8 Client-Server: Deadlock Avoidance by Design

Chapter 7 demonstrated with two examples, one obvious and the other less so, that deadlocked systems can be constructed quite easily even if the thought given to the design would suggest otherwise. A design pattern is required that ensures deadlock and also livelock freedom. Brinch Hansen [ref] formulated a design approach for operating systems in the 1970s based upon a client-server architecture. It is a slightly updated version of that design approach that is presented here as the client-server design pattern. It is captured in two simple rules, together with a method for analysing a network.

1. A client process that issues a request to a server process guarantees to accept any response from that server immediately. A client – server interaction requires a client request upon the server but it is not necessary for there to be a communication from the server to the client process.

2. A server process that accepts a request from a client process guarantees to return a response to the client process within finite time. In addition, a server process will never send a message to any of its clients without having first received a request from a client. A server process can behave as a client to another server process.

3. Deadlock and livelock will not occur in such a network of client and server processes provided a labelling of the client and server ends of the interactions between processes does not result in a completed circuit of clients and servers.

8.1 Analysing the Queue Accessing System

The `Queue` system discussed previously in Section 6.2 is, in fact, an example of a system that implements the above set of client – server rules.

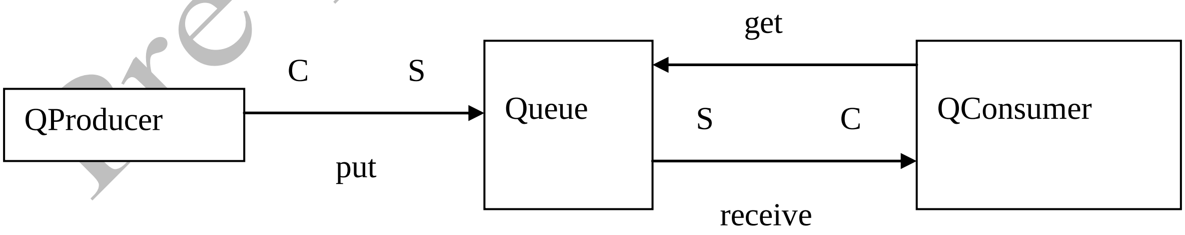

Figure 8-1 Client - Server Labelling of the Queue Processing System

The process `QProducer` acts as a client to `Queue` process, which acts as a server. This interaction does not involve a return communication from `Queue` to `QProducer`. `QConsumer` also acts as a client to the `Queue` process but in this interaction the `QConsumer` process does expect a response. The client behaviour of the `QProducer` process is captured in the following code snippet taken from Listing 5-6. The client request is captured in the `write` method on the channel `put` {2}.

```
01        for ( i in 1 .. iterations ) {
02          put.write(i)
03        }
```

Similarly, the client behaviour of `QConsumer` is shown in the following snippet and is taken from Listing 5-7. The client request is captured in the `write` on the `get` channel {5}, which can be simply interpreted as a signal for the `Queue` process. As a client the `QConsumer` must be ready to receive a response from the `Queue` process as soon as it is available. This is simply achieved by `read`ing the response on the `receive` channel {6}. Crucially, no other processing takes place between the request {5} and the response {6}.

```
04        while (running) {
05          get.write(1)
06          def v = receive.read()
07        }
```

The `Queue` process, see Listing 5-8, simply alternates over the `put` and `get` channels and thus can never generate an output of its own accord and thus behaves as a server for both its interactions. The network, shown in Figure 8-1, contains no circuits and thus is guaranteed to be free from deadlock and livelock.

8.2 Client and Server Design Patterns

The behaviour of processes that implement the Client and Servers patterns is given in the following design templates.

```
08    class ClientTemplate implements CSProcess {

09      def ChannelOutput request
10      def ChannelInput  response        // may not be required

11      void run() {
12        // initialise
13        while (true) {
14          // create server request object
15          request.write ( requestObject ) // could be a signal
16          result = response.read()        // may not be required
17          // process result
18        }
19      }
20    }
```

Template 8-1 Client Design Template

A process that behaves as a client (Template 8-1) will have an output channel upon which it makes `requests` to its server {2}. It will probably have an input channel upon which it `receives responses` {3} from the server but this may not always be necessary. A client process may undertake some initialisation {5} before entering the main loop of the process {6}. Depending upon the nature of the interaction the client process will either create a `requestObject` {7} or cause a signal to be sent to the server {8}if no explicit data is required for the server to respond to the client process. The client process will immediately wait for the response from the server if there is one {9}. The process will then continue processing.

```
21     class ServerTemplate implements CSProcess {

22       def ChannelInput request
23       def ChannelOutput response        // may not be required

24       void run() {
25         // initialise
26         while (true) {
27           def requestObject = request.read()  // may be a signal
28           // process requestObject
29           // determine any result, may require request to another server
30           response.write(result)        // may not be required
31           // update any internal state
32         }
33       }
34     }
```

Template 8-2 Server Design Template

The server template (see Template 8-2) indicates that a server process requires a request {15} input channel and may have a response {16} output channel if there is an explicit result written to the corresponding client process. The server may undertake some initialisation {18} after which it enters the main loop of the process {19}. The server responds to client requests by either reading some form of `requestObject` or a signal {20}. The nature of the request is determined unless that is implicit as a result of receiving a signal {21}. The server then determines the result, which may require access to another server {22} after which the result is written to the client {23}. The server then may have to update some internal state {24} before repeating the loop.

By inspection we can determine that the client template does implement the behavioural requirements of a client given previously in that once it has made a request on a server it is immediately ready to receive the response from that server. Similarly the server process template shows that the server will respond in finite time. Two cases need to be considered. If the server makes no request to another server then the response must be fully determined within the server process and thus this can be completed in finite time as there will be no other communication, which is the only source of indefinite delay, provided the computation is finite. If the server process makes a client style request on another server then provided the requested server maintains the client server contract then this server can respond in finite time.

8.3 Analysing the Crossed Servers Network

Figure 8-2 shows the client-server labelling of the network shown previously in Figure 7-2. The relationship between the Client and Server processes is not the problem and in fact by inspection of the Client process, Listing 7-3 lines {39-40}, it can be seen that this process does in fact implement client behaviour.

The problem lies with the `Server` processes, where there is a circuit from one `Server` to the other and back again. Even if we implemented the `Server`, Listing 7-4, so that it did not interleave access, it would still deadlock. In the implementation, shown in Listing 7-4, the chance of deadlock occurring is compounded because the `Server` does not wait for a response from the other `Server` but starts another client interaction. A simplistic solution could be attempted by making the Server process act as a client when it is accessing the other Server. This will decrease the incidence of deadlock but it will still happen less frequently, which perhaps makes it even more annoying for the `Client` processes. Thus a different solution has to be found.

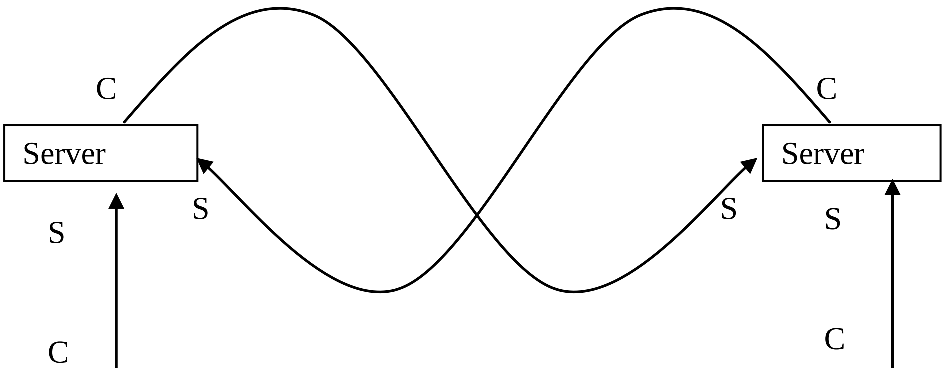

Figure 8-2 Client - Server Labelling of the Crossed Server Network

8.4 Deadlock Free Multi-Client and Servers Interactions

Figure 8-3 shows a solution to the problem that is achieved by the use of a multiplexer. A multiplexer is a process that accepts inputs from a number of input channels and then outputs these input communications on a single output channel. A set of simple multiplexers is described in Appendix ??.

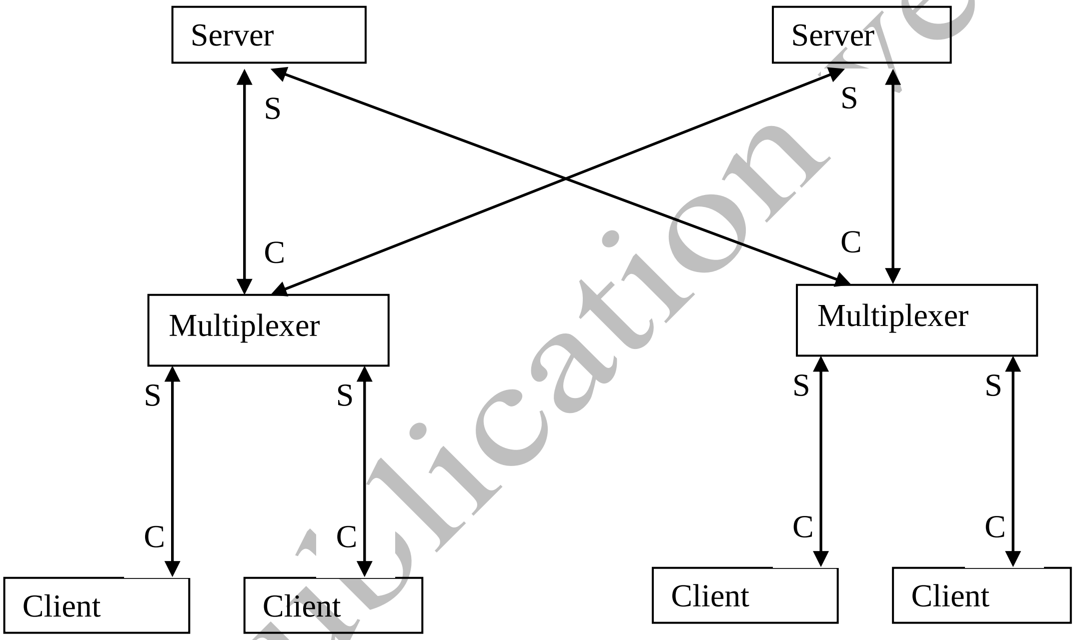

Figure 8-3 Multi-Client and Server Network

A Client makes a request upon a Multiplexer behaving as a server, which contains the data required to determine upon which Server the required data resides. The Multiplexer then behaves as a client and makes a request to the required Server. The corresponding data is then returned from the Server to the Multiplexer and then finally the Client receives the data it requested. By inspection of the network we can see there are no circuits of client server labelling and hence the network will be deadlock free, provided the processes are implemented in a manner that respects the rules of the client-server design pattern.

8.4.1 The Multiplexer Process

Listing 8-1 shows the multiplexer used in this system. `CSMux` is more complex than the simple multiplexer concept described previously. Requests are received from `Clients` by `CSMux` behaving as a server and then `CSMux` determines the `Server` upon which the required data is to be found. The request is then forwarded to a `Server` by the `CSMux` behaving as a client. `CSMux` waits for the response from a `Server` that it then returns to the original `Client` behaving as a server. `CSMux` utilises properties of type `ChannelInputList` and `ChannelOutputList`. These are two helper classes created as part of the

Groovy Parallel capability, see Appendix 2. As their names suggest these provide lists of channel input ends and channel output ends respectively.

```
08     class CSMux implements CSProcess {

09       def ChannelInputList inClientChannels
10       def ChannelOutputList outClientChannels
11       def ChannelInputList fromServers
12       def ChannelOutputList toServers
13       def serverAllocation = [ ]

14       void run() {
15         def servers = toServers.size()
16         def muxAlt = new ALT (inClientChannels)

17         while (true) {
18           def index = muxAlt.select()
19           def key = inClientChannels[index].read()
20           def server = -1

21           for ( i in 0 ..< servers) {
22             if (serverAllocation[i].contains(key)) {
23               server = i
24               break
25             }
26           }

27           toServers[server].write(key)
28           def value = fromServers[server].read()
29           outClientChannels[index].write(value)

30         }
31       }
32     }
```

Listing 8-1 The Multiplexer Coding

The `ChannelInputList inClientChannels` {9} is a list of input channel ends from each of the `Clients` connected to `CSMux`. Similarly, `fromServers` {11} is a list of the channel input ends coming from each of the `Servers` connected to `CSMux`. The list of out channels ends that connects `CSMux` to its `Clients` is contained in the property `outClientChannels` {10} and the outputs from `CSMux` to the connected `Servers` is passed as property `toServers`{12}. The property `serverAllocation` is a `List` of `Lists` such that each internal `List` contains the keys of the values held respectively in each `Server`. There is one list element per `Server` in `serverAllocation` {13}. The `size()` method is used to find the number of `Servers` {15} because there must be as many `Server` processes as there are channels in `toServers`. The alternative, `muxAlt` {16}, is simply constructed from the `inClientChannels ChannelInputList`.

Within the loop {17-30}, the `index` of the enabled alternative is `selected` {18} and its value used to read a `key` value from the corresponding element of `inClientChannels` {19}. The `for` loop {21-26} is used to determine in which `server` the `key` is located. The value of the `key` is written to the `server` element of the `ChannelOutputList toServers` {27}. As this is the start of a client style interaction the `value` corresponding to the `key` is `read`, as soon as it is available on the `server` element of the `ChannelInputList fromServers` {28}. This maintains the client-server relationship between `CSMux` and the `Server`. The `value` is then written to the `index` element of `outClientChannels` {29} thereby completing the server style interaction between `CSMux` and the originating `Client` process.

This interaction typifies a more complex client and server interaction whereby the client makes a request on a server style process which then becomes a client to another server. This can be undertaken as many times as the application requires and provided there are no circuits in the clients and servers is guaranteed to be deadlock free, provided the processes implement the client and server behaviours as defined previously.

8.4.2 The Server Process

The coding of the `Server` process is shown in Listing 8-2. A `Server` has two channel list properties, one, `fromMux` is the input channels from `CSMux` {34} and the other {35}, `toMux`, provides the output channels to `CSMux`. The property `datamap` {36} is used to hold the `Map` of keys and values held within this `Server`. An alternative `serverAlt` is used to alternate over the `fromMux` input channels {38}. Once an enabled alternative has been selected {40}, its `index` is used to read the `key` value from the corresponding element of `fromMux` {41}. This `key` value is then used to access `dataMap`, the value of which is written to the related `CSMux` process using the `index` element of `toMux` {42}.

This simple interaction implements the simplest form of server behaviour, whereby the server reads the request and responds immediately to the request with the required data value.

```
33     class Server implements CSProcess{

34       def ChannelInputList fromMux
35       def ChannelOutputList toMux
36       def dataMap = [ : ]

37       void run() {
38         def serverAlt = new ALT(fromMux)

39         while (true) {
40           def index = serverAlt.select()
41           def key = fromMux[index].read()
42           toMux[index].write(dataMap[key])
43         }
44       }
45     }
```

Listing 8-2 The Server Process Definition

8.4.3 Exercising the System of Clients and Servers

Listings 8-3 to 8-6 give the parts of the script that cause the system of `Clients`, `CSMux` processes and `Servers` to be invoked and permit the number of `Clients` per `CSMux` to be varied at run time. The number of `Servers` is limited to 2, identified as `Server` zero and `Server` one. Similarly there are two `CSMux` processes providing the multiplex capability, referred to as `CSMux` zero and one respectively. In particular, it can be seen that the `Client` process definition used in Chapter 7 is reused {46}, further reinforcing that it already implemented the required client behaviour.

Listing 8-3 initially obtains the number of `clients` {48} and then uses this value together with `servers` {48} to create a set of `One2OneChannel` arrays {49-56}. The naming convention uses C to refer to a `Client` connection, M a `CSMux` connection and S a `Server` connection. Thus, `M0ToC0` {52} provides the connection from `CSMux` zero to the `Clients` attached to that multiplexer and `M1ToS` connects `CSMux` one to both `Servers`.

```
46     import c7.Client

47     def clients = Ask.Int ("Number of clients per server; 1 to 9 ? ", 1, 9)
48     def servers = 2

49     One2OneChannel[] C0ToM0 = Channel.createOne2One (clients)
50     One2OneChannel[] M0ToC0 = Channel.createOne2One (clients)
51     One2OneChannel[] C1ToM1 = Channel.createOne2One (clients)
52     One2OneChannel[] M1ToC1 = Channel.createOne2One (clients)

53     One2OneChannel[] M1ToS = Channel.createOne2One (servers)
54     One2OneChannel[] M0ToS = Channel.createOne2One (servers)

55     One2OneChannel[] S0ToM = Channel.createOne2One (servers)
56     One2OneChannel[] S1ToM = Channel.createOne2One (servers)
```

Listing 8-3 Channel Declarations

Listing 9-4 demonstrates how these channel arrays are converted to instances of `ChannelInputList` and `ChannelOutputList` by simply calling the constructor of the required class {57-64}. For the channels lists that provide the cross connections between the `Server` and `CSMux` processes we first create new, empty, instances of the necessary channel lists, to which the required channel elements are `appended` {65-76}. The regularity of the coding arises because all the elements at one end of a channel list have to be allocated to different processes whereas they all accessed by a single process at the other end.

```
57      def clientsToM0 = new ChannelInputList (C0ToM0)
58      def clientsToM1 = new ChannelInputList (C1ToM1)

59      def M0ToClients = new ChannelOutputList(M0ToC0)
60      def M1ToClients = new ChannelOutputList(M1ToC1)

61      def Mux0ToServers = new ChannelOutputList(M0ToS)
62      def Mux1ToServers = new ChannelOutputList(M1ToS)

63      def Server0ToMuxes = new ChannelOutputList (S0ToM)
64      def Server1ToMuxes = new ChannelOutputList (S1ToM)

65      def Server0FromMuxes = new ChannelInputList()
66      Server0FromMuxes.append(M0ToS[0].in())
67      Server0FromMuxes.append(M1ToS[0].in())

68      def Server1FromMuxes = new ChannelInputList()
69      Server1FromMuxes.append(M0ToS[1].in())
70      Server1FromMuxes.append(M1ToS[1].in())

71      def Mux0FromServers = new ChannelInputList ()
72      Mux0FromServers.append(S0ToM[0].in())
73      Mux0FromServers.append(S1ToM[0].in())

74      def Mux1FromServers = new ChannelInputList ()
75      Mux1FromServers.append(S0ToM[1].in())
76      Mux1FromServers.append(S1ToM[1].in())
```

Listing 8-4 Channel List Definitions

Listing 8-5 shows the definition of the `Map` data structures {77-80} that are passed to each `Server` instance as the property `dataMap` {36}. The `serverKeyLists` {81-82} comprise the sets of key values associated with each Server and are passed to each `CSMux` as property `serverAllocation` {13}. Similarly, the list of key values which each `Client` process is to access is defined separately for the `Clients` connected to each `CSMux` process {83, 84}. These `Lists` are passed to a Client process as property `selectList` {Listing 7-3, 33}.

```
77      def server0Map = [1:10, 2:20, 3:30, 4:40, 5:50,
78                        6:60, 7:70, 8:80, 9:90, 10:100]

79      def server1Map = [11:110,12:120,13:130,14:140,15:150,
80                        16:160,17:170,18:180,19:190,20:200]

81      def serverKeyLists = [ [1,2,3,4,5,6,7,8,9,10],
82                             [11,12,13,14,15,16,17,18,19,20] ]

83      def client0List = [1,2,3,14,15,6,7,18,9,10]

84      def client1List = [11,12,13,4,5,16,17,8,19,20]
```

Listing 8-5 Server Maps and Client List Definitions

Listing 8-6 constructs the network of processes required to run the system. The `collect` method is used to construct a list of processes that are returned as new `Client` process instances in the associated closure. Two such `Lists` are required one for each set of `Clients` attached to each of the `CSMux` processes {86-97}. In the definition of a new `Client` note how the individual elements of the channel arrays are accessed and further note that the ends of the channels so referenced are not part of the previously defined channel lists.

```
85     def network = [ ]

86     def server0ClientList = (0 ..< clients).collect { i ->
87                              return new Client ( requestChannel: C0ToM0[i].out(),
88                                                  receiveChannel: M0ToC0[i].in(),
89                                                  clientNumber: i,
90                                                  selectList: client0List)
91                               }

92     def server1ClientList = (0 ..< clients).collect { i ->
93                              return new Client ( requestChannel: C1ToM1[i].out(),
94                                                  receiveChannel: M1ToC1[i].in(),
95                                                  clientNumber: i+10,
96                                                  selectList: client1List)
97                               }

98     network << new CSMux ( inClientChannels: clientsToM0,
99                             outClientChannels: M0ToClients,
100                            fromServers: Mux0FromServers,
101                            toServers: Mux0ToServers,
102                            serverAllocation: serverKeyLists)

103    network << new CSMux ( inClientChannels: clientsToM1,
104                            outClientChannels: M1ToClients,
105                            fromServers: Mux1FromServers,
106                            toServers: Mux1ToServers,
107                            serverAllocation: serverKeyLists)

108    network << new Server ( fromMux: Server0FromMuxes,
109                            toMux: Server0ToMuxes,
110                            dataMap: server0Map)

111    network << new Server ( fromMux: Server1FromMuxes,
112                            toMux: Server1ToMuxes,
113                            dataMap: server1Map)

114    new PAR(network + server0ClientList + server1ClientList).run()
```

Listing 8-6 The Process Definitions to Run a Variable Number of Clients per CSMux With Two Servers

The empty `List network` is then populated with the required instances of the `CSMux` and `Server` processes {98-113}. Finally, a `PAR` is invoked {114} by passing the sum of all the process lists as its parameter, which can then be `run()`. The output from an execution of the network is analysed by ensuring that the Server dataMaps are accessed in the order specified in client0List and client1List. It should be noted that the Client processes attached to CSMux zero are numbered from 0 and those to CSMux one from 10. It can be observed that all the Clients access all the required elements of the servers in the order specified. If the system is executed with one `Client` per `CSMux`; then the version that deadlocked in Chapter 7 can be seen to be operating entirely as expected because the elements accessed from each server are those that deadlocked previously. The major change is that we can now run multiple clients per server, thereby increasing the complexity of the interactions that are being undertaken.

8.5 Summary

In this chapter the concept of the client-server design pattern has been introduced. This is the most important design pattern we shall use and will become fundamental to all the subsequent designs used in the rest of the book.

8.6 Exercises

1. Modify the `Client` process `c7.Client` so that it can ensure that the values returned from the `Server` arrive in the order expected according to their `selectList` property. It should print a suitable message that the test has been undertaken and whether it passed or failed. You are not to use the `GroovyTestCase` mechanism because this would require that the `CSMux` and `Server` processes would have to terminate, which would require a lot of unnecessary programming. A solution is demonstrated in chapter ?? that eliminates the need for tested processes to terminate.