20 Redirecting Channels: A Self Monitoring Process Ring

In Chapter 18 it was shown how Mobile Agents can be constructed using serializable `CSProcesses`. In Chapter 10 a solution was developed to the problem of a ring of processes that circulated messages around themselves. No consideration was given to the problems that might happen if messages were not taken from the ring immediately. In this chapter we explore a solution to the problem that utilises two mobile agents that dynamically manage the ring connections in response to a node of the ring detecting that incoming messages are not being processed sufficiently quickly.

The solution as presented does not require any form of central control to initiate the corrective action. The agents are invoked by the node when it is determined that the processing of incoming messages has stopped. The solution essentially builds an Active Network [,] at the application layer, rather than the usual normal network layer. The solution also utilises the `Queue` and `Prompter` processes developed in Chapter 5. These provide a means of providing a finite buffer between the ring node process and the process receiving the messages. Additionally, a console interface has been added to the message receiver process so that users can manipulate its behaviour and more easily observer the effect that the agents have on the overall system operation.

20.1 Architectural Overview

Figure 20-1 shows the process structure of one node and also its relationship to its adjoining nodes. It is presumed that there are other nodes on the ring all with the same structure. It shows the state of the system once it has been detected that `RingElement`-n has stopped receiving messages. The net channel connections joining `RingElement` –n to the ring have been removed and replaced by the connection that goes between `RingElement` n-1 and `RingElement` n+1.

The figure also shows the additional processes used to provide the required management. The `RingElement` outputs messages into the `Queue` process, instead of directly into the `Receiver` process. The `Prompt` process requests messages from the `Queue` which it passes on to the `Receiver` process. Whenever the `Queue` process is accessed, either for putting a new message or getting a message in response to a `Prompt` request, the number of messages in the `Queue` is output to the `StateManager` process. The `StateManager` process is able to determine how full the `Queue` is and depending on pre-defined limits will inform the `RingElement` that the `Receiver` has stopped inputting messages or has resumed. This will be the trigger to send either the `StopAgent` or the `RestartAgent` around the network.

The `Queue` contains sufficient message slots to hold two messages from each node. The signal to indicate that the receiver has stopped inputting messages is generated by the `StateManager` when the `Queue` is half full. A naive solution would just create an infinite queue to deal with the problem and not worry about the fact that the messages were no longer being processed by the `Receiver`. However, this is not sensible because were the situation to be sustained over a long period the processor would run out of memory and would fail in a disastrous manner. It is thus much better to deal with the situation rather than

ignore it. The `Sender` process has been modified in as much that a delay has been introduced so that there is a pause between the sending of a message and the next. This was done so that the operation of the revised system could be more easily observed.

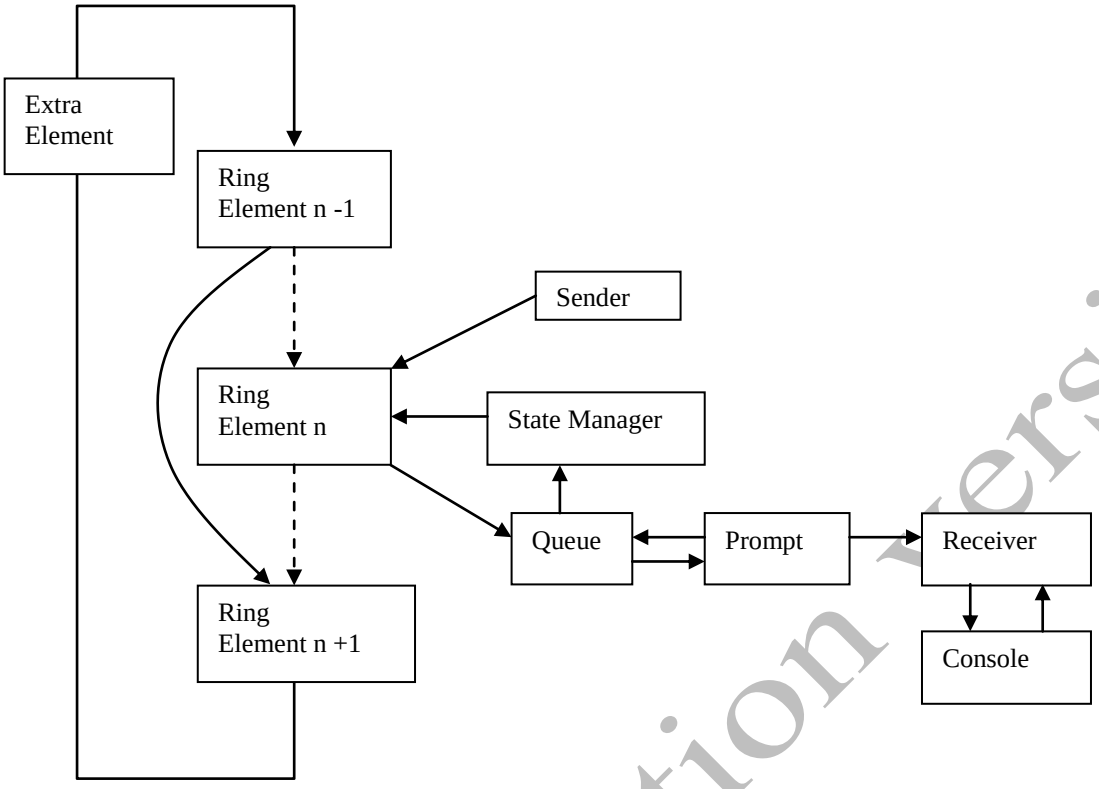

Figure 20-1 The Architecture of the Self Monitoring Ring Structure

20.2 The Receiver process

Listing 32-1 shows the modified `Receiver` process, the behaviour of which is modified by the `Console` process. The channel `fromElement` {2} is the input channel from the `Prompt` process. The remaining channel properties connect the `Receiver` process to the `Console` process. The channel `outChannel` {3} is used to display messages on the `Console` and the channel `fromChannel` {5} is used to input messages from the `Console` The `clear` channel {4} is used to reset the input area of the `Console`.

The process can receive inputs on either the `fromConsole` or `fromElement` channel and an `ALT` is constructed to this effect {8}. The main loop {11-28} waits for the enabling of an alternative {12,13} and then deals with that input. The expected input from the Console {15-24} is either any string to stop the `Receiver` process, typically "`stop`", followed by "`go`" to restart the `Receiver` process. If the input is from the ring element then a message is sent to the `Console` writing the content of the data received in the `Console`'s output area. The `Console` process is implemented using the `GEclipseConsole` process.

```
01     class Receiver implements CSProcess {

02       def ChannelInput fromElement
03       def ChannelOutput outChannel
04       def ChannelOutput clear
05       def ChannelInput fromConsole
06
07       def void run() {
08         def recAlt = new ALT ([ fromConsole, fromElement])
09         def CONSOLE = 0
10         def ELEMENT = 1

11         while (true) {
12           def index = recAlt.priSelect()
13           switch (index) {

14             case CONSOLE:
15               def state = fromConsole.read()
16               outChannel.write("\n go to restart")
17               clear.write("\n")
18               while (state != "\ngo") {
19                 state = fromConsole.read()
20                 outChannel.write("\n go to restart")
21                 clear.write("\n")
22               }
23               outChannel.write("\nresuming ...\n")
24               break

25             case ELEMENT:
26               def packet = fromElement.read()
27               outChannel.write ("Received: " + packet.toString() + "\n")
28               break
29           }
30         }
31       }
32     }
```

Listing 20-1The Receiver Process

20.3 The Prompter Process

The `Prompter` process, shown in Listing 20-2 has channels that communicate with the `Queue` process as follows. The channel `toQueue` {34} is used by the `Prompter` to signal {39} that it ready to read an item from the `Queue`. The channel `fromQueue` {35} is used to input an item from the `Queue`, which is immediately written to the `Receiver` process {40} using the `toReceiver` channel {36}.

```
33     class Prompter implements CSProcess{

34       def ChannelOutput toQueue
35       def ChannelInput fromQueue
36       def ChannelOutput toReceiver

37       void run() {
38         while (true) {
39           toQueue.write(1)
40           toReceiver.write ( fromQueue.read() )
41         }
42       }
43     }
```

Listing 20-2 The Prompter Process

20.4 The Queue Process

The `Queue` process shown in Listing 20-3, is very similar to that described in Chapter 6 except that an additional output channel `toStateManager` has been added {47} to the properties. The property `slots` {49} is used to specify the number of items that can be held in the queue. The channels `fromPrompter` and `toPrompter` {46, 48} are the respective channel ends of the Queue's `toQueue` and `fromQueue` channels {34, 35}. The channel `fromElement` {45} is used to receive inputs from the ring element process.

```
44     class Queue implements CSProcess {

45       def ChannelInput fromElement
46       def ChannelInput fromPrompter
47       def ChannelOutput toStateManager
48       def ChannelOutput toPrompter
49       def int slots

50       void run() {
51         def qAlt = new ALT ( [ fromElement, fromPrompter ] )
52         def preCon = new boolean[2]
53         def ELEMENT = 0
54         def PROMPT = 1
55         preCon[ELEMENT] = true
56         preCon[PROMPT] = false
57         def data = []
58         def counter = 0
59         def front = 0
60         def rear = 0

61         while (true) {
62           def index = qAlt.priSelect(preCon)
63           switch (index) {

64             case ELEMENT:
65               data[front] = fromElement.read()
66               front = (front + 1) % slots
67               counter = counter + 1
68               toStateManager.write(counter)
69               Break

70             case PROMPT:
71               fromPrompter.read()
72               toPrompter.write( data[rear])
73               rear = (rear + 1) % slots
74               counter = counter - 1
75               toStateManager.write(counter)
76               break
77           }
78           preCon[ELEMENT] = (counter < slots)
79           preCon[PROMPT] = (counter > 0 )
80         }
81       }
82     }
```

Listing 20-3 The Queue Process

Each time an item is either put into or removed from the queue, depending on the enabled case, the value of the `counter`, which holds the number of items stored in the Queue is output to the `StateManager` process using the channel `toStateManager` {68, 75}.

20.5 The State Manager Process

The role of the `StateManager` process, shown in Listing 20-4, is to determine how full the `Queue` has become. It inputs the number of items in the `Queue` as the variable `usedSlots` {91} from `Queue` using the channel `fromQueue` {84}. It does this every time the `Queue` process either adds or removes an item. The variable `limit` {88} is half the maximum number of items that can be held in the `Queue`, which is available by means of the property `queueSlots` {86}. The Boolean variable `aboveLimit` determines whether the number of items in the `Queue` is greater than or equal to the `limit` value {92}. The variable `state` {89} indicates the current state of the relationship between `usedSlots` and `limit` and is initially "`NORMAL`".

There are two cases of interest; first, when the number of items in the `Queue` is increasing and goes above the `limit` {93-96}, which means the `Queue` is now half full because the `Receiver` has stopped processing items. In this case we need to inform the ring element process of this situation by outputting the "`ABOVE_LIMIT`" `state` on the `toElement` channel. The other case {97-99} is when the `Queue` has been above the limit and is now emptying, because the `Receiver` has resumed processing. In this case we reset `state` to "`NORMAL`" and write it to the `toElement` channel.

```
83     class StateManager implements CSProcess{

84       def ChannelInput fromQueue
85       def ChannelOutput toElement
86       def int queueSlots

87       void run() {
88         def limit = queueSlots / 2
89         def state = "NORMAL"

90         while (true) {
91           def usedSlots = fromQueue.read()
92           def aboveLimit = ( usedSlots >= limit)

93           if ((state == "NORMAL") && ( aboveLimit)) {
94             state = "ABOVE_LIMIT"
95             toElement.write("STOP")
96           }

97           if ((state == "ABOVE_LIMIT") && ( !aboveLimit)) {
98             state = "NORMAL"
99             toElement.write("RESTART")
100          }
101        }
102      }
103    }
```

Listing 20-4 The State Manager Process

20.6 The Stop Agent

The StopAgent, shown in Listing 20-5, is constructed within the RingAgentElement process Listing 20-7. It is activated whenever a RingAgentElement process receives an "ABOVE_LIMIT" message from its StateManager process.

The channel properties toLocal {105} and fromLocal {106} will be used to connect the StopAgent to a host process when it arrives at a new RingAgentElement process. As such these properties are not initialised in its constructor. The property homeNode {107} is used to hold the node identity of the RingAgentElement that has detected the fault The property previousNode {108} is used to hold the node identity of the RingAgentElement that precedes the node that has detected the fault condition. This is the node that will be required to redirect its output channel to the one following the faulty node. The Boolean property initialised {109} is used to indicate whether the property nextNodeInputEnd {110} has been set. The nextNodeInputEnd holds the net channel input end of the channel to which the previous node's net channel output has to be redirected. This can only be obtained once the StopAgent has moved from the faulty node to the next node.

The methods connect {111-115} and disconnect {116-119} are used to initialise and then remove the channel connections between the StopAgent and the host process. The run method {120-132} initially outputs the properties homeNode, previousNode and initialised to the host process using the channel toLocal. If the StopAgent has not been initialised it reads the nextNodeInputEnd from the host process using the channel fromLocal. The host process then sends the StopAgent a Boolean gotThere indicating whether or not the StopAgent has arrived at the node identified by previousNode. If the StopAgent has arrived then the value of nextNodeInputEnd is written to the host process.

An implication of this design is that a network must consist of at least two RingAgentElements and an AgentExtraElement process otherwise there would be no possibility of undertaking the redirection.

```
104   class StopAgent implements MobileAgent {

105     def ChannelOutput toLocal
106     def ChannelInput fromLocal
107     def int homeNode
108     def int previousNode
109     def boolean initialised
110     def NetChannelLocation nextNodeInputEnd

111     def connect (List c) {
112       this.toLocal = c[0]
113       this.fromLocal = c[1]
114
115     }

116     def disconnect () {
117       this.toLocal = null
118       this.fromLocal = null
119     }

120     void run() {
121       toLocal.write(homeNode)
122       toLocal.write(previousNode)
123       toLocal.write(initialised)

124       if ( ! initialised) {
125         nextNodeInputEnd = fromLocal.read()
126         initialised = true
127       }

128       def gotThere = fromLocal.read()
129       if ( gotThere ) {
130         toLocal.write(nextNodeInputEnd)
131       }
132     }
133   }
```

Listing 20-5 The Stop Agent

20.7 The Restart Agent

The `RestartAgent` shown in Listing 20-6 undoes the effect of the `StopAgent` once the faulty node detects that the fault has been removed. Its properties are very similar and are used in the same way as those with similar names in the `StopAgent`. The `firstHop` {140} property is used to indicate whether the `RestartAgent` has arrived at the node that succeeds the node that was faulty. This has to be dealt with especially if deadlock is not to occur in the system as will be explained in 21.8.2.2 and 21.8.2.3.

The `run` method of the `RestartAgent` {149-153} simply writes `firstHop` to the host process and the sets `firstHop false` if it is `true`. The values of `homeNode` and `previousNode` are then written to the host process, which processes them accordingly.

```
134   class RestartAgent implements MobileAgent {

135     def ChannelOutput toLocal
136     def ChannelInput fromLocal
137     def int homeNode
138     def int previousNode
139     def boolean firstHop

140     def connect (List c) {
141       this.toLocal = c[0]
142       this.fromLocal = c[1]
143
144     }

145     def disconnect () {
146       this.toLocal = null
147       this.fromLocal = null
148     }

149     void run() {
150       toLocal.write(firstHop)
151       if (firstHop) { firstHop = false }
152       toLocal.write(homeNode)
153       toLocal.write(previousNode)
154     }
155   }
```

Listing 20-6 The Restart Agent

20.8 The Ring Agent Element Process

The `RingAgentElement` process is somewhat lengthy and thus its description will be subdivided into the sections it comprises. The structure is very similar to that described in Chapter 12 with the modifications required to deal with the agent processing. It should be recalled that the deadlock free architecture developed in Chapter 12 was optimised so that any node that had data to put onto the ring could do so provided an empty packet had been received from the ring. Thus the number of packets circulating around the ring was the same as the number of nodes, excluding the extra element used to overcome one aspect of the deadlock profile of the network.

20.8.1 Properties and Initialisation

The properties and initialisation of the variables used in the operation of the `AgentRingElement` process is shown in Listing 20-7. The channels `fromRing` {157} and `toRing` {158} are net channels used to connect this node to the preceding and following nodes respectively. Messages are received from the `Sender` process on the channel `fromSender` {159}. Similarly outputs of the state of the `Queue` are input from `StateManager` on the `fromStateManager` channel {160}. In this revised version messages received for this node are now output to the `Queue` process on the `toQueue` {161} channel rather than directly to the `Receiver` process. The `element` {162} property holds the identity of this node as an integer. The nodes are numbered from 1 upwards, in sequence, with the `AgentExtraElement` given the identity 0.

The `run` method {163}, initially defines the channels and channel ends used to connect this process to any agent. The mechanism is exactly the same as that described previously in Chapter 19. The two agents are then constructed {170-175} and the properties that need to be defined are initialised. It should be noted that the preceding node to `AgentRingElement` with identity 1 will have its previous node as having identity 0, which is the `AgentExtraElement`. Thus the `AgentExtraElement` has also had to be modified to deal with the arrival of agents in a manner very similar to that to be presented. The precise changes for the `AgentExtraElement` will not be described but can be determined from the accompanying web site.

```
156   class RingAgentElement implements CSProcess {

157     def ChannelInput fromRing
158     def ChannelOutput toRing
159     def ChannelInput fromSender
160     def ChannelInput fromStateManager
161     def ChannelOutput toQueue
162     def int element

163     void run() {

164       def One2OneChannel N2A = Channel.createOne2One()
165       def One2OneChannel A2N = Channel.createOne2One()
166       def ChannelInput toAgentInEnd = N2A.in()
167       def ChannelInput fromAgentInEnd = A2N.in()
168       def ChannelOutput toAgentOutEnd = N2A.out()
169       def ChannelOutput fromAgentOutEnd = A2N.out()

170       def stopper = new StopAgent ( homeNode: element,
171                                     previousNode: element - 1,
172                                     initialised: false)

173       def restarter = new RestartAgent ( homeNode: element,
174                                          previousNode: element - 1,
175                                          firstHop: true)

176       def NetChannelLocation originalToRing = toRing.getChannelLocation()

177       def failedList = [ ]

178       def RING = 0
179       def SENDER= 1
180       def MANAGER = 2
181       def ringAlt = new ALT ( [ fromRing, fromSender, fromStateManager ] )
182       def preCon = new boolean[3]
183       preCon[RING] = true
184       preCon[SENDER] = true
185       preCon[MANAGER] = true
186       def emptyPacket = new RingPacket ( source: -1, destination: -1 ,
187                                          value: -1 , full: false)
188       def localBuffer = new RingPacket()
189       def localBufferFull = false

190       def restartBuffer = null
191       def restarting = false
192       def stopping = false

193       toRing.write ( emptyPacket )

194       while (true) {
195         def index = ringAlt.select(preCon)
196         switch (index) {
```

Listing 20-7 The Properties and Initialisation of the Ring Element Process

The variable `originalToRing` {176} is initialised to the `NetChannelLocation` of the `toRing` channel when the process is initialised. If this channel were to be redirected then it is easier to have a record of the original value pre-stored in the process when it comes to reinstating the original connection. The list variable `failedList` {177} will be modified by interaction with the agent to indicate the node(s) that have become faulty. This means that messages destined for a faulty node can be stopped from being sent.

The alternative `ringAlt` {181} has been extended to include the channel `fromStateManager`, so that inputs from the `StateManager` are considered. The related `preCon` element `preCon[MANAGER]` is always `true` {185} as such inputs must always be processed. The variable `restartBuffer` is used when a `RestartAgent` is received {190} and the Boolean variables `restarting` and `stopping` {191, 192} are also used to manage agent processing and their use will be described later.

20.8.2 Dealing With Inputs From The Ring

The main loop of the process essentially deals with incoming packets from the ring, the `Sender` process and the `StateManager` process as indicated by the first part of the loop which selects the enabled

alternative from these inputs. The incoming packets from the ring are of three types; data packets, StopAgents and RestartAgents. Each of these cases will be dealt with separately.

20.8.2.1 Ring Packet Processing

Listing 20-8 shows how RingPackets are dealt with.

```
197             case RING:
198               def ringBuffer = fromRing.read()
199               if ( ringBuffer instanceof RingPacket) {
200                 if ( ringBuffer.destination == element ) {
201                   toQueue.write(ringBuffer)
202                   if (stopping) {
203                     stopping = false
204                     toRing.write(stopper)
205                   }
206                   else {
207                     if (restarting) {
208                       restarting = false
209                       toRing.write(restartBuffer)
210                     }
211                     else {
212                       if ( localBufferFull ) {
213                         toRing.write ( localBuffer )
214                         preCon[SENDER] = true
215                         localBufferFull = false
216                       }
217                       else {
218                         toRing.write ( emptyPacket )
219                       }
220                     }
221                   }
222                 }
223                 else {
224                   if ( ringBuffer.full ) {
225                     toRing.write ( ringBuffer )
226                   }
227                   else {
228                     if (stopping) {
229                       stopping = false
230                       toRing.write(stopper)
231                     }
232                     else {
233                       if (restarting) {
234                         restarting = false
235                         toRing.write(restartBuffer)
236                       }
237                       else {
238                         if ( localBufferFull ) {
239                           toRing.write ( localBuffer )
240                           preCon[SENDER] = true
241                           localBufferFull = false
242                         }
243                         else {
244                           toRing.write ( emptyPacket )
245                         }
246                       }
247                     }
248                   }
249                 }
250               }
```

Listing 20-8 RingPacket Processing

If ringBuffer {200} contains a message destined for this node then it is written to the Queue process using the toQueue channel {201}. The subsequent action taken depends on the state of various Boolean variables. If stopping is true indicating that the StateManager has detected a fault then the StopAgent stopper is written to the ring {204}. This action is given the highest priority. If restarting is true indicating that a RestartAgent has been received then the instance saved in restartBuffer is written to the ring {209}. If a message has been received from the Sender process then it is written to the ring and the preCon and variable values are modified so that another message can be received from Sender {213}. If none of the above conditions are true then an empty packet is written to the ring {218}.

If ringBuffer {200} contains a message that is intended for another node then the message is simply written to the ring {224}. The only other possible case is that a message has been received that is the emptyPacket {227} and thus the processing required is governed, as before, by the state of the variables associated with, stopping, restarting and the state of the buffer that holds messages from the Sender process. If none of these requires any action then an emptyPacket is written to the ring.

20.8.2.2 Stop Agent Processing

The coding associated with StopAgent processing is shown in Listing 20-9. The previously read ringBuffer {198} is placed in the variable theAgent {252}. The agent is then connected to this process and executed {253-255} in the same manner as described in Chapter 19. The interaction with the agent can now commence with the reading of the failed node identity {256}, which can be appended to the failedList {257}. The targetNode for the agent can then be read {258} and then the indication of whether the agent has been initialised or not into alreadyInitialised {259}. If the agent has not been initialised then the channel location of the input fromRing channel can be written to the agent {261}.

```
251               if (ringBuffer instanceof StopAgent) {
252                 def theAgent = ringBuffer
253                 theAgent.connect ( [fromAgentOutEnd, toAgentInEnd] )
254                 def agentManager = new ProcessManager (theAgent)
255                 agentManager.start()
256                 def failedNode = fromAgentInEnd.read()
257                 failedList << failedNode
258                 def targetNode = fromAgentInEnd.read()
259                 def alreadyInitialised = fromAgentInEnd.read()

260                 if ( ! alreadyInitialised ) {
261                   toAgentOutEnd.write (fromRing.getChannelLocation())
262                 }

263                 if (element == targetNode) {
264                   toAgentOutEnd.write(true)
265                   def NetChannelLocation revisedToRing = fromAgentInEnd.read()
266                   toRing = NetChannelEnd.createAny2Net(revisedToRing)
267                   agentManager.join()
268                   theAgent.disconnect()
269                 }

270                 else {
271                   toAgentOutEnd.write(false)
272                   agentManager.join()
273                   theAgent.disconnect()
274                   toRing.write(theAgent)
275                 }
276               }
```

Listing 20-9 Stop Agent Processing

The remainder of the processing deals with whether or not the agent has arrived at the required destination node, which is the node preceding the faulty node. By the time the agent has travelled to the destination node all the intervening nodes will have had their failedList updated so they will no longer be sending messages to the failed node.

If the agent has arrived at the destination node {263} then true is written to the agent {264} and the channel location to be used for subsequent outputs by this destination node is read from the agent as revisedToRing {265}. The channel toRing is then assigned the Any2Net channel created from revisedToRing {266}. An Any2Net channel has been used so that more than one process can write to the channel, which is required in this situation. When the faulty node detects that it can resume processing because the Receiver process has started to accept messages again; a RestartAgent will be written on the original toRing channel.

The agentManager then joins with the agent {267} waiting for the latter to terminate at which point theAgent can be disconnected {268}. There is no need to write theAgent to the next node as it was the faulty node and we now know that its input, fromRing, has been bypassed. More importantly we do

not write an emptyPacket to the ring even though the agent has taken up an emptyPacket, when it was first written to the ring. If we were to write an emptyPacket to the ring there would be more packets than nodes and deadlock would ensue.

If theAgent has not yet arrived at its destination then processing is much simpler. A false signal is written to the agent {271}. The process then waits for the agent to terminate {272} after which it can disconnect itself from the agent {273}. The process can then write theAgent to the ring {274}.

20.8.2.3 Restart Agent Processing

RestartAgent processing is shown in Listing 20-10, which starts with assigning {277} theAgent from ringBuffer into which it was originally read {198}. In the same manner as before, theAgent can be connected to the host process, allocated to an agentManager and started {278-280}.

Three values are then read from the agent, firstHop {281}, resumedNode {282} and targetNode {284} using the channel fromAgent. As the agent passes through each element it needs to modify the list of failed nodes by removing the identifier of the resumedNode from the failedList {283}.

If this is the first move made by the agent the firstHop will be true {285}. In this case the resuming node will have injected the RestartAgent into the ring of nodes as an extra packet (see the next section 23.8.3) and this needs to be dealt with specially, if deadlock is not to occur. All the processing between agent and host element has been completed so all that remains is to wait for the agent to terminate and to disconnect it from the host {286, 287}. Instead of writing theAgent onto the ring it is placed in a buffer, restartBuffer {288} and restarting is set true {289}. In due course when either an empty packet arrives at the node or a packet is received that is destined for this node the restartBuffer will be written to the ring {207-209, 233-235} and restarting is reset to false.

The remainder of the processing deals with determination of whether or not the agent has arrived at the node, targetNode, which needs to have its toRing output channel returned to its original setting so that the resumed node is reconnected to the network. This is dealt with by resetting toRing to the originalToRing value previously saved {176}. The host then waits for the agent to terminate and disconnects itself from the agent. {294-5}. Finally, an emptyPacket is written to the ring {296} because the previously faulty node is now operating again. If the agent has not arrived at the targetNode it simply waits for theAgent to terminate, disconnects itself and then writes theAgent to the next node {299-301}.

```
277                 def theAgent = ringBuffer
278                 theAgent.connect ( [fromAgentOutEnd, toAgentInEnd] )
279                 def agentManager = new ProcessManager (theAgent)
280                 agentManager.start()
281                 def firstHop = fromAgentInEnd.read()
282                 def resumedNode = fromAgentInEnd.read()
283                 failedList = failedList - [resumedNode]
284                 def targetNode = fromAgentInEnd.read()

285                 if (firstHop) {
286                   agentManager.join()
287                   theAgent.disconnect()
288                   restartBuffer = theAgent
289                   restarting = true
290                 }

291                 else {
292                   if (element == targetNode) {
293                     toRing = NetChannelEnd.createAny2Net (originalToRing)
294                     agentManager.join()
295                     theAgent.disconnect()
296                     toRing.write ( emptyPacket )
297                   }

298                   else {
299                     agentManager.join()
300                     theAgent.disconnect()
301                     toRing.write(theAgent)
302                   }
303                 }
304               }
305             }
```

Listing 20-10 Restart Agent Processing

20.8.3 Dealing With Inputs From the StateManager Process

The `StateManager` only generates inputs to the `RingAgentElement`, whenever the state of the `Queue` is such that either a ring element needs to be removed from the ring or reinstated. The associated processing is shown in Listing 20-11. If the `fromStateManager` alternative {195} is enabled then the `MANAGER` case is processed {306} of the `switch` specified at {185}. The `state` is read `fromStateManager` {307}.

If "`STOP`" is read then `stopping` is set `true`, which will cause the `StopAgent`, `stopper` {170}, to be output when the next `emptyPacket` is read by the node {202-204} or a packet arrives {228-230} that is destined for this node. At this point the node is still operating normally with respect to incoming packets. Its operation will only be modified when it no longer receives packets on its `fromRing` channel because it has been redirected. In fact the faulty node is not modified in any way as its `fromRing` channel is still connected to the previous node. Its `toRing` channel is still connected and will, in due course, be used to output the `RestartAgent`, when it resumes normal operation.

If the value "`RESTART`" has been read then all that is required is to write {312} the `RestartAgent`, `restarter` {173} to the ring. This is an additional packet being placed on the ring without there having been a packet received because the node in this case has been disconnected from the ring. This justifies the `firstHop` processing previously described {285-290} otherwise deadlock would occur.

```
306           case MANAGER:
307              def state = fromStateManager.read()
308              if (state == "STOP") {
309                stopping = true
310              }
311              else {
312                toRing.write ( restarter )
313              }
314              break
```

Listing 20-11 StateManager Input Processing

20.8.4 Dealing With Inputs From the Sender Process

Listing 20-12 shows the processing concerned with messages received from the `Sender` process. The `Sender` process writes messages to the ring element node at regular intervals and sends messages to all the other nodes. Messages can only be received when the `localBuffer` is not full and its element in the `preCon` array is true. Once a message packet is received `fromSender` {316}. A test is undertaken to determine whether or not its `destination` is in the `failedList` {317}. If the message is destined for a failed node it is effectively ignored and the control variable `localBufferFull` is not changed. This means that the message will not get written to the ring and will be overwritten by another message. This has the effect that while a node has failed it will receive no messages and for the time that it has failed any messages that it should have received will be lost. This of course is not the most sensible course of action but it does make it much easier to observe the dynamic operation of the system.

```
315             case SENDER:
316               localBuffer = fromSender.read()
317               if ( ! failedList.contains(localBuffer.destination) ) {
318                 preCon[SENDER] = false
319                 localBufferFull = true
320               }
321               break
```

Listing 20-12 Sender Process Input Processing

20.9 Running A Node

The easiest way of instantiating the system is to create a script for each node and the extra element that create the node directly. Such a script for node 2 is shown in Listing 20-13.

```
322    Node.getInstance().init(new TCPIPNodeFactory())

323    def int nodeId = 2
324    def int sentMessages = 150
325    def int nodes = 4
326    def fromRingName = "ring2"
327    def toRingName = "ring3"

328    def fromRing = CNS.createNet2One(fromRingName)
329    def toRing = CNS.createAny2Net(toRingName)

330    def processNode = new AgentElement ( fromRing: fromRing,
331    toRing: toRing,
332    element: nodeId,
333    iterations: sentMessages,
334    nodes: nodes)

335    new PAR ([ processNode]).run()
```

Listing 20-13 Running Node 2

The properties required by the `AgentElement` process, which creates the network of `RingAgentElement`, `Sender`, `Queue`, `Prompter`, `Receiver` and `StateManager` processes for each node, are defined {321-327}. The ring channels are defined noting that the `toRing` channel {329} is defined as `Any2Net` because when a channel is redirected as seen in Figure 20-1 the receiving node has two channel connections on its `fromRing` input.

20.10 Observing The System's Operation

The accompanying web site contains all the processes and scripts and can be used to create a network of four nodes. Once all the nodes are operating, the console associated with each `Receiver` process will be visible. A `Receiver` process can be made to stop receiving messages by typing "stop" into the input area of the console. It will be observed that the other nodes will continue operation but will not send any messages to the failed node. The failed node can be restarted by typing "go" into the console input area. At which point it will be observed that the messages that have been saved in the `Queue` appear immediately and at some time later messages from the other nodes start to appear. However it can also be

observed that the sequence of message values has a gap that corresponds to the time for which the node was not receiving messages.

20.11 Summary

This chapter has demonstrated an agent based system that has neither a central control system nor a specific host to which agents return messages or data. The control is distributed around the system so that agents commence when a local situation develops that requires their intervention. The agents have a limited life span pertaining to the time they are required. They are then destroyed. If the node becomes faulty again the original agent is reused.

20.12 Challenges

1. Does the processing deal with the case when two adjacent nodes fail? If not, what changes are required?

1. Modify the processing so that messages for faulty nodes are not lost but retained for later transmission once they resume normal processing.

2. Once a node fails because the Receiver has been stopped it no longer receives empty packets and therefore cannot send messages onto the ring, even though the Sender process is still functional. Modify the behaviour so that a failed node can still send messages. The real problem is that great care has to be taken to ensure that the system does not deadlock.