10 Deadlock Revisited: Circular Structures

In previous chapters the concept of a client-server design pattern has been introduced and then it has been applied to a number of simple examples. The primary requirement of the pattern is that any resulting network should not contain any circuits of client and server labels. Needless to say, that if we have a ring of processes then a circuit is inevitable. Hence, we shall investigate a ring of processes to explore how, even though the client-server pattern cannot be applied, we can construct a system that is deadlock free.

The aim of the application is to construct a message passing structure from one node to another by providing a set of message passing elements that connect each node to the next. The simplest way of doing this is to create a ring of message passing nodes to which message sender and receiver processes are attached to each node. Figure 10-1 shows the basic structure with a client-server labelling that demonstrates immediately that deadlock will occur, even ignoring the effect of the Sender and receiver processes. It is obvious that the set of channels that connect the Ring Element processes has to be broken in some way. Deadlock will occur trivially when every Ring Element attempts to either input or output a message at the same time. Thus we have to find a way of breaking the ring

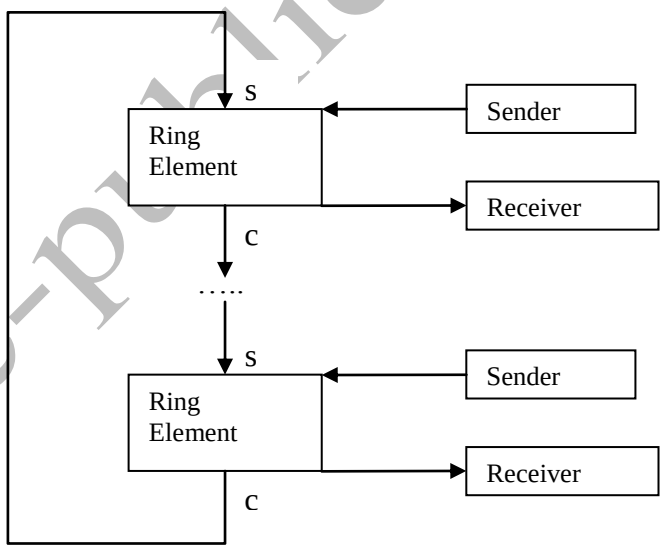

Figure 10-1 The Basic Message passing System

10.1 A First Sensible Attempt

The simplest way of the breaking the ring of channels connecting the Ring Element processes is to add another element to the ring which does the input and output operations in a different order to that undertaken by the Ring Elements. This will mean that there is at least one element on the ring that is

always able to undertake an input operation if all the other Ring Elements are trying to output to the ring. This is shown in Figure 10-2.

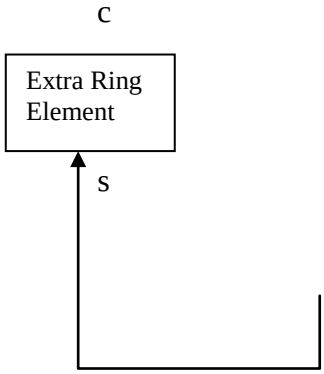

Figure 10-2 Adding the Extra Ring Element

The client-server labelling has not altered and still indicates a problem but we now know that the Extra Ring Element undertakes its input – output operations in a different order to the Ring Elements. The behaviour of a Ring Element is shown in Listing 10-1.

```
01          def RING = 0
02          def LOCAL= 1
03          def ringAlt = new ALT ( [ fromRing, fromLocal ] )
04          while (true) {
05            def index = ringAlt.priSelect()
06            switch (index) {
07              case RING:
08                def packet = (RingPacket) fromRing.read()
09                if ( packet.destination == element ) {
10                  toLocal.write(packet)
11                }
12                else {
13                  toRing.write (packet)
14                }
15                break
16              case LOCAL:
17                def packet = (RingPacket) fromLocal.read()
18                toRing.write (packet)
19                break
```

Listing 10-1 The Ring Element Process Behaviour

A Ring Element alternates over inputs from the ring and from its local sender process {3}. In a loop {4} it determines the enabled alternative, giving priority to inputs from the ring {5}. An enabled input from the ring is `read` {8} as a `RingPacket`, and if the message is for this element, it is written to the local receiver {10}, otherwise it is written to the ring for onward transmission. If the enabled alternative is an input from the local sender then it is read {17} and written to the ring {18}.

The behaviour of the Extra Ring Element is shown in Listing 10-2.

```
20          def packet = new RingPacket (source:-1, destination:-1,
21                                       value:-1, full: false )
22          while (true) {
23            toRing.write( packet )
24            packet = (RingPacket) fromRing.read()
```

Listing 10-2 The Behaviour of the Extra Ring Element

Given that the Ring Elements initially input from the ring, or local Sender process then the Extra Ring Element has to output a packet, so that a Ring Element has a packet to read. A `RingPacket` is defined {20, 21} which is then written to the ring {23}. Thereafter the process simply reads a `RingPacket` from the ring {24} and then outputs it to the ring {23}. The empty packet will continue to circulate forever.

10.1.1 Evaluation

The accompanying web site contains a version of this first attempt `c12.examples.Runv0.groovy` that has print statements inserted within it to show the effect of this solution formulation. The user is able to indicate the number of nodes in the network when the system is executed. The messages received by each receiver process are displayed using a `GEclipseConsole` process. A network with 4 nodes will additionally have the extra node numbered as node 0. The output changes with each execution of the network but never terminates. In a 4 node system each node should receive 3 messages from each of the other nodes, that is, each should receive 9 messages. Typically, no node receives all its messages and some nodes receive no messages. Inspection of the system console print messages indicates that the extra node does indeed output its empty packet and that this read by the next node in the ring. This means that the other nodes have no input on their ring input channels and so they read a message from their local sender process. The sender processes attempt to send their messages as quickly as possibly. This then has the effect of sending many messages on to the ring, which at some stage may deadlock when every node, including the extra node attempt to undertake an input or an output operation. Just when this occurs depends on the particular execution sequence. It is obvious that we have to find a way of managing the number of messages in the ring.

10.2 An Improvement

A simple improvement can be seen quite easily. If a node sends a message to another node on behalf of its local node then the receiving node undertakes to send a message back to the original source that the message has been read. This means that each node can only ever have one packet on the ring at any one time. On the first part of its journey it contains the desired message and then once it has been processed by the destination node it is returned with an empty flag. The definition of the `RingPacket` used to send messages around the system is shown in Listing 10-3. The property `source` {26} gives the number of the node that sent the message and `destination` {27} is the node to which it is to be sent. The actual message is contained in the property `value` {28} and the Boolean `full` {29} indicates whether the packet contains a message or is just an empty packet. A `toString` method (not shown) is provided to enable printing of the packet on the console window and also on the `GEclipseConsole` processes.

```
25    class RingPacket implements Serializable, JCSPCopy {

26      def int source
27      def int destination
28      def int value
29      def boolean full

30      def copy () {
31        def p = new RingPacket( source: this.source, value: this.value,
32                                destination: this.destination, full: this.full)
33        return p
34      }
35    }
```

Listing 10-3 The RingPacket Class definition

A first running, `c12.examples.Runv1.groovy`, of this modification typically results in even worse performance than the initial version. On reflection this is obvious. The Extra Ring Element process still outputs an empty packet onto the ring and thus there will be no space for the messages to rotate around the ring. The solution is to modify the Extra Ring Element process so that it provides an empty space on the ring of nodes so that a communication can take place. This does mean that the Extra Ring Element has to read and then write a packet. This behaviour is shown in Listing 10-4.

```
36          while (true) {
37            def packet = (RingPacket) fromRing.read()
38            toRing.write( packet )
39          }
```

Listing 10-4 The Modified Behaviour of The Extra Ring Element

The execution of `c12.examples.Runv1a.groovy` now results in the proper operation of the network with all Receivers getting and outputting the expected messages. The solution does however have some limitations in that only one packet is ever in circulation for each node as shown in the behaviour given in Listing 10-5.

The solution uses alternative pre-conditions to control the input of message from the ring or from the local sender {40-43}. Initially, messages can be input either from the ring or from the local sender {44, 45}. The `index` of the enabled alternative is determined using a `select` method call {47}.

For messages read from the ring {50}, it is first determined whether the message has its `destination` at this `element` {51}. It is then necessary to determine whether or not the packet is `full` {52}. If the packet is full then we can write a `copy` of the packet to the local receiver process {53}. After which we can update the content of the packet for its return journey to its originating node, because a copy was written to the local receiver process. The `destination` and `source` properties of the packet are updated accordingly {54, 55}, the `packet.full` indication is set `false` {56} and the revised packet written to the ring {57}. If the received packet is not `full` {59} then this is a returned packet and the ring element process can now input a message from its local sender, requiring an update to the associated pre-condition {60}. If the initial packet was not destined for this node element {63} then it is simply written to the ring {64}.

Messages read from the local sender process {68} are immediately written to the ring {69} and the pre-condition controlling input from the local sender is set `false`. As described above, this pre-condition will only be set `true`, when the returned empty packet has been received.

```
40        def RING = 0
41        def LOCAL= 1
42        def ringAlt = new ALT ( [ fromRing, fromLocal ] )
43        def preCon = new boolean[2]
44        preCon[RING] = true
45        preCon[LOCAL] = true

46        while (true) {
47          def index = ringAlt.select(preCon)
48          switch (index) {

49            case RING:
50              def packet = (RingPacket) fromRing.read()
51              if ( packet.destination == element ) {
52                if ( packet.full ) {
53                  toLocal.write(packet.copy())
54                  packet.destination = packet.source
55                  packet.source = element
56                  packet.full = false
57                  toRing.write(packet)
58                }
59                else {
60                  preCon[LOCAL] = true
61                }
62              }
63              else {
64                toRing.write (packet)
65              }
66              break

67            case LOCAL:
68              def packet = (RingPacket) fromLocal.read()
69              toRing.write (packet)
70              preCon[LOCAL] = false
71              break
72          }
73        }
```

Listing 10-5 The Ring Element That Expects A Returned Empty Packet

10.2.1 Evaluation

This solution, though functional, does still have some performance limitations in that an element has to wait for a sent packet to be returned before the next message can be sent. This means that on average half the network is filled with empty packets. The next solution removes this restriction by allowing the reuse of an empty packet if a node is ready to send a message from its local sender process.

10.3 A Final Resolution

The behaviour shown in Listing 10-6 shows the behaviour modification required to use an empty packet, as it passes through a node that is ready to output a local message. The set up of the preconditions and the alternative are the same as the previous version {74-79}. An `emptyPacket` is defined {80, 81} as is a buffer {82} to hold messages from the local sender process. A Boolean flag, `localBufferFull` {83} is used to signify whether or not the `localBuffer` is full. The first thing each `RingElement` node does is to output an `emptyPacket` {84}, which has the effect of initialising the system. In general, this initial empty packet will only pass as far as the next node, which by then will have input a message from its local sender and will thus be able to use this empty packet. The extra element has the revised behaviour given in Listing 10-4. As before, the `index` of the enabled alternative is determined {86) and the appropriate `case` selected {87}.

If the `selected` alternative is to read an input packet from the local sender process {117} this is read into the `localBuffer` {118}, the pre-condition flag for this alternative is set `false` {119} and the `localBufferFull` flag set `true`{120}.

```
74        def RING = 0
75        def LOCAL= 1
76        def ringAlt = new ALT ( [ fromRing, fromLocal ] )
77        def preCon = new boolean[2]
78        preCon[RING] = true
79        preCon[LOCAL] = true

80        def emptyPacket = new RingPacket ( source: -1, destination: -1 ,
81                                           value: -1 , full: false)
82        def localBuffer = new RingPacket()
83        def localBufferFull = false
84        toRing.write ( emptyPacket )

85        while (true) {
86          def index = ringAlt.select(preCon)
87          switch (index) {

88            case RING:
89              def ringBuffer = (RingPacket) fromRing.read()
90              if ( ringBuffer.destination == element ) {
91                toLocal.write(ringBuffer)
92                if ( localBufferFull ) {
93                  toRing.write ( localBuffer )
94                  preCon[LOCAL] = true
95                  localBufferFull = false
96                }
97                else {
98                  toRing.write ( emptyPacket )
99                }
100             }
101             else {
102               if ( ringBuffer.full ) {
103                 toRing.write ( ringBuffer )
104               }
105               else {
106                 if ( localBufferFull ) {
107                   toRing.write ( localBuffer )
108                   preCon[LOCAL] = true
109                   localBufferFull = false
110                 }
111                 else {
112                   toRing.write ( emptyPacket )
113                 }
114               }
115             }
116             break

117           case LOCAL:
118             localBuffer = fromLocal.read()
119             preCon[LOCAL] = false
120             localBufferFull = true
121             break
122         }
123       }
```

Listing 10-6 The Final Ring Element Process

If the `selected` alternative relates to an input from the ring then the message packet is read into a `ringBuffer` {89} and the subsequent processing is determined by the state of that message. If the `destination` of the message is for this node {90} then the message is written to the local receiver process {91}. This means that the node can output the `localBuffer` to the ring if it is full and update the flags associated with the buffer {92-95}; otherwise an `emptyPacket` is written to the ring {98}. If the `ringBuffer` does not have this node as its `destination` {101} then if the `ringBuffer` is full it is simply written to the ring {103}, otherwise the `localBuffer` is processed in the way described previously {106-112}.

10.3.1 Evaluation

This final version has resulted in a solution that routes messages around a circular network, which is inherently prone to deadlock. This version does not suffer from the drawbacks of the previous solution in that an empty packet only travels around the network until it comes to a node that needs to send a message from its `localBuffer` to another node. An argument has been presented that explains why

deadlock will not occur because client-server label does not provide a categorical solution and furthermore indicates that deadlock will occur.

10.4 Summary

This chapter has analysed a set of processes that inherently tend to deadlock. Two algorithms have been developed that overcome the problems. The benefit of one solution over the other has been explained, though this is difficult to measure unless the system is run over a real network, where each Ring Element process can be placed on a specific processor of that network.