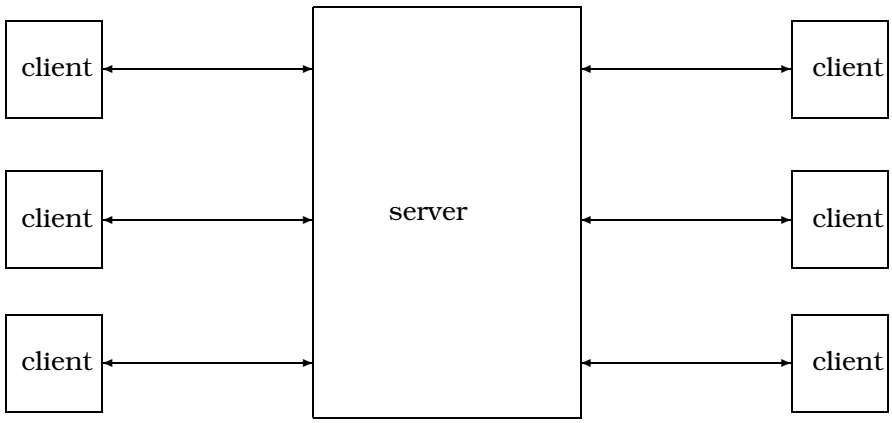
CS1Bh Lecture Note 20

Client/server computing

A modern computing environment consists of not just one computer, but several. When designing such an arrangement of computers it might at first seem that the fairest approach is to make all of the individual computers in the collection equally powerful, with processors of equal speed and the same amount of memory. In this way it would seem that the users of this computer system would be equally well resourced and that the resources are shared out in the fairest possible way.

In practice, such an equal distribution of resources typically does not provide the best service to users. It is frequently the case that users will have occasional tasks which are more time-consuming or memory-hungry than others. Tasks such as these will then be delayed, possibly at a time when many of the other available computers are under-used.

An alternative distribution of resources which helps with this problem is to buy at least one machine which is more powerful than then rest (with a faster processor and with more memory) and have the other machines arranged so that users may connect to the more powerful machine when they need to. With this arrangement the users of the system are provided with occasional access to a more powerful computer than they otherwise would have had. The more powerful machine is called the *server* and the less powerful machines used to connect to the server are called the *clients*.



The *client/server* design provides users with a means to issue commands which are sent across a network to be received by a server which executes their commands for them. The results are then sent back to the client machine which sent the request in order that the user may see the results. Sending the request and receiving the reply

over the network will take time but this expense is often offset by the fact that the server will be able to execute the command faster than the client itself could do, if it would even be able to do it at all, given its lesser memory resources.

In order for client and server to communicate successfully, they must do so according to a set of rules which regulate the form which the communication must take. This set of rules is called a *protocol* and the communications protocol which is most widely used today is TCP/IP (Transport Control Protocol/Internet Protocol). (Another is UDP, the Unreliable Datagram Protocol). TCP/IP provides an addressing mechanism which allows clients to set up connections with servers. The addressing system makes use of *host names* and *port numbers*.

20.1 Hosts and ports

A *host* is a machine which is connected to an IP network. A host can be identified by a *host name* (such as `scar.dcs.ed.ac.uk`) which translates into a numeric IP address (such as 129.215.216.18). The textual form of the name is much easier to remember than the numeric address and can be translated into the numeric address using a service called the Domain Name Service (DNS).

A *port number* is used to communicate with a process which is running on a host. Particular services provided by a host will be associated with a particular port number. For example, port 25 is standardly used for electronic mail and port 80 is used for Web communication using HTTP. On Unix systems (including Linux) the meanings of the assigned ports can be found by examining the file `/etc/services`.

Together, a host name (or IP address) and a port number uniquely identify a particular process running on a particular machine on the network. An analogy is that the IP address is rather like a telephone number, connecting you to a particular location, and the port number is rather like an telephone extension number, connecting you to a particular telephone receiver at that location. Together a host name and a port number allow us to create a *socket*.

20.2 Sockets

A *socket* is a connection to another machine. A server will listen for connections on a particular port. When a client tries to connect to the same port a socket connection is established. The socket behaves as two pairs of an input stream and an output stream. Viewed from the server side, the input stream is read to get commands from the client and the output stream is used to write results back to the client. Viewed from the client side, the output stream is used to write commands to the server and the input stream is used to read the results from the server. Of course these pairs of streams match up to make a bi-directional communication channel between client and server.

Keeping in mind that sockets are made up of input and output streams is useful because we are always aware that input and output operations can fail. Communication between hosts can fail also, perhaps because we do not have permission to connect to a particular host or because we do not have permission to use a particular port.

20.3 Client/server computing in Java

Having met the concepts of client/server computing, we now progress to discovering how to implement these systems in Java. We will make use of classes from the `java.io` package, as we might expect, but we will also make use of classes from Java's networking package `java.net`. As an example here we will consider constructing a simple messaging service, allowing the client to send messages to be printed on the console of the server. A reply is sent back to acknowledge that the message has been received.

The system is structured as two separate Java programs, each with a `main()` method which is invoked from the Java interpreter, initiated with the `java` command. Thus there are two separate instances of a Java virtual machine executing at the same time. One runs on the server. The other runs on the client machine. No memory is shared between the client and the server and so information cannot be communicated via *shared variables* which both the client and the server can access. The only option is that information must be sent as 'messages'. This method of communication between co-operating programs is known as *message passing*.

20.3.1 Starting the server

We begin by initiating the `Server` program on the server, which for the sake of definiteness, we will take to be `scar.dcs.ed.ac.uk`, using port 5055, which is not needed for any other purpose.

```
[scar]stg:  java Server
Listening on port 5055
```

20.3.2 Starting the client

At this point we can run the client program on another machine (perhaps using the machine `bleaurgh.dcs.ed.ac.uk`). The `Client` program assumes that it is communicating with `scar` on port 5055 but it could be easily made more general by accepting the host name and port number as command line arguments. After an inital pause while connection is established we eventually receive acknowledgements to our messages as shown below. Simultaneously, the messages are received by the server.

```
[scar]stg:  java Server
Listening on port 5055
First message sent
Second message sent
```

```
[bleaurgh]stg:  java Client
Message received
Message received
```

The program could easily be turned into a simple `talk` application, allowing sentences typed at one side to be sent to the other and replies to be sent back similarly.

20.4 The server program

The following program is run on the server. Lines 7 to 13 allow a server socket to be created, and ensure that the program will exit with an explanatory message if this socket could not be created. Lines 15 to 21 establish a client connection to this socket, where an error will again cause the server to exit with an explanatory error message. In lines 29 to 35 lines of text are read repeatedly and acknowledged until a null `String` object is received.

```
import java.io.*;                                                    // 1
import java.net.*;                                                   // 2
                                                                     // 3
class Server {                                                       // 4
   public static void main   (String[] args)                         // 5
          throws IOException {                                       // 6
      ServerSocket sock = null;                                      // 7
      try {                                                          // 8
         sock = new ServerSocket(5055);                              // 9
      } catch (IOException e) {                                      // 10
         System.out.println ("Could not listen: " + e);              // 11
         System.exit(1);                                             // 12
      }                                                              // 13
      System.out.println ("Listening on port 5055");                 // 14
      Socket clientSocket = null;                                    // 15
      try {                                                          // 16
         clientSocket = sock.accept();                               // 17
      } catch (IOException e) {                                      // 18
         System.out.println ("Accept failed: " + e);                 // 19
         System.exit(1);                                             // 20
      }                                                              // 21
      // Communication has been established                         // 22
      InputStreamReader is = new InputStreamReader                   // 23
         (clientSocket.getInputStream());                            // 24
      BufferedReader input = new BufferedReader(is);                 // 25
      PrintWriter client = new PrintWriter                           // 26
         (clientSocket.getOutputStream());                           // 27
                                                                     // 28
      String line = input.readLine();                                // 29
      while (line != null) {                                         // 30
         System.out.println (line);                                  // 31
         client.println ("Message received");                        // 32
         client.flush();                                             // 33
         line = input.readLine();                                    // 34
      }                                                              // 35
   }                                                                 // 36
}                                                                    // 37
```

20.5 The client program

The following program is run on the client. It sends only two message strings to the server, which it assumes to be running on `scar.dcs.ed.ac.uk`, listening on port 5055. Lines 9 to 11 establish a buffered input stream which allows replies from the server to be read by invoking the `readLine()` method of the `input` object. This object is of the `BufferedReader` class and has been created by getting the input stream object from the socket. This object, which is seen as the input stream from the client side, is seen as the output stream from the server side. The strings which are returned as the result of the `readLine()` method invocations are printed on the console of the client machine by using the `println()` method provided by the `System.out` object.

```
import java.io.*;                                                    // 1
import java.net.*;                                                   // 2
                                                                     // 3
class Client {                                                       // 4
   public static void main   (String[] args)                         // 5
           throws IOException {                                      // 6
      Socket sock = new Socket ("scar.dcs.ed.ac.uk", 5055);          // 7
      // Communication has been established                         // 8
      InputStreamReader is = new InputStreamReader                   // 9
         (sock.getInputStream());                                    // 10
      BufferedReader input = new BufferedReader(is);                 // 11
      PrintWriter server = new PrintWriter                           // 12
         (sock.getOutputStream());                                   // 13
                                                                     // 14
      server.println("First message sent");                         // 15
      server.flush();                                                // 16
      System.out.println(input.readLine());                          // 17
      server.println("Second message sent");                        // 18
      server.flush();                                                // 19
      System.out.println(input.readLine());                          // 20
   }                                                                 // 21
}                                                                    // 22
```

Careful thought is required to ensure that a pair of communicating programs such as these will work together. The messages written by one side need to be read by the other in order for the communication to succeed. One the server side, we see that it first reads a message and then writes an acknowledgement. On the client side we see that it first writes a message and then reads an acknowledgement. Symmetric communication such as this will succeed, but it is easy to mis-match the communication, in which case both parties in the communication can be left waiting to read a message sent by the other. Such *deadlocks* are often resolved by limiting communication to be received within a certain time. This solution deals also with problems due to one machine or the other suffering a software or hardware crash. This form of planned timed interrupt is called a *time-out*.

20.6 Communicating objects

Message passing is a simple form of communication between programs which run on different computers. In particular, it does not exploit the fact that both of the programs which are used here are Java programs, with a common means of representing objects in memory. Because of this, we cannot communicate by message passing any objects more complex than strings. Java does provide a means of communicating general objects between instances of Java virtual machines running on different hosts where the objects to be communicated are serialised on the sender's side and sent across the network to be deserialised on the receiving side. This form of communication is called *remote method invocation (RMI)*. However, this method of remote communication is too complex for us to study in Computer Science 1 and we will postpone the study of remote method invocation until later years of the course.

Stephen Gilmore, 2001/04/16 15:41:17.