

# CS1Bh Lecture Note 16

## Case Study: Client/Server

In this case study we look at providing a slightly more realistic client/server application and explore some of the issues that arise in this context. Specifically we:

1. Explore how to use the methods described in Note 15 to provide a simple client/server system to deliver the simple student database described Computer Science 1Ah.
2. Provide a more detailed description of the problems that can arise in systems involving independently running programs and discuss how these problems might be avoided..
3. Discuss what is needed to provide a genuine client/server solution over TCP/IP and outline some of the issues that arise in that context.

### A simple database server

In Computer Science 1Ah course we considered the implementation of a simple student database. Here we combine the code developed for the database with that developed in Note 15 to provide the outline of a database server for student information. The Server class is very similar to that provided in Note 15:

```
import java.io.*;
import java.net.*;

class Server {
    public static void main (String[] args) throws IOException {
        ServerSocket sock = null;
        Class2 cs1 = new Class2();
        while (true) {
            try {
                sock = new ServerSocket(5055);
            } catch (IOException e) {
                System.out.println ("Could not listen on port: " + e);
                System.exit(1);
            }
            System.out.println ("Listening on port 5055");
            Socket clientSocket = null;
```

```

    try {
        clientSocket = sock.accept();
    } catch (IOException e) {
        System.out.println ("Accept failed: " + e);
        System.exit(1);
    }
    // Communication has been established

    Service dialog = new Service(cs1);
    dialog.serve(clientSocket);
    clientSocket.close();
    System.out.println ("Client finished -- await new request.");
    sock.close();
    sock = null;
}
}
}

```

The difference between this server and that of Note 15 is that this server never halts. Instead it repeatedly loops creating a new server socket, accepting a client connection, servicing that connection by creating a new `Service` object, then closing socket after that connection has been terminated by the client.

The `Service` class handles a particular connection with a client. The code is:

```

import java.io.*;
import java.net.*;

class Service {
    private Class2 group;

    public Service(Class2 c) {
        this.group = c;
    }

    public void serve(Socket clientSocket) throws IOException {
        InputStreamReader is =
            new InputStreamReader(clientSocket.getInputStream());
        BufferedReader input = new BufferedReader(is);
        PrintWriter client =
            new PrintWriter(clientSocket.getOutputStream());

        String line = input.readLine();
        while (!line.equals("finish")) {
            if (line.equals("add")) {
                client.println("OK"); client.flush();
                Student student = new Student();
                student.surname = input.readLine();
                student.forename = input.readLine();
                System.out.println("Added: " +

```

```

        student.surname +
        ", " +
        student.forename);
    group.insert(student);
} else if (line.equals("find")){
    client.println ("OK"); client.flush();
    Student student = group.find(input.readLine());
    if (student == null) {
        client.println("Sorry - no such person.");
        client.flush();
    } else {
        client.println(student.forename);
        client.flush();
    }
} else {
    client.println ("NOK"); client.flush();
}
line = input.readLine();
}
}
}

```

In order for the server to provide the required service the client is expected to follow a *protocol*. A protocol is just the expected pattern of interaction. This is often the subject of standardisation in order to ease common interactions. For example, communication via IR ports on PDAs usually follows a standard protocol that allows users to share data between equipment provided by different manufacturers. Here the protocol is:

1. The client sends the server a message with the name of the service it would like to use next. Here the server will serve requests to add or find records. If the client requests to finish then the interaction is over and the server will set up a new socket and await a connection request.
2. The server responds with OK if it will provide the requested service and NOK otherwise.
3. If the client gets the OK response it provides the necessary data and awaits the result of the operation. In the case of adding records, the surname and forename are required and no data is returned. In the case of a find, the surname is required and the forename of the corresponding student is returned.

The following program is a very simple client application that makes use of the server. Once this program has run the server closes the associated socket and begins listening for the next client connection. So, for example, this client can be run repeatedly and the server will service the requests contained in the client program.

```
import java.io.*;
import java.net.*;

class Client {
    public static void main (String[] args) throws IOException {
        Socket sock = new Socket ("localhost", 5055);
        // Communication has been established
        InputStreamReader is = new InputStreamReader
            (sock.getInputStream());
        BufferedReader input = new BufferedReader(is);
        PrintWriter server = new PrintWriter
            (sock.getOutputStream());
        // Add Mike F.
        server.println("add");
        server.flush();
        System.out.println(input.readLine());
        server.println("Fourman");
        server.flush();
        server.println("Mike");
        server.flush();

        // Add Paul J
        server.println("add");
        server.flush();
        System.out.println(input.readLine());
        server.println("Jackson");
        server.flush();
        server.println("Paul");
        server.flush();

        // Add Murray C.
        server.println("add");
        server.flush();
        System.out.println(input.readLine());
        server.println("Cole");
        server.flush();
        server.println("Murray");
        server.flush();

        // Find Paul.
        server.println("find");
        server.flush();
        System.out.println(input.readLine());
    }
}
```

```

server.println("Jackson");
server.flush();
System.out.println(input.readLine());

// Try to find Stuart.
server.println("find");
server.flush();
System.out.println(input.readLine());
server.println("Anderson");
server.flush();
System.out.println(input.readLine());

// Ask for a non-existent service.
server.println("foind");
server.flush();
System.out.println(input.readLine());

// Finish up.
server.println("finish"); server.flush();
}
}

```

## Common programming errors

In client/server applications we have more than one independently executing program. Synchronisation occurs between programs when one program reads on a stream provided by another program. The reading program *blocks* on that read until data is written to the stream. Coordinating the actions of a number of programs by such synchronising operations can be difficult to achieve. In particular, we see new kinds of errors arising from interaction between programs rather than from inside one or other of the programs. Two common kinds of errors are:

**Race conditions:** These errors arise when the actions of two programs are left unsynchronised and the outcome of their interaction is a result of their relative speed of operation. Since this depends on various changing conditions, the overall system can behave unpredictably. For example, if two clients are left to compete for access to a single server the behaviour of the system could depend on which client establishes communication with the server first. Such errors can be very hard to detect because the behaviour of the system under test may be predictable (because the test environment may always ensure one particular client wins the race) but the system may be unpredictable in the field where circumstances are more variable.

**Deadlock:** This arises when there is a circular wait for resources in a system. For example, in our simple client/server application if the client is waiting for the server to reply and the server is waiting for the client to reply both will wait forever. Ensuring deadlock freedom is a complex subject but there is a simple strategy to avoid deadlock in systems:

1. Number each resource in the system that more than one program competes for, giving a different number to each resource.
2. Design a system where each program can acquire and release resources as they are needed.
3. Ensure that all programs acquire resources in ascending order. I.e. a program cannot ask for a lower numbered resource than the maximum of the numbers of the resources it controls at any time.

For example, suppose the two resources are (1) that server A will respond to a message and (2) that server program B will respond to a message. Suppose that two clients require both resources to do a task, then because they are required to allocate the resources in the same order they will compete over resource (1), one of the clients will win and will then be able to acquire resource (2) without any competition from the losing client. The ordering principle ensures no circular waits can arise.

## Depending on a server

The server we have constructed is very simple. Constructing a good server is difficult because they are intended to operate in an open environment where the possibility of erroneous clients or malicious attack cannot be ruled out. To construct a dependable server requires many features. The particular choice of features depends on the context. We list a few to start you thinking about the issues.

## Public Key Encryption

An important technology used by any dependable server is an encryption method. This allows us to ensure certain facts about users and messages. One of the most widely used is *public key encryption*. This is a method whereby a user of the encryption system has keys that allow the encoding and decoding of messages. The keys come in pairs. One key is called the *public key* it is widely known, the other is the *private key* and is private to the user. Messages encoded with the public key can be decoded only with the private key, messages encoded with the private key can only be decoded with the public key. This means that to send a private message to someone I encode my message with their public key and send it to them. They can then decode it with their private key. If a person wants to send a public message that is known to come from them then they encode the message with their private key and send it. If the recipient can decode the message with the sender's public key then they know the message was indeed sent by the sender. Combinations of these two kinds of message allow the development of elaborate security protocols.

## Important features in server design

The following features are reasonably important in server design. This is not intended to be an exhaustive list but it does raise many issues in server design.

**Robustness:** The server should not expect client code to be error free. In particular, it should be impossible for client code to lead to a deadlock by failing to follow the established protocol. If need be the server should employ timeouts or other methods to ensure it provides a continuing service.

**Authentication:** It is often the case that only some clients should be permitted to use a particular service. Passwords cannot be sent as plain text across open networks. Can you find a way to authenticate a user using public key encryption (assuming the server knows the users' public keys).

**Secrecy:** Data needs to be encrypted to keep data private when it is sent over public networks. The use of simple encryption may not be enough to guarantee security of data. For example a malicious attack may involve "replaying" previous encrypted messages to confuse or disrupt the system.

**Availability:** Denial of service attacks are particularly problematic. These can involve very large numbers of requests coming from a number of coordinated sites. The server should attempt to identify such attacks and ignore messages involved in the attack.

**Timeliness:** Often servers are required to respond within tight time deadlines in order to ensure quality of service. This can be problematic, in particular where the load on the server can be very variable.

## Summary

- Providing a service by passing messages between the client and the server requires that we establish a *protocol* that describes the pattern of messages that we expect to exchange. Failure to observe the protocol will lead to unexpected consequences.
- Carefully designed protocols allow us to detect and recover from errors in client/server applications. This can also contribute to easing problems in the development and debugging of client applications
- Applications involving two or more independently running programs are subject to new classes of behaviour. In particular *race conditions* and *deadlock* can be difficult to reproduce, identify and resolve.
- Providing a *dependable* server can considerably increase the design complexity of the server.

## Questions

1. How would you extend the example server to offer more of the facilities provided by the `Class2` class?
2. How would you improve the design of the server to make it less susceptible failure provoked by badly written client code?
3. Is it possible to relax the discipline suggested for avoiding deadlock in distributed systems? What is your scheme?
4. Using public key encryption as your encryption technique can you design a login/authentication procedure that means both parties are reasonably confident of one another's identity.

*Note written by Stuart Anderson.  
David Aspinall, 2003/03/02 14:47:39.*