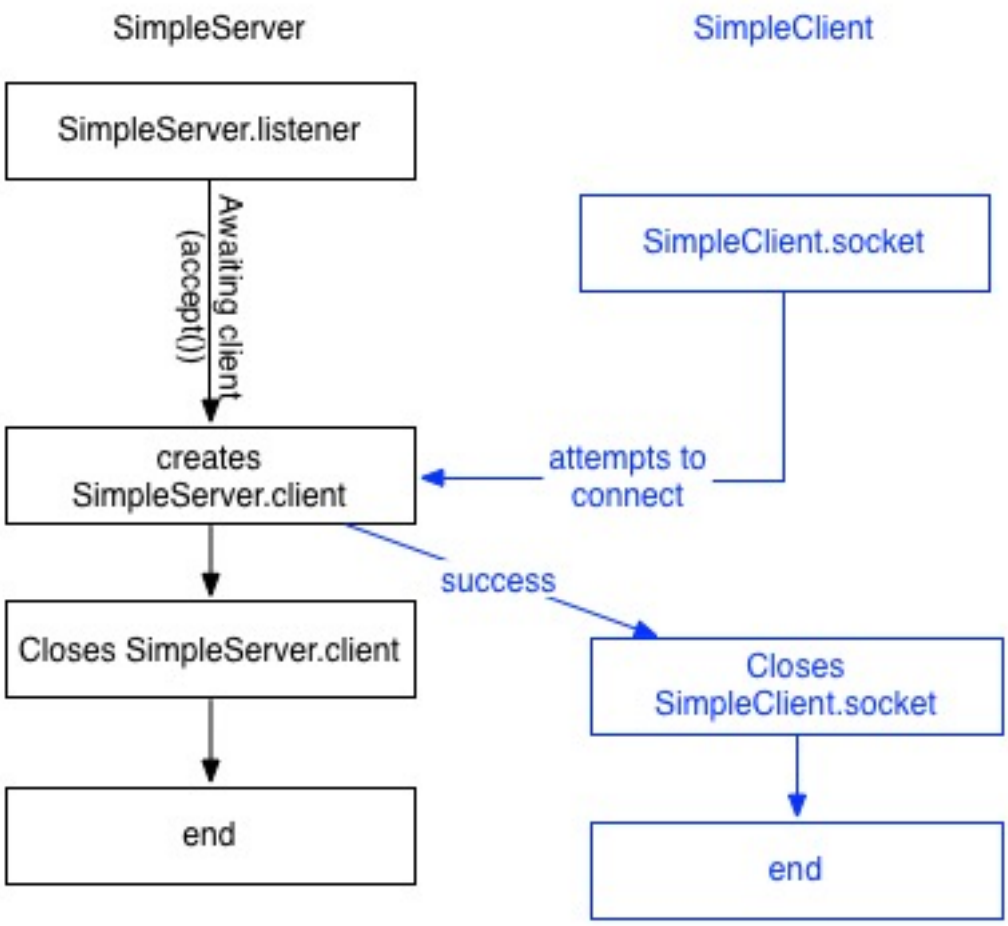
SimpleServer
SimpleClient
SimpleServer.listener
Awaiting client
(accept())
SimpleClient.socket
creates
SimpleServer.client
attempts to
connect
success
Closes SimpleServer.client
Closes
SimpleClient.socket
end
end

Figure 1:

- `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`

```
import java.io.OutputStreamWriter;
public class OneCharServer {
    public static void main(String[] args) {
        ServerSocket listener = null;
        Socket client = null;
        try {
            listener = new ServerSocket(8765);
            client = listener.accept(); // wait for a single client
            System.out.println("Client has arrived!");
            // At this point a client has arrived

            OutputStreamWriter os = new OutputStreamWriter(client.getOutputStream());

            os.write('x');
            os.flush();

            client.close();
            listener.close();
        }catch(IOException e) {
            e.printStackTrace();
        }
    }
}
```

```
import java.io.IOException;
import java.net.Socket;
import java.io.InputStreamReader;

public class OneCharClient {
    public static void main(String[] args) {
        Socket s = null;
        try {
            s = new Socket("127.0.0.1",8765); // IP Address and port
            // We are connected!
            InputStreamReader sr = new InputStreamReader(s.getInputStream());
```

```
            int c = sr.read();
            System.out.println((char)c); // Print the character
            sr.close(); // Close the reader
            s.close(); // close the socket
        }catch(IOException e) {
            e.printStackTrace();
        }
    }
}
```

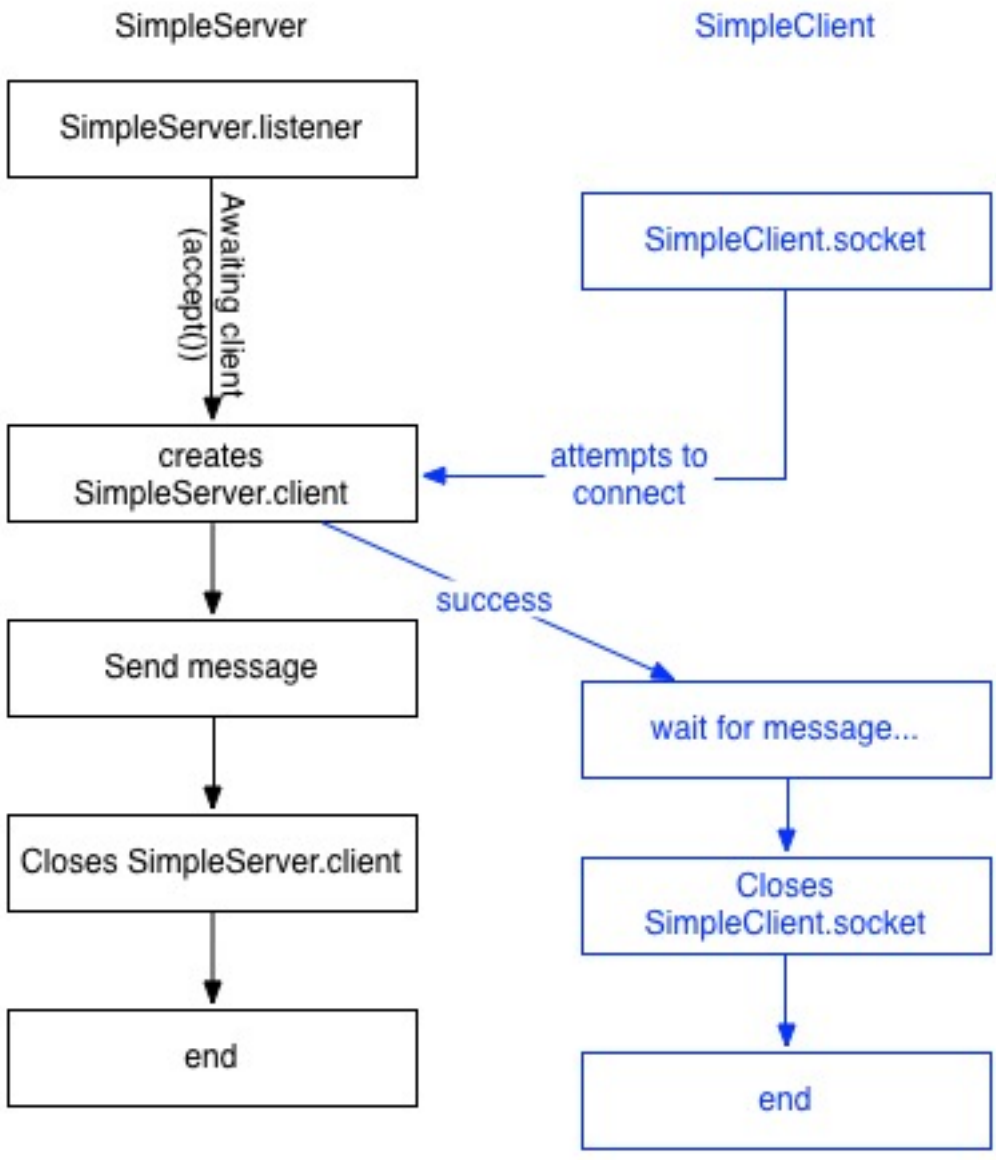

Figure 2:

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 `char`s 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`

```
import java.net.Socket;
import java.net.ServerSocket;
import java.io.IOException;
import java.io.OutputStreamWriter;

public class Server {
    public static void main(String[] args) {
        ServerSocket listener = null;
        Socket client = null;
        try {
            listener = new ServerSocket(8765);
            client = listener.accept(); // wait for a single client
            System.out.println("Client has arrived!");
            // At this point a client has arrived
            OutputStreamWriter os = new OutputStreamWriter(client.getOutputStream());

            Scanner textInput = new Scanner(System.in);
            while(true) {
                String line = textInput.nextLine();
                if(line.equals("END")) {
                    break;
                }
                os.write(line + '\n');
                os.flush();
            }
            textInput.close();
            client.close();
            listener.close();
        }catch(IOException e) {
            e.printStackTrace();
        }
    }
}
```

```
import java.io.IOException;
import java.net.Socket;
import java.util.Scanner;
import java.io.InputStreamReader;

public class Client {
```

```
    public static void main(String[] args) {
        Socket s = null;
        try {
            s = new Socket("127.0.0.1",8765); // IP Address and port
            // We are connected!
            Scanner sr = new Scanner(s.getInputStream());
            while(sr.hasNextLine()) {
                System.out.println(sr.nextLine());
            }
            sr.close(); // Close the reader
            s.close(); // close the socket
        }catch(IOException e) {
            e.printStackTrace();
        }
    }
}
```

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

The previous examples kill the server as soon as it has received and killed the first client connection. If we want to allow multiple clients simultaneously, we can embed the client sockets in threads. We will now look at an example client-server application that where the client periodically sends the current date and time to all clients.

- `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

```
import java.io.*;
import java.net.*;
import java.util.Date;
public class DateServer {
    private static int PORT = 8765;
    private static Socket client;
    private static ServerSocket listener;
    public static void main(String[] args) {
        try {
            listener = new ServerSocket(PORT);
            client = listener.accept();
            PrintWriter out = new PrintWriter(client.getOutputStream(),true);
            while(true) {
                out.println((new Date()).toString());
                Thread.sleep(500);
            }
        }catch (InterruptedException e) {
            e.printStackTrace();
        }catch (IOException e) {
            e.printStackTrace();
        }finally {
            try {
                client.close();
                listener.close();
            }catch(IOException e) {}
        }
    }
}
```

```
import java.net.*;
import java.io.*;
import java.util.Scanner;
public class DateClient {
    private static String server = "127.0.0.1";
    private static int PORT = 8765;
    private static Socket socket;
    public static void main(String[] args) {
        try{
            socket = new Socket(server,PORT);
            Scanner reader = new Scanner(socket.getInputStream());
            while(reader.hasNextLine()) {
                System.out.println(reader.nextLine());
            }
            reader.close();
        }catch(IOException e) {
            e.printStackTrace();
        }finally {
            try {
                socket.close();
```

```
            }catch(IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

- 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`?

```
import java.io.*;
import java.net.*;
import java.util.Date;
public class DateServer2 {
    private static int PORT = 8765;
    private static ServerSocket listener;
    public static void main(String[] args)  {
        try {
            listener = new ServerSocket(PORT);
            while(true) {
                // Socket a = listener.accept();
                // Handler h = new Handler(a);
                // h.start();
                new Handler(listener.accept()).start();
            }
        }catch(IOException e) {
            e.printStackTrace();
        }finally {
            try{
                listener.close();
            }catch(IOException e){
                e.printStackTrace();
            }
        }
    }

    public static class Handler extends Thread {
        private Socket socket;
        public Handler(Socket socket) {
            this.socket = socket;
        }
        public void run() {
            try {
                System.out.println("New connection started on thread " + this.getName());
                PrintWriter out = new PrintWriter(socket.getOutputStream(),true);
                out.println("Hello - welcome to the date server. You're on thread " + this.getName());
                while(true) {
                    String message = (new Date()).toString();
                    out.println(message);
                    Thread.sleep(500);
                }
            }catch(InterruptedException e) {
```

```
                e.printStackTrace();
            }catch(IOException e) {
                e.printStackTrace();
            }finally {
                try {
                    this.socket.close();
                }catch(IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

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

---

```
import java.net.Socket;
import java.io.IOException;
import java.util.Scanner;
public class Reader implements Runnable {
    private Socket socket;
    public Reader(Socket s) {
        this.socket = s;
    }
    public void run() {
        try {
            Scanner sc = new Scanner(this.socket.getInputStream());
            while(sc.hasNextLine()) {
                System.out.println(sc.nextLine());
            }
            sc.close();
        }catch(IOException e) {
            e.printStackTrace();
        }
    }
}
```

```
import java.net.Socket;
import java.io.IOException;
import java.io.OutputStreamWriter;
import java.util.Scanner;
public class Writer implements Runnable {
    private Socket socket;
    public Writer(Socket s) {
        this.socket = s;
    }
```

```
    public void run() {
        try {
            Scanner sc = new Scanner(System.in);
            OutputStreamWriter os = new OutputStreamWriter(socket.getOutputStream());
            String line;
            while(!(line = sc.nextLine()).equals("END")) {
                os.write(line + '\n');
                os.flush();
            }
            sc.close();
            os.close();
        }catch(IOException e) {
            e.printStackTrace();
        }
    }
}
```

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

```
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
public class WalkyTalkyServer {
    public static void main(String[] args) {
        ServerSocket s = null;
        Socket client = null;
        try {
            s = new ServerSocket(8765);
            client = s.accept();
            Thread readThread = new Thread(new Reader(client));
            Thread writeThread = new Thread(new Writer(client));
            readThread.start();
            writeThread.start();
            try {
                readThread.join();
                writeThread.join();
            }catch(InterruptedException e) {
                e.printStackTrace();
            }
        }catch(IOException e) {
            e.printStackTrace();
        }finally {
            try {
                client.close();
                s.close();
            }catch(IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```
import java.net.Socket;
import java.io.IOException;
```

```
public class WalkyTalkyClient {
    public static void main(String[] args) {
        Socket server = null;
        try {
            server = new Socket("127.0.0.1",8765);
            Thread readThread = new Thread(new Reader(server));
            Thread writeThread = new Thread(new Writer(server));
            readThread.start();
            writeThread.start();
            readThread.join();
            writeThread.join();
        }catch(InterruptedException e) {
            e.printStackTrace();
        }catch(IOException e) {
            e.printStackTrace();
        }finally {
            try {
                server.close();
            }catch(IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

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`

The following class, `Message` is an object that we will transmit. It implements `Serializable` but this doesn't require implementing any methods.

```
import java.io.Serializable;
public class Message implements Serializable {
    private String messageText;
    private String senderName;
    public Message(String messageText, String senderName) {
        this.messageText = messageText;
        this.senderName = senderName;
    }
    public String toString() {
        return this.senderName + ": " + this.messageText;
    }
}
```

`Reader` and `Writer` are very similar to the previous example, but now send and receive `Message` objects. Note that the reader has to catch the `ClassNotFoundException` in case the object that appears is of a class that this code doesn't know about.

```
import java.net.Socket;
import java.io.IOException;
import java.io.ObjectInputStream;
public class Reader implements Runnable {
    private Socket socket;
    public Reader(Socket s) {
        this.socket = s;
    }
    public void run() {
        try {
            ObjectInputStream sc = new ObjectInputStream(this.socket.getInputStream());
            Message message;
            while((message = (Message)sc.readObject())!=null) {
                System.out.println(message);
            }
            sc.close();
        }catch(ClassNotFoundException e) {
            e.printStackTrace();
        }catch(IOException e) {
            e.printStackTrace();
        }
    }
}
```

```
import java.net.Socket;
import java.io.IOException;
import java.io.ObjectOutputStream;;
import java.util.Scanner;
public class Writer implements Runnable {
    private Socket socket;
    public Writer(Socket s) {
        this.socket = s;
    }
    public void run() {
        try {
            Scanner sc = new Scanner(System.in);
            ObjectOutputStream os = new ObjectOutputStream(socket.getOutputStream());
            String line;
            System.out.println("What is your name?");
            String name = sc.nextLine();
            while(!(line = sc.nextLine()).equals("END")) {
                os.writeObject(new Message(line,name));
            }
            sc.close();
            os.close();
        }catch(IOException e) {
            e.printStackTrace();
        }
    }
}
```

The server and client are almost identical to the previous example.

```
import java.net.Socket;
public class MessageServer {
```

```
    public static void main(String[] args) {
        ServerSocket s = null;
        Socket client = null;
        try {
            s = new ServerSocket(8765);
            client = s.accept();
            Thread readThread = new Thread(new Reader(client));
            Thread writeThread = new Thread(new Writer(client));
            readThread.start();
            writeThread.start();
            try {
                readThread.join();
                writeThread.join();
            }catch(InterruptedException e) {
                e.printStackTrace();
            }
        }catch(IOException e) {
            e.printStackTrace();
        }finally {
            try {
                client.close();
                s.close();
            }catch(IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```
import java.net.Socket;
import java.io.IOException;
public class MessageClient {
    public static void main(String[] args) {
        Socket server = null;
        try {
            server = new Socket("127.0.0.1",8765);
            Thread readThread = new Thread(new Reader(server));
            Thread writeThread = new Thread(new Writer(server));
            readThread.start();
            writeThread.start();
            readThread.join();
            writeThread.join();
        }catch(InterruptedException e) {
            e.printStackTrace();
        }catch(IOException e) {
            e.printStackTrace();
        }finally {
            try {
                server.close();
            }catch(IOException e) {
                e.printStackTrace();
            }
        }
    }
```

```
}
```

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

The server:

```
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.ArrayList;

public class ChatServer implements Runnable{
    private class ClientRunner implements Runnable {
        private Socket s = null;
        private ChatServer parent = null;
        private ObjectInputStream inputStream = null;
        private ObjectOutputStream outputStream = null;
        public ClientRunner(Socket s,ChatServer parent) {
```

```
            this.s = s;
            this.parent = parent;
            try {
                outputStream = new ObjectOutputStream(this.s.getOutputStream());
                inputStream = new ObjectInputStream(this.s.getInputStream());
            }catch(IOException e) {
                e.printStackTrace();
            }
        }
        public void run() {
            // receive messages
            try {
                Message message = null;
                while((message = (Message)inputStream.readObject())!= null) {
                    this.parent.transmit(message);
                }
                inputStream.close();
            }catch(ClassNotFoundException e) {
                e.printStackTrace();
            }catch(IOException e) {
                e.printStackTrace();
            }
        }
        public void transmitMessage(Message m) {
            try {
                outputStream.writeObject(m);
            }catch(IOException e) {
                e.printStackTrace();
            }
        }
    }
    private ServerSocket server;
    private ArrayList<ClientRunner> clients = new ArrayList<ClientRunner>();
    public ChatServer() {
        try {
            server = new ServerSocket(8765);
        }catch(IOException e) {
            e.printStackTrace();
        }
    }
    public void run() {
        while(true) {
            Socket clientSocket = null;
            try {
                clientSocket = server.accept();
                System.out.println("New client connected");
                ClientRunner client = new ClientRunner(clientSocket,this);
                clients.add(client);
                new Thread(client).start();
            }catch(IOException e) {
                e.printStackTrace();
            }
        }
    }
```

```
    public void transmit(Message m) {
        for(ClientRunner c: clients) {
            if(c != null) {
                c.transmitMessage(m);
            }
        }
    }
    public static void main(String[] args) {
        Thread t = new Thread(new ChatServer());
        t.start();
        try {
            t.join();
        }catch(InterruptedException e) {
            e.printStackTrace();
        }

    }
}
```

The swing client

```
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JTextField;
import javax.swing.SwingWorker;

import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.Socket;

public class SwingChatClient extends JFrame implements ActionListener {

    private class ReadWorker extends SwingWorker<Void,Void> {
        private Socket socket = null;
        private ObjectInputStream inputStream = null;
        private SwingChatClient parent;
        public ReadWorker(Socket s, SwingChatClient parent) {
            this.socket = s;
            this.parent = parent;
            try {
                inputStream = new ObjectInputStream(this.socket.getInputStream());
            }catch(IOException e) {
                e.printStackTrace();
            }
        }
        public Void doInBackground() {
            System.out.println("Started read worker");
```

```
            Message m = null;
            try {
                while((m = (Message)inputStream.readObject())!= null) {
                    System.out.println(m);
                    parent.display(m);
                }
            }catch(ClassNotFoundException e) {
                e.printStackTrace();
            }catch(IOException e) {
                e.printStackTrace();
            }finally {
                return null;
            }
        }
    }

    private Socket server = null;
    private JTextArea textArea;
    private ObjectOutputStream outputStream;
    private JTextField messageField;
    private JButton sendButton;
    private String name = "Sarah";
    public SwingChatClient() {
        this.setSize(800,500);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        name = JOptionPane.showInputDialog(this, "What's your name?");
        JPanel panel = new JPanel(new BorderLayout());
        textArea = new JTextArea(10, 40);
        panel.add(new JScrollPane(textArea),BorderLayout.CENTER);
        this.add(panel);

        JPanel bottomPanel = new JPanel();
        messageField = new JTextField(40);
        sendButton = new JButton("Send");
        sendButton.addActionListener(this);
        bottomPanel.add(messageField);
        bottomPanel.add(sendButton);
        panel.add(bottomPanel,BorderLayout.SOUTH);

        this.setVisible(true);

        connect();

        try {
            outputStream = new ObjectOutputStream(server.getOutputStream());
        }catch(IOException e) {
            e.printStackTrace();
        }

        ReadWorker rw = new ReadWorker(server,this);
        rw.execute();
        System.out.println("HERE");
    }
    private void connect() {
```

```
        try {
            server = new Socket("127.0.0.1",8765);
            System.out.println("Connected");
        }catch(IOException e) {
            e.printStackTrace();
        }
    }
    public void display(Message m) {
        textArea.append(m.toString() + '\n');
    }
    public void actionPerformed(ActionEvent e) {
        if(e.getSource() == sendButton) {
            String messageText = messageField.getText();
            try {
                outputStream.writeObject(new Message(messageText,name));
                messageField.setText("");
            }catch(IOException ex) {
                ex.printStackTrace();
            }
        }
    }

    public static void main(String[] args) {
        new SwingChatClient();
    }
}
```