18 Mobile Agents: Going for a Trip

A mobile agent is a means by which an autonomous unit of processing can be made to visit a number of processing nodes to undertake some operation on data held at that node to be returned to some initiating node. On arrival at a node an agent will connect itself to the host node, thereby enabling it to access the host's resources. Once the interaction is complete, the agent will disconnect itself from the host's resources before moving to the next host node according to some agent transfer regime. During the course of its travels, an agent is required to collect some data from the host nodes, which it either communicates immediately or can be accessed when the agent returns to its originating node. An agent can also modify the nodes that it visits depending on the outcome of an interaction at a particular node.

18.1 Mobile Agent Interface

The MobileAgent interface is shown in Listing 18-1. It extends `CSProcess` because we want the agent to be able to run as a process on arrival at a node. It has to extend `Serializable` because the agent is to be communicated over a network. Two methods are required; `connect` {3}, which is passed a `List` of channels and other properties by which the agent is able to communicate with its host and `disconnect` {4} which is called prior to the agent moving to another node, which sets to `null` all the channel connections that were created by the `connect` method.

```
01     interface MobileAgent extends CSProcess, Serializable {
02
03        abstract connect(List x)
04        abstract disconnect()
05     }
```

Listing 18-1 The Mobile Agent Interface

18.2 A First Parallel Agent System

The first agent system will simply send an agent round a ring of host nodes, passing a `List` into the host to which the host appends another value and returns the `List` to the agent before the agent moves to the next node. On its return to the root node the agent transfers the revised list to the root node before travelling around the ring again.

18.2.1 The Agent

Listing 18-2 shows the definition of the `Agent` that will travel around the ring of host nodes. The process will interface to the host node by means of the channels `toLocal` {7} and `fromLocal` {8} and will collect data in the `List results` {9}. The `connect` method has a `List` parameter, `c` {10}, that contains two channels that are the `toLocal` and `fromLocal` channel ends respectively {11, 12}. The `disconnect` method {14-17} simply sets the local channels to `null`.

```
06    class Agent implements MobileAgent {

07      def ChannelOutput toLocal
08      def ChannelInput fromLocal
09      def results = [ ]

10      def connect ( List c ) {
11        this.toLocal = c[0]
12        this.fromLocal = c[1]
13      }

14      def disconnect () {
15        toLocal = null
16        fromLocal = null
17      }

18      void run() {
19        toLocal.write (results)
20        results = fromLocal.read()
21      }
22    }
```

Listing 18-2 The Agent Process

The `Agent`'s `run` method, which is required because the `MobileAgent` interface implements the interface `CSProcess` {18-20}, simply `writes` the value of `results` to the `toLocal` channel {19} and then reads the `results` back from the `fromLocal` channel {20}. At which point the `Agent` process will terminate.

18.2.2 The Root Process

The `Root` process initially sends the `Agent` into the ring of processes and then receives the returning agent after it has travelled around the ring to extract the `results` before sending the `Agent` around the ring again. The structure of the process is shown in Listing 18-3.

The channels `inChannel` {24} and `outChannel` {25} connect the `Root` process to the ring of processes. The property `iterations` {26} indicates how many times the `Agent` will be sent round the ring of processes. The property `initialValue` {27} is a `String` that will be placed in the `results List` as the first element of that list.

The `One2OneChannels N2A` {29} and `A2N` {30} provide the local connections between `theAgent` and this node. They cannot be accessed externally from this node and hence are defined within the `run` {28} method. The input and output ends of these local channels are obtained {31-34}. A variable, `theAgent`, is defined {35} of type `Agent` that has only its `results` property initialised. Even though `theAgent` has been defined it is not connected to this, the `Root` node, until it has been round the ring of host processes at least once. More particularly, the local connections between `theAgent` and host cannot be made until `theAgent` has been transferred to a new host.

A `for` loop is used to send `theAgent` around the ring of processes the required number of times {36}. Initially, `theAgent` is `written` to the `outChannel` {37} and then the `Root` process waits until `theAgent` can be read from its `inChannel` {38}, which will only happen once `theAgent` has passed through all the host nodes on the ring.

The `Root` node can now `connect` to `theAgent` with the appropriate ends of the local connection channels {39}. An `agentManager` of type `ProcessMananger` is defined {40} that is used to manage the operation of the interaction of `theAgent` in parallel with the `Root` node. The agentManager is then started {41}. It first reads the `returnedResults` that are written by `theAgent` {42} using the `fromAgentInEnd` input channel. The value of `returnedResults` is printed on the console window {43} and then written back to `theAgent`, modified to indicate the end of an iteration {44}, using the `toAgentOutEnd` output channel {45}. The interaction between `theAgent` and the `Root` node is now complete, with the former having terminated and the latter still running. The `agentManager joins` the `Root` process {46}, which has the effect of recovering the resources used by the `agentManager` when the process it is managing terminates.

The `Root` process can now `disconnect` `theAgent` from itself {47}. The Root process will now progress to execute any outstanding `iterations`.

```
23    class Root implements CSProcess{

24      def ChannelInput inChannel
25      def ChannelOutput outChannel
26      def int iterations
27      def String initialValue

28      void run() {

29        def One2OneChannel N2A = Channel.createOne2One()
30        def One2OneChannel A2N = Channel.createOne2One()

31        def ChannelInput toAgentInEnd = N2A.in()
32        def ChannelInput fromAgentInEnd = A2N.in()
33        def ChannelOutput toAgentOutEnd = N2A.out()
34        def ChannelOutput fromAgentOutEnd = A2N.out()

35        def theAgent = new Agent( results: [initialValue])

36        for ( i in 1 .. iterations) {
37          outChannel.write(theAgent)
38          theAgent = inChannel.read()
39          theAgent.connect ( [fromAgentOutEnd, toAgentInEnd ] )
40          def agentManager = new ProcessManager (theAgent)
41          agentManager.start()
42          def returnedResults = fromAgentInEnd.read()
43          println "Root: Iteration: $i is $returnedResults "
44          returnedResults << "end of " + i
45          toAgentOutEnd.write (returnedResults)
46          agentManager.join()
47          theAgent.disconnect()
48        }
49      }
50    }
```

Listing 18-3 The Root Process Definition

18.2.3 The Process Node

The `ProcessNode` simply provides the process that is executed at each of the nodes on the ring of processes which the agent visits. Its structure is shown in Listing 18-4. The `inChannel` and `outChannel` properties {52, 53} provide the channel connections to the ring of channels connecting all the processes together. The property `nodeId` {54} is just an integer identifier for the node. The mechanism by which the node is connected to the agent {56-61} is identical to that previously described for the `Root` process {29-34}. The value of `nodeId` is copied into a `localValue` variable {62}.

The main body of the process is an infinite loop {63} and is almost identical to that previously described for the `Root` process in that on receipt of `theAgent` {64} they are `connected` together {65} and started within a `ProcessManager` {66, 67}. The only difference is that the `results` passed from `theAgent` to this process are read into `currentList` {68}. The value of `localValue` is then appended to `currentList` {69} before it is written back to the agent {70}. Once `theAgent` has `disconnected` {72} from this process it can be `written` to the `outChannel` for transfer to the next process on the ring {73}. Finally, `localValue` is incremented by `10` {74} as this makes it easier to observe the behaviour after a number of iterations around the ring of processes.

```
51     class ProcessNode implements CSProcess{

52       def ChannelInput inChannel
53       def ChannelOutput outChannel
54       def int nodeId

55       void run() {

56         def One2OneChannel N2A = Channel.createOne2One()
57         def One2OneChannel A2N = Channel.createOne2One()

58         def ChannelInput toAgentInEnd = N2A.in()
59         def ChannelInput fromAgentInEnd = A2N.in()
60         def ChannelOutput toAgentOutEnd = N2A.out()
61         def ChannelOutput fromAgentOutEnd = A2N.out()

62         def int localValue = nodeId

63         while (true) {
64           def theAgent = inChannel.read()
65           theAgent.connect ( [fromAgentOutEnd, toAgentInEnd] )
66           def agentManager = new ProcessManager (theAgent)
67           agentManager.start()
68           def currentList = fromAgentInEnd.read()
69           currentList << localValue
70           toAgentOutEnd.write (currentList)
71           agentManager.join()
72           theAgent.disconnect()
73           outChannel.write(theAgent)
74           localValue = localValue + 10
75         }
76       }
77     }
```

Listing 18-4 The Process Node Definition

A sample of the output from the console window is shown in Output 18-1. The number of nodes, excluding the Root node is 6 and the agent will travel round the ring of processes three times. The initial value passed to the results property of the agent was “ex1”. This execution of the network of processes is achieved using a script that runs each process as a concurrent process within a single JVM using the script `RunAgentSystem`, available on the accompanying web site.

```
Number of Nodes ? 6
Number of Iterations ? 3
Initial List Value ? ex1
Root: Iteration: 1 is ["ex1", 1, 2, 3, 4, 5, 6]
Root: Iteration: 2 is ["ex1", 1, 2, 3, 4, 5, 6, "end of 1",
                                11, 12, 13, 14, 15, 16]
Root: Iteration: 3 is ["ex1", 1, 2, 3, 4, 5, 6, "end of 1",
                                11, 12, 13, 14, 15, 16, "end of 2",
                                21, 22, 23, 24, 25, 26]
```

Output 18-1 Sample Console Window for the First Agent System

At the end of iteration 1 we observe that the `nodeId` of each node has been appended to the `results` list. At the end of iteration 2, we observe that the “end of” iteration marker has been added to results and then the modified `localValue` (incremented by 10 {74}) has been appended. At the end of iteration 3 we observe that the “end of” marker for the second iteration and the doubly incremented `localValues` have also been appended to `results`. Thus we have constructed an agent that traverses a ring of processes, collecting data from each node and retaining it within its own internal structures. The agent makes these collected data values able to a root node, before resuming its transit around the network.

18.3 Running the Agent on a Network of Nodes

More realistically we need to run the processes and root on separate nodes of a TCP/IP network such that each process runs in its own JVM. This is simply achieved by the `RunNode` script Listing 18-5 and a `RunRoot` script Listing 18-6.

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

79     def int nodeId = Ask.Int ("Node identification? ", 1, 9)
80     def Boolean last = Ask.Boolean ("Is this the last node? - ( y or n):")

81     def fromRingName = "ring" + nodeId
82     def toRingName = (last) ? "ring0" : "ring" + (nodeId + 1)

83     def fromRing = CNS.createNet2One(fromRingName)
84     def toRing = CNS.createOne2Net(toRingName)

85     def processNode = new ProcessNode ( inChannel: fromRing,
86                                         outChannel: toRing,
87                                         nodeId: nodeId)

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

Listing 18-5 The Run Node Script

It is presumed that the `CNS` is running, after which any number of nodes can be initiated, provided the nodes are identified from 1 to n in sequence, where n is the number of such nodes. The node identification is used to distinguish the names of the net channels used to construct the ring of processes. A node is initialised {78}, after which its `nodeId` is obtained {79}. It is determined whether this is the `last` node or not {80}. The names of the net channels can then be formed. The input to a node, `fromRingName` {81}, is always the name “ring*n*”, where n is the `nodeId`. The output from a node depends upon whether it is the `last` node or not {82}. In the case of the `last` node its output name is “ring0”, otherwise it is “ring” suffixed by the `nodeId` of the next node in sequence. These names can then be used to create the network channels appropriately {83, 84}. The node can now be constructed {85-87} and executed {88}.

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

90     def int iterations = Ask.Int ("Number of Iterations ? ", 1, 9)
91     def String initialValue = Ask.string ( "Initial List Value ? ")

92     def fromRingName = "ring0"
93     def toRingName = "ring1"

94     def fromRing = CNS.createNet2One(fromRingName)
95     def toRing = CNS.createOne2Net(toRingName)

96     def rootNode = new Root ( inChannel: fromRing,
97                               outChannel: toRing,
98                               iterations: iterations,
99                               initialValue: initialValue )

100    new PAR ( [rootNode] ).run()
```

Listing 18-6 The Run Root Script

The script to run the root node is very similar, except that we need to determine the number of `iterations` {90} and the `initialValue` of the `results` list {91}. The input to the root node is always named “ring0” and its output “ring1” {92, 93}. The node is then constructed {96-99} and executed {100}. The output from this set of nodes is similar to that of the above system and in particular, the output from the root node is identical for the same number of nodes and iterations. This can be observed by running the required node scripts available on the accompanying web site.

18.4 Result Returning Agent

The previous, relatively simple agent will be modified so that as it passes from node to node as well as collecting a value from the node, it returns that value directly to the root node. The only modifications required are to the agent and the root process. The node process is not changed in any way because the processing is contained within the agent itself.

18.4.1 The BackAgent Specification

The `BackAgent` is shown in Listing 18-7. An additional property, `backChannel`, is required {104} that is the location of an anonymous net channel used by the agent to return values back to the root node.

```
101    class BackAgent implements MobileAgent {

102      def ChannelOutput toLocal
103      def ChannelInput fromLocal
104      def NetChannelLocation backChannel
105      def results = [ ]

106      def connect ( List c ) {
107        this.toLocal = c[0]
108        this.fromLocal = c[1]
109      }

110      def disconnect (){
111        toLocal = null
112        fromLocal = null
113      }

114      void run() {
115        def toRoot = NetChannelEnd.createOne2Net (backChannel)
116        toLocal.write (results)
117        results = fromLocal.read()
118        def last = results.size - 1
119        toRoot.write(results[last])
120        toRoot.destroyWriter()
121      }
122    }
```

Listing 18-7 The BackAgent Specification

The `run` method {114-121} is also modified slightly to permit the return value communication. The agent initially makes the connection for the `backChannel` creating an anonymous net output channel `toRoot` {115}. The interaction with the node is the same as before {116, 117}. The index of the `last` element in the `results` list is determined {118} and this element is then written to the `toRoot` channel {119}. Finally, the resources associated with the `toRoot` channel are recovered {120}.

18.4.2 The Back Root Process

Listing 18-8 shows the structure of the `BackRoot` process. The properties of the process are the same as for `Root` (Listing 18-3), except that an additional property, `backchannel` {104} is required to provide the anonymous `NetChannelInput` of the channel that connects the `BackAgent` to the `BackRoot`, when it is running in a process node. The channels required to connect the `BackRoot` process to the `BackAgent`, when the agent is running in the `BackRoot` process are then defined and their input and output ends created {130-135}. The channel address of the `backChannel` is then obtained and stored as `backChannelLocation` {136}.

```
123    class BackRoot implements CSProcess{

124      def ChannelInput inChannel
125      def ChannelOutput outChannel
126      def int iterations
127      def String initialValue
128      def NetChannelInput backChannel

129      void run() {

130        def One2OneChannel N2A = Channel.createOne2One()
131        def One2OneChannel A2N = Channel.createOne2One()
132        def ChannelInput toAgentInEnd = N2A.in()
133        def ChannelInput fromAgentInEnd = A2N.in()
134        def ChannelOutput toAgentOutEnd = N2A.out()
135        def ChannelOutput fromAgentOutEnd = A2N.out()

136        def backChannelLocation = backChannel.getChannelLocation()

137        def theAgent = new BackAgent( results: [initialValue],
138                                      backChannel: backChannelLocation)

139        def rootAlt = new ALT ( [inChannel, backChannel])
140        outChannel.write(theAgent)
141        def i = 1
142        def running = true
143        while ( running) {
144          def index = rootAlt.select()
145          switch (index) {
146            case 0:
147              theAgent = inChannel.read()
148              theAgent.connect ( [fromAgentOutEnd, toAgentInEnd] )
149              def agentManager = new ProcessManager (theAgent)
150              agentManager.start()
151              def returnedResults = fromAgentInEnd.read()
152              println "Root: Iteration: $i is $returnedResults "
153              returnedResults << "end of " + i
154              toAgentOutEnd.write (returnedResults)
155              def backValue = backChannel.read()
156              agentManager.join()
157              theAgent.disconnect()
158              i = i + 1
159              if (i <= iterations) {
160                outChannel.write(theAgent)
161              }
162              else {
163                running = false
164              }
165              break

166            case 1:
167              def backValue = backChannel.read()
168              println "Root: Iteration $i: received $backValue"
169              break
170          }
171        }
172      }
173    }
```

Listing 18-8 The Back Root Process

An instance of `BackAgent` is then constructed as `theAgent` {137, 138}, with property values of a list containing the element `initialValue` and the `backChannelLocation`. The `BackRoot` process can receive inputs on its `inChannel`, when the `BackAgent` returns to the `BackRoot` process or from the `BackAgent` on the `backChannel` when `BackAgent` is running in another node. The alternative `rootAlt` captures this behaviour {139}. The agent is written to the process' `outChannel` {140}. A count variable `i` {141} and a Boolean `running` {142} are defined and initialised. The main loop of the process now commences {143} with the determination of the source of any input communication {144}.

Case 0 {146} relates to return of the agent from an iteration around the other nodes. The agent is read from `inChannel` {147} into `theAgent` and subsequent processing is the same as previously described, except that a returned value has to be read from `theAgent` on the `backChannel` {155}, which is ignored. It is interesting to note that this communication is in fact a net channel communication between two processes running on the same node because `theAgent` is now executing within the `BackRoot` process. The remainder of this alternative's coding {158-165} relates to the management of the number of iterations and the termination of the process.

Case 1 deals with an input from the agent when it is running on another node. A variable `backValue` is read from `backChannel` {167} and printed {168}.

18.4.3 Running BackRoot

The script to run `BackRoot` is shown in Listing 18-9 and again is very similar to that which ran the `Root` process before.

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

175    def int iterations = Ask.Int ("Number of Iterations ? ", 1, 9)
176    def String initialValue = Ask.string ( "Initial List Value ? ")

177    def fromRingName = "ring0"
178    def toRingName = "ring1"

179    def backChannel = NetChannelEnd.createNet2One()

180    def fromRing = CNS.createNet2One(fromRingName)
181    def toRing = CNS.createOne2Net(toRingName)

182    def rootNode = new BackRoot ( inChannel: fromRing,
183                                  outChannel: toRing,
184                                  iterations: iterations,
185                                  initialValue: initialValue,
186                                  backChannel: backChannel)

187    new PAR ( [rootNode] ).run()
```

Listing 18-9 The Script to Run BackRoot

The only differences are the definition of an anonymous `NetChannelInput backChannel` {179} and its inclusion as a property in the construction of the `BackRoot` process {186}.

18.4.4 Execution of the BackAgent System

Output from running the `BackAgent` system is shown in Output 18-2. The `BackRoot` process is run as shown in Listing 18-9 and the nodes are run using the `RunNode` process (Listing 18-5), without alteration. As the agent progresses round the network of three nodes it can be observed that the `nodeId` (1, 2 and 3) is returned to `BackRoot` from each node. The agent then returns to the `BackRoot` process where the complete contents of the `results` list are output. The agent then goes round the network again and this time augmented values (11, 12, 13) are returned to `BackRoot`. The agent returns to `BackRoot` and the extended set of values in `results` are printed. This is then repeated for the final iteration.

```
Number of Iterations ? 3
Initial List Value ? ex2
Root: Iteration 1: received 1
Root: Iteration 1: received 2
Root: Iteration 1: received 3
Root: Iteration: 1 is ["ex2", 1, 2, 3]
Root: Iteration 2: received 11
Root: Iteration 2: received 12
Root: Iteration 2: received 13
Root: Iteration: 2 is ["ex2", 1, 2, 3, "end of 1", 11, 12, 13]
Root: Iteration 3: received 21
Root: Iteration 3: received 22
Root: Iteration 3: received 23
Root: Iteration: 3 is ["ex2", 1, 2, 3, "end of 1", 11, 12, 13,
                                       "end of 2", 21, 22, 23]
```

Output 18-2 Output From the BackRoot Console Window

18.5 An Agent with Forward and Back Channels

In this variation an agent is constructed that reads a value from the root process, modifies the data held within the agent; that data is then sent to the node running the agent, where the data is again modified and returned to the agent. The agent then returns the last value added to the data back to the root node before moving to the next node. This is a relatively simple modification of `BackAgent` but demonstrates that a large amount of functionality can be built into agents built using parallel processing capabilities in conjunction with network communications.

18.5.1 The Forward and Back Agent

Listing 18-10 shows the changes made to the `run` method of the `BackAgent` (Listing 18-7) to achieve the required effect. Initially an anonymous net input channel, `fromRoot` is created {189} and its net channel location determined {190}. Once the back channel, `toRoot`, has been created {191}, it is used to write the `fromRootLocation` to the root process {192}. A value is then read from the `fromRoot` channel and appended to the `results` list {193}. Finally, the resources associated with the `fromRoot` channel are destroyed once they are no longer required {199}

```
188      void run() {

189        def fromRoot = NetChannelEnd.createNet2One()
190        def fromRootLocation = fromRoot.getChannelLocation()

191        def toRoot = NetChannelEnd.createOne2Net (backChannel)

192        toRoot.write(fromRootLocation)
193        results << fromRoot.read()

194        toLocal.write (results)
195        results = fromLocal.read()
196        def last = results.size - 1
197        toRoot.write(results[last])

198        toRoot.destroyWriter()
199        fromRoot.destroyReader()
200      }
201    }
```

Listing 18-10 The Modified Forward Back Agent

18.5.2 The Forward Back Root Process

The only changes required to the `BackRoot` process (Listing 18-8) to create the process that also has a forward channel are shown in Listing 18-11. These changes both occur in the while loop and are identical in both cases within the switch statement.

```
202        while ( running) {
203          def index = rootAlt.select()
204          switch (index) {
205            case 0:
206              theAgent = inChannel.read()
207              theAgent.connect ( [fromAgentOutEnd, toAgentInEnd] )
208              def agentManager = new ProcessManager (theAgent)
209              agentManager.start()

210              def forwardLocation = backChannel.read()
211              def forwardChannel = NetChannelEnd.createOne2Net(forwardLocation)
212              forwardChannel.write (rootValue)
213              rootValue = rootValue - 1

214              def returnedResults = fromAgentInEnd.read()
215              println "Root: Iteration: $i is $returnedResults "
216              returnedResults << "end of " + i
217              toAgentOutEnd.write (returnedResults)
218              def backValue = backChannel.read()
219              agentManager.join()
220              theAgent.disconnect()
221              i = i + 1
222              if (i <= iterations) {
223                outChannel.write(theAgent)
224              }
225              else {
226                running = false
227              }
228              break
229            case 1:
230              def forwardLocation = backChannel.read()
231              def forwardChannel = NetChannelEnd.createOne2Net(forwardLocation)
232              forwardChannel.write (rootValue)
233              rootValue = rootValue - 1

234              def backValue = backChannel.read()
235              println "Root: During Iteration $i: received $backValue"
236              break
237          }
238        }
```

Listing 18-11 The Changes Required to BackRoot to Create ForwardBackRoot

The location of the forward channel is read from `backChannel` {210, 230} into `forwardLocation`. This is then used to create the output end of an anonymous net channel `forwardChannel` {211, 231}. A variable `rootValue`, initially -1, is written to the `forwardChannel` {212 232} and then its value is decremented by 1 {213, 233}. This means that the agent and the root processes have created a pair of anonymous net channels that connect the two processes over which values can be interchanged as required by the application. The agent can then interact with the process running on the remote node as needed.

18.5.3 Forward back System Output

Output 18-3 shows typical output from the forward and back connected agent and root system. The processes were running using the same `RunNode` script as before and a minor modification {182} to the `RunBackAgent` script to invoke the `ForwardBackRoot` was required to that shown in Listing 18-9.

It can be seen that the output is very similar except that a negative number appears in the `results` list before each new value is appended. At the end of each iteration a further negative number is appended, which is the value appended when the agent is resident with the `ForwardBackRoot` process but for which no value is appended by the root process itself {210-213}.

```
Number of Iterations ? 3
Initial List Value ? ex3
Root: During Iteration 1: received 1
Root: During Iteration 1: received 2
Root: During Iteration 1: received 3
Root: Iteration: 1 is ["ex3", -1, 1, -2, 2, -3, 3, -4]
Root: During Iteration 2: received 11
Root: During Iteration 2: received 12
Root: During Iteration 2: received 13
Root: Iteration: 2 is ["ex3", -1, 1, -2, 2, -3, 3, -4, "end of 1",
                                     -5, 11, -6, 12, -7, 13, -8]
Root: During Iteration 3: received 21
Root: During Iteration 3: received 22
Root: During Iteration 3: received 23
Root: Iteration: 3 is ["ex3", -1, 1, -2, 2, -3, 3, -4, "end of 1",
                                    -5, 11, -6, 12, -7, 13, -8, "end of 2",
                                    -9, 21, -10, 22, -11, 23, -12]
```

Output 18-3 Typical Output from the Forward backward System

18.6 Let's Go On A trip

In this final version, the ring of channels connecting the processes is dispensed with. A number of independent nodes will be created each of which has a connection to a root node using an `Any2One` net channel. Each node will create a net input channel, the location of which will be sent to the root process. The root process will create a list of these individual node net channel locations, together with a net input channel location for the root process. This list of net locations will be passed to the agent. The agent will be sent to the first node in the list, where it will undertake some interaction with the local node that will cause the updating of a results list held within the agent. The agent will then disconnect itself from the node and cause itself to be written to the next node in the list of net channel locations. In due course it will return to the root node where the results list will be printed. Thus the agent is going on a trip, the precise ordering of which, it has no knowledge of in advance.

18.6.1 The Trip Agent

The `TripAgent`, shown in Listing 18-12, has local channels {240, 241} that enable its connection to the node upon which it is hosted. The property `tripList` {242} will hold the net channel locations that form the trip the agent will travel. The `pointer` property {243} indicates the next element in `tripList` that is the location to which the agent will travel. The result property is a list that will be modified as the agent travels to each node. The `connect` and `disconnect` methods are identical to those used in previous agents {245-252}.

The `run` method initially `writes` the current `results` list to the node process {254} using the `toLocal` channel and then reads the modified version of `results` from the channel `fromLocal` {255}. The remainder of the processing deals with `tripList` processing.

It is presumed that the zero'th element of `tripList` contains the net channel location for the root process. Thus, once the value of `pointer` reaches zero, the trip has finished and in this case a simple message is printed {263} because the agent can be sent to no other nodes.

If the value of `pointer` is greater than zero {256} then the agent can be transferred to the next node in `tripList`, indicated by (`pointer - 1`). A net channel location is obtained from `tripList` using the `List` method `get()` and this is then used to create an anonymous `One2Net` output channel variable called `nextChannel`{258}. The agent then `disconnects` itself from the local node because the `toLocal` and `FromLocal` properties will not be `Serializable` as they refer to addresses within this node. The agent can now be written to `nextChannel` using the reference to itself `this` {260}.

```
class TripAgent implements MobileAgent {

  def ChannelOutput toLocal
  def ChannelInput fromLocal
  def tripList = [ ]
  def int pointer
  def results = [ ]

  def connect ( List c ) {
    this.toLocal = c[0]
    this.fromLocal = c[1]
  }

  def disconnect (){
    toLocal = null
    fromLocal = null
  }

  void run() {
    toLocal.write (results)
    results = fromLocal.read()

    if (pointer > 0) {
      pointer = pointer - 1
      def nextChannel = NetChannelEnd.createOne2Net (tripList.get(pointer))
      disconnect()
      nextChannel.write(this)
    }
    else {
      println "Agent has returned to TripRoot"
    }
  }
}
```

Listing 18-12 The Trip Agent Definition

18.6.2 The Trip Node Process

Listing 18-13 shows the coding of the `TripNode` process. The property `toRoot` {268} is the net output channel by which the process can communicate its net input channel location to the root process. The property `nodeId` is the unique integer identification of this node {269}. Within the `run` method {270} channels are created {271,272} together with their channel ends {273-276} which provide the internal channel mechanism by which the agent communicates with the host node, as described previously (see section 21.2.2).

An anonymous net input channel is then defined, `agentInputChannel` {277}, and its channel location is written to the root process using the `toRoot` net output channel {278}. The node process now waits until it can read `theAgent` from the `agentInputChannel` {279}.

Using the local channels, `theAgent` can be connected to the local node and then executed using a `ProcessManager` {280-282}. The interaction with the agent then takes place {283-285}, after which the `agentManager` can `join` the node process {286}, so that in this case they can both terminate.

```
class TripNode implements CSProcess{

  def ChannelOutput toRoot
  def int nodeId

  void run() {

    def One2OneChannel N2A = Channel.createOne2One()
    def One2OneChannel A2N = Channel.createOne2One()
    def ChannelInput toAgentInEnd = N2A.in()
    def ChannelInput fromAgentInEnd = A2N.in()
    def ChannelOutput toAgentOutEnd = N2A.out()
    def ChannelOutput fromAgentOutEnd = A2N.out()

    def agentInputChannel = NetChannelEnd.createNet2One()
    toRoot.write ( agentInputChannel.getChannelLocation())

    def theAgent = agentInputChannel.read()

    theAgent.connect ( [fromAgentOutEnd, toAgentInEnd] )
    def agentManager = new ProcessManager (theAgent)
    agentManager.start()

    def currentList = fromAgentInEnd.read()
    currentList << nodeId
    toAgentOutEnd.write (currentList)

    agentManager.join()
  }
}
```

Listing 18-13 The Trip Node Process

18.6.3 The Trip Root Process

Listing 18-14 shows the coding of the `TripRoot` process. This is very similar to previous root processes until the part that deals with the inputting of the net channel input locations from the nodes. The `fromNodes` channel {290} is the net input channel used by each of the nodes to communicate the location of the net channel to be used by the agent in forming its `tripList`. The channels used to connect locally to the agent are set up {294-299}.

The `tripList` is initialised with the net channel location of the `fromNodes` channel and will be the last element to be accessed in the list thereby ensuring that `TripRoot` is the last process in the trip {300}. The `for` loop {301-304} then inputs the `fromNodes` channel the net input channel location of each of the nodes, which are appended to `tripList`. The next section of coding {305-310} gets the last element of `tripList`, which becomes the net location to which the agent will be sent first. An anonymous net output channel, `firstNodeChannel`, is created from the location. An instance of the `TripAgent` is then constructed as `theAgent` after which it can be `written` to the `firstNodeChannel`.

The remainder of the coding shows the return of `theAgent` after the trip. It will be read from the channel `fromNodes` {311}. The process interaction between `theAgent` and the `TripRoot` process is very similar to other such root nodes {312-319}.

```
class TripRoot implements CSProcess{

  def ChannelInput fromNodes
  def String initialValue
  def int nodes

  void run() {

    def One2OneChannel N2A = Channel.createOne2One()
    def One2OneChannel A2N = Channel.createOne2One()

    def ChannelInput toAgentInEnd = N2A.in()
    def ChannelInput fromAgentInEnd = A2N.in()
    def ChannelOutput toAgentOutEnd = N2A.out()
    def ChannelOutput fromAgentOutEnd = A2N.out()

    def tripList = [ fromNodes.getChannelLocation() ]

    for ( i in 0 ..< nodes) {
      def nodeChannelLocation = fromNodes.read()
      tripList << nodeChannelLocation
    }

    def firstNodeLocation = tripList.get(nodes)
    def firstNodeChannel = NetChannelEnd.createOne2Net(firstNodeLocation)
    def theAgent = new TripAgent( tripList: tripList,
                                   results: [initialValue],
                                   pointer: nodes)
    firstNodeChannel.write(theAgent)

    theAgent = fromNodes.read()
    theAgent.connect ( [fromAgentOutEnd, toAgentInEnd] )
    def agentManager = new ProcessManager (theAgent)
    agentManager.start()
    def returnedResults = fromAgentInEnd.read()
    println "TripRoot: has received $returnedResults "
    toAgentOutEnd.write (returnedResults)
    agentManager.join()
    theAgent.disconnect()
  }
}
```

Listing 18-14 The Trip Root Process

18.6.4 Running a Trip Node Process

The script to run a node of the system is shown in Listing 18-15. The `Any2Net` channel `toRoot` forms the channel between the nodes and the root process.

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

def int nodeId = Ask.Int ("Node identification? ", 1, 9)

def toRoot = CNS.createAny2Net("toRoot")

def processNode = new TripNode ( toRoot: toRoot,
                                  nodeId: nodeId)

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

Listing 18-15 The Script To Run TripNode

18.6.5 Running the Trip Root Process

The script to run the root of the system is shown in Listing 18-16. The `Net2Onet` channel `fromNodes` forms the net input channel from the nodes to the root process.

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

def String initialValue = Ask.string ( "Initial List Value ? ")
def int nodes = Ask.Int ("Number of nodes? ", 1, 9)

def fromNodes = CNS.createNet2One("toRoot")

def rootNode = new TripRoot ( fromNodes: fromNodes,
                              nodes: nodes,
                              initialValue: initialValue )

new PAR ( [rootNode] ).run()
```

Listing 18-16 The Script to Run TripRoot

18.6.6 Output From the Trip System

The output shown in Output 18-4 was produced by a system that comprised four nodes and the trip root process. The nodes were initialised in numerical sequence but, as can be seen, the agent visited the nodes in a different order. This reflects the way in which the underlying system deals with inputs on an `Any2One` net channel and the order in which processes are executed.

```
Initial List Value ? ex4
Number of nodes? 4
TripRoot: has received ["ex4", 3, 1, 2, 4]
Agent has returned to TripRoot
```

Output 18-4 Typical Output From the Trip System

18.7 Summary

Agents are generally considered to have their roots in actor models which are self contained, interactive, concurrently executing objects, having internal state and that respond to messages from other agents [Nwana]. More prosaically, an agent is that which "denotes something that produces or is capable of producing and effect" [Rothermel] and which can migrate to many hosts thereby demonstrating that the "concept of mobile agent supports 'process mobility'". Mobile Agents are also considered to have their own thread of control and to respond to received messages [Pham]. More recently, [Chalmers] has argued that more correctly a CSP process together with the required network communication can be seen to implement the relatively simple Mobile Agent concept described above. In the next chapter we shall introduce a mobile process capability, where a process is loaded over a network to undertake processing at a host node.