21 Mobile Agents and Processes: Process Discovery

In this chapter we consider the difference between mobile agents and mobile processes by offering a defining example because this area is currently very confused as to their precise meaning. Previously, we have used these terms in a manner that reflects these definitions but up till now there has been no need to be specific about their differences because there has been no conflict.

In this example we are going to dynamically add nodes to a network. Nodes that are added may not have all the processes they require; even the initial nodes may not contain a full repertoire of processes. However, we shall assume that all the initial nodes do contain all the required data manipulation processes. When a node receives a data input for which it does not have the required process it will send an agent around the currently connected nodes. The goal of the agent is to locate a node with the required process, acquire the process, return to its original node and transfer the process into its home node. Upon receipt of such a process the home node will dynamically connect it into its internal channel structure and thereby cause the process to execute. Thus the node will now be able to process any further data of the same type.

It can be seen from the above description that the mobile agent has a specific goal, or possibly goals that it seeks to achieve. This may require the agent to visit many nodes in order to find the solution to its goal. Once the goal has been achieved, including any return to its home node it then ceases to exist. Other agents with similar goals may be created but each will have a predetermined life expectancy. By contrast a mobile process is one that can be moved from one node to another, plugged into the channel structure at the receiving node and then continues to run as part of the node until such time as the node itself is no longer required. In some cases the mobile process may retain the ability to communicate with its original node, as was the case of the Meeting Organiser system described in Chapter 19.

The system architecture is shown in Figure 21-1. The `DataGenerator` process provides a named network input channel that can be connected to by any node, shown as a dotted arrow, thereby creating a networked `Any2One` channel. Similarly, the `Gatherer` process provides a named network input channel that can also be connected to by any node as indicated by the dotted arrow.

On creation, a `NodeProcess` simply needs to be told the names of these channels in order to be connected to both the `DataGenerator` and `Gatherer` processes. Once these connections have been made, the `NodeProcess` creates a number of net input channels as follows. The From Data Generator channel provides a means by which data can be received from the `DataGenerator` by the `NodeProcess`. The Agent Visit Channel is the channel upon which agents from other nodes will be input so they can interact with this node. The Agent Return Channel is the channel used by an agent to return to its originating node.

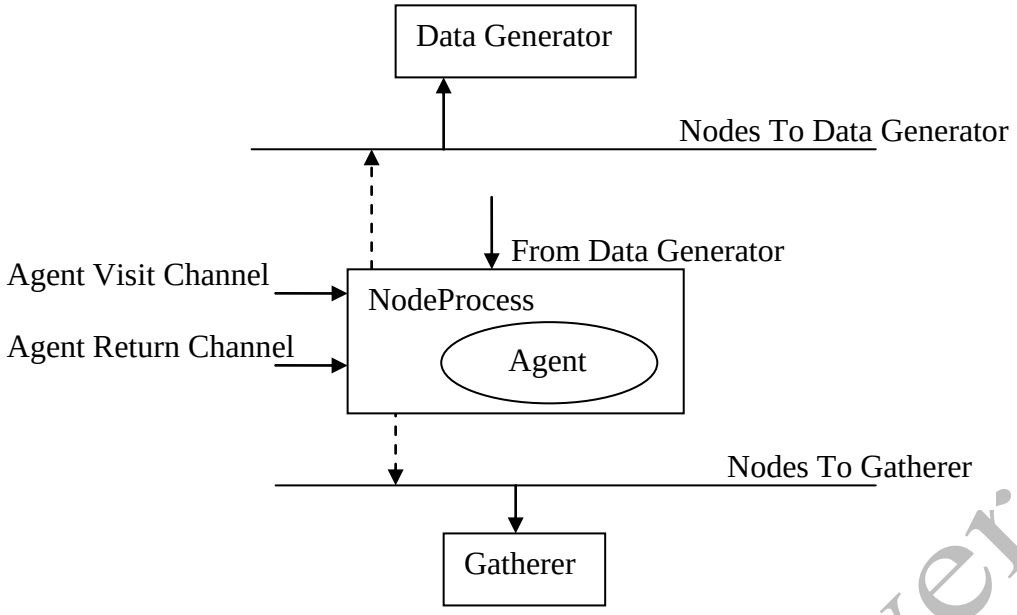

Figure 21-1 Architecture of the Mobile Processes and Agents System

Once theses channels have been created the `NodeProcess` outputs the location of the From Data Generator and the Agent Visit Channel to the `DataGenerator` using the Nodes to Data Generator channel. On receiving these locations the `DataGenerator` creates a `One2One` channel from it to the node using the From Data Generator channel location. The `DataGenerator` maintains a list of all the Agent Visit Channel locations, which it outputs to all of the connected `NodeProcesses` whenever the list changes. The `NodeProcess` uses this information to update its Agent with the locations of the Agent Visit Channels that it can use when it searches for a data processing process. In addition, the Node also ensures that the Agent holds the location of the Agent Return Channel so that a returning Agent knows its home location.

Once the system has been invoked, the `DataGenerator` randomly sends data object instances of any type to any of the nodes. If a `NodeProcess` already has an instance of the required data processing process the data is sent to that process where it is manipulated and subsequently output to the `Gatherer` process. If the node does not have an instance of the required process then it informs the Agent of the process it requires and causes the Agent to be sent to the first location on its list of Agent Visit Channel locations. In due course the Agent will return, the new process will be transferred to the Node and it will be connected into the Node as described previously. As soon as a Node sends an Agent to find a required process it creates another instance of its Agent so that should another data object arrive for which it does not have the data processing process then an Agent can be sent to find it immediately.

The operation of a Node matches the interactions described above. On receipt of an input it determines if it is a list of Agent Visit Channel locations and if so updates the Agent appropriately. If it is an instance of a data object, it determines its type and if it already has an instance of the required process sends the data object to the required process. Otherwise, it sends the Agent the required process type information, which the Agent can use when visiting the other nodes. Each of the processes in a `NodeProcess` is invoked using the `ProcessManager` class. When a process is received by a `NodeProcess` it creates a channel by which the `NodeProcess` can send data objects to it. All such processes are connected to the Nodes To Gatherer channel. Once a `NodeProcess` has received three such processes, its internal architecture would be as shown in Figure 21-2, ignoring its Agent.

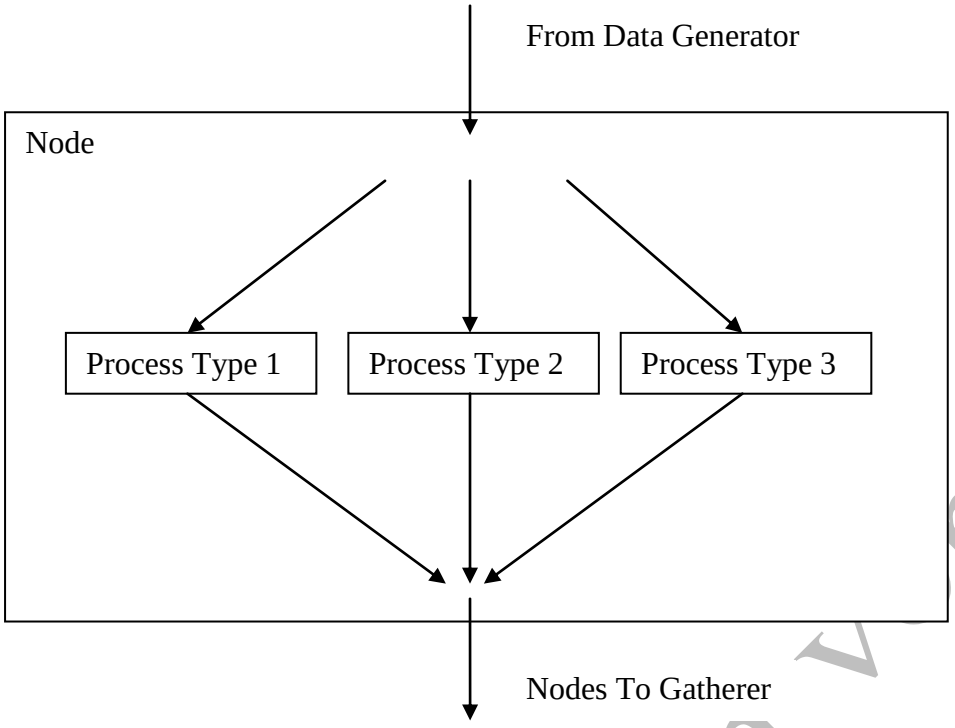

Figure 21-2 The Node Architecture

21.1 The Adaptive Agent

The agent implements the interface described in Chapter 19 and thus it needs to define channels by which it can connect to processing nodes and also methods called `connect` and `disconnect` need to be defined. These are shown in Listing 21-1. The channel `fromInitialNode` {2} provides the input to the agent from the node where it is created and initialised with the task it is to undertake. The channel `fromVisitedNode` {3} provides an input from any node the agent visits and similarly `toVisitedNode` {4} provides an output connection a visited node. Finally, `toReturnedNode` {5} provides a connection to the initial node once the agent has returned from its trip.

An agent can be in one of three states as represented by the value of the variables `initial`, `visiting` or `returning` {6-8}. When they are created agents are in the `initial` state {6}, which means they have yet to be sent on a trip by the node which created them and they are connected to the node by the channel `fromInitialNode`. In the `visiting` state {7} an agent has left the creating node and is attached to another node by the `fromVisitedNode` and `toVisitedNode` channels. Finally, in the `returned` state {8} the agent has returned to the creating node and is connected to it by the `toReturnedNode` channel.

```
01    class AdaptiveAgent implements MobileAgent, Serializable {

02      def ChannelInput fromInitialNode
03      def ChannelInput fromVisitedNode
04      def ChannelOutput toVisitedNode
05      def ChannelOutput toReturnedNode
06      def initial = true
07      def visiting = false
08      def returned = false
09      def availableNodes = [ ]
10      def requiredProcess = null
11      def returnLocation
12      def processDefinition = null
13      def homeNode

14      def connect (List c) {
15        if (initial) {
16          fromInitialNode = c[0]
17          returnLocation = c[1]
18          homeNode = c[2]
19        }
20        if (visiting) {
21          fromVisitedNode = c[0]
22          toVisitedNode = c[1]
23        }
24        if (returned) {
25          toReturnedNode = c[0]
26        }
27      }

28      def disconnect() {
29        fromInitialNode = null
30        fromVisitedNode = null
31        toVisitedNode = null
32        toReturnedNode = null
33      }
```

Listing 21-1 The Adaptive Agent Properties and connect Method

The list `availableNodes` {9} is used to hold the net channel locations of the nodes' `agentVisitChannels` known to the system and is the set of nodes it can visit. The property `requiredProcess` {10} will be initialised to the name of the process for which the agent will search. When the agent is initialised the value of `returnLocation` {11} will be set to the net channel location of the node's `agentReturnChannel`. The property `processDefinition` {12} will hold the required process' definition once it has been located in a visited node. The value of `homeNode` {13} will be set to the identity of the node from which the agent originates.

The behaviour of the `connect` method {14-27} varies depending upon its state. In all cases it is passed a `List` containing one or more elements. These elements will be the value of some of the properties described above as required by the current state of the agent. The `disconnect` method {28-33} is always the same and ensures that any channel properties of the agent are set to `null`.

The `run` method of the agent is shown in Listing 21-2 and its behaviour depends upon the state of the agent. It should be noted that in any state the agent always terminates. If the agent is in the `returned` state it writes a list comprising the `processDefinition` and the name of the `requiredProcess` to its `toReturnedNode` channel {35-37}. It is presumed that the agent will always find the `requiredProcess`. Once this communication has been completed the agent will terminate completely and do no further processing whatsoever and will thus in due course be garbage collected.

```
34    void run( ) {

35      if (returned) {
36        toReturnedNode.write([processDefinition, requiredProcess])
37      }

38      if (visiting) {
39        toVisitedNode.write(requiredProcess)
40        processDefinition = fromVisitedNode.read()
41        if ( processDefinition != null ) {
42          toVisitedNode.write(homeNode)
43          visiting = false
44          returned = true
45          def nextNodeLocation = returnLocation
46          def nextNodeChannel = NetChannelEnd.createOne2Net(nextNodeLocation)
47          disconnect()
48          nextNodeChannel.write(this)
49        }
50        else {
51          disconnect()
52          def nextNodeLocation = availableNodes.pop()
53          def nextNodeChannel = NetChannelEnd.createOne2Net(nextNodeLocation)
54          nextNodeChannel.write(this)
55        }
56      }
57      if (initial) {
58        def awaitingTypeName = true
59        while (awaitingTypeName) {
60          def d = fromInitialNode.read()
61          if ( d instanceof List) {
62            for ( i in 0 ..< d.size) { availableNodes << d[i] }
63          }
64          if ( d instanceof String) {
65            requiredProcess = d
66            awaitingTypeName = false
67            initial = false
68            visiting = true
69            disconnect()
70            def nextNodeLocation = availableNodes.pop()
71            def nextNodeChannel =NetChannelEnd.createOne2Net(nextNodeLocation)
72            nextNodeChannel.write(this)
73          }
74        }
75      }
76    }
77  }
```

Listing 21-2 The Adaptive Agent's run Method

If the agent is `visiting` another node {38-56}; it first writes the name of the `requiredProcess` to the visited node {39} and always reads a reply on the `fromVisitedNode` channel {40} into its `processDefinition` property. Its subsequent behaviour depends on whether or not the visited node had the required process definition. If the `processDefinition` is not `null` the agent writes the identity of the agent's `homeNode` to the visited node {42} so that it can keep a record of the nodes to which it has sent the process definition. The value of the state variables `visiting` and `returned` are updated as required {43, 44}. The value of `nextNodeLocation` is set to the agent's `returnLocation` {45} and this is then used to create a `NetChannelOutput` {46}. The agent `disconnect`s from the visited node {47} and then writes itself back to its original node {48}. The behaviour of the agent when it has not received a process definition is very similar except that it `pop`s the next visiting node location from its list of `availableNodes` {52} and uses that to create a net channel {53} on which it writes itself to the next node on its trip {54}.

When the agent is in the `initial` state then it has to wait until such time as it receives the name of a process which it is to find. During that time it may also receive notification of the registration of a new node in the network, which it has to add to its list of `availableNodes`. This behaviour is captured {57-75} by first defining a Boolean `awaitingTypeName` to `true`, which can then be used to control a terminating `while` loop {59-74}. An object is read in from the `fromInitialNode` channel, and its type is determined {61, 64}. If it is a `list` then the agent has received an update to the list of available nodes

{62}. If it is a `String` then the agent has received the name of a process for which it should search. The agent makes the necessary preparation before writing itself to the first node in the list of `availableNodes` it can visit. The name of the `requiredProcess` is stored {65} and then the values of `awaitingName` {66}, `initial` {67} and `visiting` {68} are updated. The agent `disconnects` {69} itself and then causes itself to be written to another node {70-72}.

21.2 The Node Process

The `NodeProcess`, unusually, has no channel properties, instead the names of the globally available network channels are passed as the parameters, `toGathererName` {79} and `toDataGenName` {80}. These names are then used to connect to the respective network channels. The `processList` {81} is used to hold any initial data processing processes with which the node may be initialised; as further processes are obtained these will be appended to this list. The `vanillaList` {82} contains a possibly empty list of data processing processes; initially it is identical to the `processList`. Once processes have been initialised within the `processList` they cannot then be used as the basis for sending a data processing process to another node because they will contain internal connections within the node that cannot be `disconnected` and which also render it not serializable. The process definitions in the `vanillaList` are never directly executed and are thus serializable; they are used to send to other nodes when requested by a visiting agent. The property `nodeId` {83} is used to uniquely identify a particular node.

The first part of the `NodeProcess`'s `run` method deals with its connection to the outside network environment and the internal structures needed to invoke the data processing processes. The fist requirement is to connect to the `DataGenerator` process. The `CNS` can be used directly to make the network output channel connection, using the network channel name, `toDataGenName` {85}. The network input channel, `fromDataGen`, connecting the `DataGenerator` process to the node is created as an anonymous channel {86}. The input channels by which agents visit and return to a node are now created as a set of anonymous network channels. The `agentVisitChannel` and its associated location are defined {87-88}, followed by the `agentReturnChannel` {89-90}. The `agentReturnChannelLocation` is used subsequently in the initialisation of this `Node`'s agent. The location of the `agentVisitChannel` together with the location of the `fromDataGen` channel and `nodeId` are written to the `DataGenerator` using an object that has a single list property, `dgl`, using the channel `toDataGen` {91-93}.

The list `connectChannels` {94} is used to create a set of internal channels that are used subsequently within the node to connect to the data processing processes as they are initialised. The order in which data processing processes arrive and are initialised is not determined and thus we need to record the order in which they appear, both in the `processList` and the `vanillaList`; this is the use of `typeOrder` {95} and `vanillaOrder` {96} respectively. The list `currentSearches` {97} is used to record the names of the data processing processes for which an agent has already been sent. A node can initiate multiple parallel searches for different data processing processes and thus needs to ensure that it does not create an agent to undertake a search that has already been started. The variable `cp` {98} is used to count the number of processes in `processList`.

```
78  class NodeProcess implements CSProcess {
79    def String toGathererName
80    def String toDataGenName
81    def processList = null
82    def vanillaList = null
83    def int nodeId

84    void run() {
85      def toDataGen = CNS.createAny2Net(toDataGenName)
86      def fromDataGen = NetChannelEnd.createNet2One()

87      def agentVisitChannel= NetChannelEnd.createNet2One()
88      def agentVisitChannelLocation = agentVisitChannel.getChannelLocation()
89      def agentReturnChannel= NetChannelEnd.createNet2One()
90      def agentReturnChannelLocation = agentReturnChannel.getChannelLocation()

91      toDataGen.write( new DataGenList (dgl:[fromDataGen.getChannelLocation(),
92                                             agentVisitChannelLocation,
93                                             nodeId] ) )

94      def connectChannels = [ ]
95      def typeOrder = [ ]
96      def vanillaOrder = [ ]
97      def currentSearches = [ ]
98      def cp = 0

99      if (processList != null) {
100       for ( i in 0 ..< processList.size) {
101         def processType = processList[cp].getClass().getName()
102         def typeName = processType.substring(0,
103                                              processType.indexOf("Process"))
104         typeOrder << typeName
105         vanillaOrder << typeName
106         connectChannels[cp] = Channel.createOne2One()
107         def pList = [connectChannels[cp].in(), nodeId, toGathererName]
108         processList[cp].connect(pList)
109         def pm = new ProcessManager(processList[cp])
110         pm.start()
111         cp = cp + 1
112       }
113     }

114     def NodeToInitialAgent = Channel.createOne2One()
115     def NodeToVisitingAgent = Channel.createOne2One()
116     def NodeFromVisitingAgent = Channel.createOne2One()
117     def NodeFromReturningAgent = Channel.createOne2One()

118     def NodeToInitialAgentInEnd = NodeToInitialAgent.in()
119     def NodeToVisitingAgentInEnd = NodeToVisitingAgent.in()
120     def NodeFromVisitingAgentOutEnd = NodeFromVisitingAgent.out()
121     def NodeFromReturningAgentOutEnd = NodeFromReturningAgent.out()

122     def myAgent = new AdaptiveAgent()
123     myAgent.connect([ NodeToInitialAgentInEnd,
124                       agentReturnChannelLocation, nodeId])
125     def initialPM = new ProcessManager(myAgent)
126     initialPM.start()

127     def nodeAlt = new ALT([ fromDataGen,
128                             agentVisitChannel,
129                             agentReturnChannel])
130     def currentVisitList = [ ]
```

Listing 21-3 The Node Process Definitions

The next section {99-113} deals with the instantiation of any data processing processes in the `processList` that can be ignored if `processList` is `null` {99}. In this exemplar the names of the data types and associated data processing process have been restricted so that the name of a data type is specified as `Type1` and its associated data process as `Type1Process`, for example. This has been utilised in the coding of this section where the names of data types have been extracted from the associated data process. The coding is undertaken for each data process in `processList` {100-112}. The name of a data process is extracted into `processType` {101} using the `getClass` and `getName` methods. The name of the data type is then obtained and placed in `typeName` {102-103} using the `substring` method and knowledge of the structure of the names as described previously. The `typeName` is then appended to both

typeOrder {104} and vanillaOrder {105}. The channel which connects the NodeProcess to the specific data processing process now needs to be created and stored in the list connectChannels {106}, in the same relative position as typeName appears in typeOrder. A list pList is now created {107} comprising the in() end of this newly created channel, the nodeId and the name of the network channel that connects the data processing process to the Gatherer process. This list is then used as a parameter of the connect method call on the current element of processList {108}, which will cause the building of all the required channel connections. An instance of ProcessManager is now created {109} with the current element of processList as its process and it is then started {110}.

The channels used internally to connect an agent to the NodeProcess are now defined {114-117} and then the channel ends that will be required to be sent to the agent so that it can communicate with the NodeProcess are created {118-121}. An instance of the AdaptiveAgent is now created as myAgent {122} and a call to it's connect method {123-124} ensures that myAgent can; communicate with the NodeProcess, knows its agentReturnChannelLocation and also its home node identity. The agent is now started using another instance of ProcessManager {125-126}.

Once a NodeProcess is running it can receive inputs from the channels known as fromDataGen, the agentVisitChannel and the agentReturnChannel and thus these are all incorporated into an alternative nodeAlt {127-129}. Finally, an empty list currentVisitList is defined that will be used subsequently to maintain the list of agentVisitChannels that an agent, sent from this NodeProcess, can visit.

Processing within a NodeProcess is dependent on which channel is selected in nodeAlt {132}. Each case is dealt with in a separate Listing. Listing 21-4 shows the processing when an input from the DataGenerator process is received. The data is read into the variable d {134} and subsequent processing depends upon the data type. If the input is an instance of AvailableNodeList {135} then it is necessary to remove this node's agentVisitChannelLocation from the anl property of the input because there is no point sending an agent to its home node if we already know the node does not contain the required data processing process. The members of d.anl are then appended to currentVisitList {138} and written to the local agent using the NodeToInitialAgent {139} channel. Note that it is necessary to explicitly refer to the out() end of the channel when writing to it because the channel was created within the process and not passed in as a ChannelOutput property of the process.

If the input data is one of the data types recognised by the system then the specific data type is obtained {142} and subsequent processing depends on the availability of that data type's process in the processList. If the data type is known to the node because it is a member of typeOrder {143} then its position in that list can be determined {144-153}. The data can then be sent to the required data processing process by writing to the corresponding member of the connectChannels list {154}.

If the input data is not recognised then a search has to be started to obtain the required data processing process, only if a search has not already been started {157}. In this case the type for which the search is being started is appended to currentSearches {158} and also written to the agent using the NodeToInitialAgent channel {159}. The join method is then called {160} on the ProcessManager, initialPM, running the agent, which causes the NodeProcess to wait until the agent has stopped execution. Another instance of the AdaptiveAgent is then created {161} and connected {162-163} to the node. It is then passed to a new ProcessManager instance {164} and started {165}. The currentVisitList is then written to the newly created agent {166}.

```
131     while (true) {
132       switch ( nodeAlt.select() ) {
133         case 0:
134           def d = fromDataGen.read()
135           if ( d instanceof AvailableNodeList ) {
136             d.anl.remove(agentVisitChannelLocation)
137             currentVisitList = [ ]
138             for ( i in 0 ..< d.anl.size) { currentVisitList << d.anl[i] }
139             NodeToInitialAgent.out().write(currentVisitList)
140           }
141           else {
142             def dType = d.getClass().getName()
143             if ( typeOrder.contains(dType) ) {
144               def i = 0
145               def notFound = true
146               while (notFound) {
147                 if (typeOrder[i] == dType) {
148                   notFound = false
149                 }
150                 else {
151                   i = i + 1
152                 }
153               }
154               connectChannels[i].out().write(d)
155              }
156             else {  // do not have process for this data type
157               if ( ! currentSearches.contains(dType)) {
158                 currentSearches << dType
159                 NodeToInitialAgent.out().write(dType)
160                 initialPM.join()
161                 myAgent = new AdaptiveAgent()
162                 myAgent.connect([ NodeToInitialAgentInEnd,
163                                   agentReturnChannelLocation, nodeId ])
164                 initialPM = new ProcessManager(myAgent)
165                 initialPM.start()
166                 NodeToInitialAgent.out().write(currentVisitList)
167               }
168               else {
169               }
170             }
171           }
172           break
```

Listing 21-4 Node Process: Data Input Processing

Listing 21-5 shows the processing that occurs when an agent visits another node. The agent is read from the `agentVisitChannel` {174} and then appropriately `connect`ed to the node {175-176} by means of the required channel ends. An instance of `ProcessManager` is created {177} to run the visiting agent {178}. The node then reads the type of the data processing process required from the visiting agent into `typeRequired` {179}. If the node contains in the `vanillaOrder` list the `typeRequired` {180} then a search of the `vanillaOrder` is undertaken to find the position of the process {181-190}. The required process can then be obtained from the `vanillaList` of processes and written to the agent using `the NodeToVisitingAgent.out()` channel {191}. The visiting agent then writes its home node identity to the node using the `NodeFromVisitingAgent.in()` channel into `agentHome` {192}. If the node does not have the `typeRequired` in its `vanillaOrder` then a `null` is written to the visiting agent {195}. The interaction with the visiting agent is now complete so the node process can `join` the agent, waiting for it to terminate {197}, thereby completing the processing of a visiting agent.

```
173         case 1:
174           def visitingAgent = agentVisitChannel.read()
175           visitingAgent.connect([NodeToVisitingAgentInEnd,
176                                  NodeFromVisitingAgentOutEnd ])
177           def visitPM = new ProcessManager(visitingAgent)
178           visitPM.start()
179           def typeRequired = NodeFromVisitingAgent.in().read()
180           if ( vanillaOrder.contains(typeRequired) ) {
181             def i = 0
182             def notFound = true
183             while (notFound) {
184               if (vanillaOrder[i] == typeRequired) {
185                 notFound = false
186               }
187               else {
188                 i = i + 1
189               }
190             }
191             NodeToVisitingAgent.out().write(vanillaList[i])
192             def agentHome = NodeFromVisitingAgent.in().read()
193           }
194           else {  // do not have process for this data type
195             NodeToVisitingAgent.out().write(null)
196           }
197           visitPM.join()
198           break
```

Listing 21-5 Node Process: Visiting Agent Processing

The processing associated with a returned agent is shown in Listing 21-6. The `returnAgent` is read from the `agentReturnChannel` {200} and then `connected` to the node {201}. An instance of `ProcessManager` is created for the `returnAgent` {202} in which it is started {203}. The `returnList` is obtained from the `returnAgent` using the `NodeFromReturningAgent.in()` channel{204}. This completes the interaction with the return agent and thus the node process can `join` the `returnPM` awaiting its termination {205}. The data elements from the `returnList` can now be processed updating the node data as required. Specifically, the `returnedType` can be removed from the list of `currentSearches` {207} and appended to the `typeOrder` list {208}. A channel is created that is used to `connect` the node to the newly acquired data processing process {209} in the `connectChannels` list. The body of the data processing process, returned as `returnList[0]` is appended to the `processList` {210}. A list of properties required for connection is then created as `pList` {211} and `connected` {212}. An instance of the `ProcessManager` is created {213} and used to `start` the process {214}. The value of `cp`, which keeps a record of the first empty element in `processList`, is then incremented {215}.

```
199         case 2:
200           def returnAgent = agentReturnChannel.read()
201           returnAgent.connect([NodeFromReturningAgentOutEnd])
202           def returnPM = new ProcessManager (returnAgent)
203           returnPM.start()
204           def returnList = NodeFromReturningAgent.in().read()
205           returnPM.join()
206           def returnedType = returnList[1]
207           currentSearches.remove([returnedType])
208           typeOrder << returnedType
209           connectChannels[cp] = Channel.createOne2One()
210           processList << returnList[0]
211           def pList = [connectChannels[cp].in(), nodeId, toGathererName]
212           processList[cp].connect(pList)
213           def pm = new ProcessManager(processList[cp])
214           pm.start()
215           cp = cp + 1
216           break
217       }
218     }
219   }
220 }
```

Listing 21-6 Node Process: Return Agent Processing

21.3 The Data Generator Process

The version of the DataGenerator processes presented here assumes there are only three different types of data that can be processed called Type1, Type2 and Type3. It also presumes that at least three nodes will be initialised before the system starts to run. It is also assumed that all the required data processing processes will be available, in some combination, in the first three nodes that are run.

The ChannelInput fromNodes {222} is the channel used by all the NodeProcesses to communicate with the DataGenerator process. The interval {223} is used as a timer alarm in an alternative that is selected every cycle to determine whether any new nodes have been registered.

The ChannelOutputList toNodes {225} contains the list of net channel output ends used to communicate data to each of the registered NodeProcesses. Similarly, the agentVisitChannelList {226} maintains a list of all the agentVisitChannelLocations sent to the DataGenerator. The allocationList {227} holds the identity of each registered NodeProcess. The variable rng {228} is an instance of Random and timer is an instance of CSTimer {229}. The DataGenerator process alternates over the fromNodes channel and a timer alarm as indicated in the ALT dgAlt {230}. Each data type has its own associated type instance value {231-233} initialised as shown and there is also a global instanceValue {234} initialised to zero. These instance values will be used to differentiate instances of generated data type objects. A count of the number of registered nodes is kept in nodesRegistered {235}.

The never ending loop {236-278} of the process now starts; by defining and initialising two Boolean variables, checkingForNewNodes {237} and nodeAppended {238}; and by setting the timer alarm {239}. A while loop is now entered {240-254} that is always executed at the start of every cycle; and during initialisation of the process will ensure that at least three nodes are registered {240}. The loop commences with the selection of a ready alternative from dgAlt {241}.

```
221 class DataGenerator implements CSProcess {

222   def ChannelInput fromNodes
223   def interval = 1000

224   void run() {
225     def ChannelOutputList toNodes = new ChannelOutputList()
226     def agentVisitChannelList = [ ]
227     def allocationList = [ ]
228     def rng = new Random()
229     def timer = new CSTimer()
230     def dgAlt = new ALT ([fromNodes, timer])
231     def type1Instance = 1000
232     def type2Instance = 2000
233     def type3Instance = 3000
234     def instanceValue = 0
235     def nodesRegistered = 0

236     while (true) {
237       def checkingForNewNodes = true
238       def nodeAppended = false
239       timer.setAlarm (timer.read() + interval)
240       while (checkingForNewNodes || (nodesRegistered < 3)){
241         switch ( dgAlt.select()) {
242           case 0:
243             def nodeData = fromNodes.read()
244             toNodes.append ( NetChannelEnd.createOne2Net ( nodeData.dgl[0]))
245             agentVisitChannelList << nodeData.dgl[1]
246             allocationList << nodeData.dgl[2]
247             nodesRegistered = nodesRegistered + 1
248             nodeAppended = true
249             break
250           case 1:
251             checkingForNewNodes = false
252             break
253         }
254       }
255       if (nodeAppended) {
256         toNodes.write(new AvailableNodeList ( anl: agentVisitChannelList))
257       }

258       def nNodes = toNodes.size()
259       def nodeId = rng.nextInt(nNodes)
260       switch ( rng.nextInt(3) + 1) {
261         case 1:
262           toNodes[nodeId].write ( new Type1 (typeInstance: type1Instance,
263                                              instanceValue: instanceValue ))
264           type1Instance = type1Instance + 1
265           break
266         case 2:
267           toNodes[nodeId].write ( new Type2 (typeInstance: type2Instance,
268                                              instanceValue: instanceValue ))
269           type2Instance = type2Instance + 1
270           break
271         case 3:
272           toNodes[nodeId].write ( new Type3 (typeInstance: type3Instance,
273                                              instanceValue: instanceValue ))
274           type3Instance = type3Instance + 1
275           break
276       }
277       instanceValue = instanceValue + 1
278     }
279   }
280 }
```

Listing 21-7The Data Generator Process

If a new node is registering itself then data from that node is read from the channel `fromNodes` into `nodeData` {243}. The `nodeData` is of type `DataGenList` that has a single list property `dgl` from which various data items can be extracted. Thus `dgl[0]` contains the location of the node's `fromDataGen` net channel location, which can be used to create a net channel end that can be `append`ed to the list `toNodes` {244}. The content of `dgl[1]` is the registering node's `agentVisitChannelLocation` that can be `append`ed to the `agentVisitChannelList` {245}. Finally, `dgl[2]` contains the identity of the registering node and is `append`ed to the `allocationList` {246}. The values of `nodesRegistered` and `nodeAppended` are updated appropriately {247, 248}.

If the alternative selected from `dgAlt` is that corresponding to the `timer` alarm then the value of `checkingForNewNodes` is set `false` {251}. In all but the initialisation phase of the system this will cause the loop to terminate so that `DataGenerator` can progress to the next phase. If a node has been registered (or three nodes during the initialisation phase) then the updated `agentVisitChannelList` is `written` as the property `anl` of a new `AvailableNodeList` object to all the registered nodes in parallel using the `ChannelOutputList toNodes` {255-257}.

The last part {258-278} of the `DataGenerator` process creates a random data object of one of the defined types and sends it to a randomly chosen node on the network. The variable `nNodes` {258} is set to the number of nodes in the network. It is then used to choose the `nodeId` {259} to which the data object will be written. The `switch` {260} determines the type identifier of the data type to be generated. It is encoded that one of three types will be created. In each case only the `Type` of the created data varies; the content of each `Type` is identical. The `Type` instance variable for example `type2Instance` is incremented {269} as is the global `instanceValue` {277}. The `instanceValue` is used to uniquely identify each data object that is created.

21.4 The Gatherer Process

The `Gatherer` process, shown in Listing 21-8, simply reads all inputs it receives on its net `Any2One` input channel, `fromNodes` {285} and then prints out the received data `d` {286}. It should be noted that all data objects that are generated by `DataGenerator` will not be processed by the `Gatherer` process. If a `NodeProcess` receives a data object for which it does not have the required data type processing process then an agent is created and sent to find the required data processing process from another node. At this point the `NodeProcess` throws away the incoming data object because it does not know how to process it. The effect of this is that there is a discontinuity in the sequence of the global `instanceValues` read by the `Gatherer` process that reflects the amount of time that it takes the agent to find the required data processing process and return to the originating node.

```
281 class Gatherer implements CSProcess{
282   def ChannelInput fromNodes
283   void run() {
284     while (true) {
285       def d = fromNodes.read()
286       println "Gathered ${d.toString()}"
287     }
288   }
289 }
```

Listing 21-8 The Gatherer Process

21.5 Definition of the Data Processing Processes

The basic structure of a data Type process is shown in Listing 21-9, which shows the `Type1` class definition. The only difference between them is the `class` name and the value in `typeName` {291}. The `modify` {294-296} method is called in the associated data type processing process. Obviously, this effect could be achieved polymorphically using a single data type processing process but that would have meant that the need to go and retrieve the required processes over the next would have been removed and that was the purpose of the example.

```
290 class Type1 implements Serializable {
291   def typeName = "Type1"
292   def int typeInstance
293   def int instanceValue

294   def modify ( value) {
295     typeInstance = typeInstance + value
296   }

297   def String toString(){
298     return "Instance of $typeName instance
                          $typeInstance value $instanceValue"
299   }
300 }
```

Listing 21-9 The Type1 Class Definition

The data processing processes implement the abstract class `GroovyMobileProcess` shown in Listing 21-10. This itself extends the `MobileProcess` defined within JCSP by adding two further abstract methods `connect` and `disconnect` that allow the integration of a mobile process into an existing channel structure and, if required, its removal.

```
301 abstract class GroovyMobileProcess extends MobileProcess
302                                    implements Serializable {
303   abstract connect(x)
304   abstract disconnect()
305 }
```

Listing 21-10 Groovy Mobile Process Abstract Class Definition

Listing 21-11 shows an implementation of `GroovyMobileProcess` for the `Type1` data processing process called `Type1Process`. The `String` property `toGatherName` {307} is the name of the network channel by which this process can connect itself to the net input channel of the `Gatherer` process. The property `inChannel` {308} is used to connect this process to the `NodeProcess`. The property `nodeId` {309} uniquely identifies the node in which this process is executing.

The `connect` {310-314} method is called by the `NodeProcess` {212} once the process definition has been transferred from a returned agent. The parameter of the method is a list comprising elements containing the values of the properties `inChannel`, `nodeId` and `toGatherName`. The `disconnect` method is defined, but never called; it does however implement the only required functionality, which is to make any channel reference known to the `GroovyMobileProcess` `null` {316}.

The `run` method {318-325} initially creates the connection to the network channel identified by `toGatherName` {319} as `toGatherer`. Thereafter the process reads in data from `inChannel` into a variable of defined type {321}, in this case `Type1` thereby ensuring that only data of the required type is processed. The `modify` method is then called on the input data {322}, after which the data is written to the `Gatherer` process {323}. By judicious choice of `nodeId` and `typeInstance` {295} values it is possible to determine the `NodeProcess` to which a particular instance of a data type was sent and processed.

```
306 class Type1Process extends GroovyMobileProcess implements Serializable {

307   def String toGathererName
308   def ChannelInput inChannel
309   def int nodeId

310   def connect (l) {
311     inChannel = l[0]
312     nodeId = l[1]
313     toGathererName = l[2]
314   }

315   def disconnect () {
316     inChannel = null
317   }

318   void run() {
319     def toGatherer = CNS.createAny2Net(toGathererName)
320     while (true) {
321       def Type1 d = inChannel.read()
322       d.modify(nodeId)
323       toGatherer.write(d)
324     }
325   }
326 }
```

Listing 21-11 The Type1 Process Definition

21.6 Running The System

Three scripts are required to run the system over a network, the first of which shown in Listing 21-12 will be modified to run each individual `NodeProcess` as required. The method call `Node.info.setDevice ( null )` {327} has the effect of turning off all the output generated by the CNS in the console window, which makes it much easier to read and interpret. A `Node` in the network is then created {328}. The names associated with the network channels that connect a `NodeProcess` to the `DataGenerator` and `Gatherer` processes are then defined {329, 330}. In this case a process is to be defined that contains all the data type processing processes and thus `pList` and `vList` are created with instances of these processes {331, 332}. An instance of a `NodeProcess` is now defined with all the required property definitions {333-338} and `run` {339}.

```
327 Node.info.setDevice ( null )
328 Node.getInstance().init(new TCPIPNodeFactory())

329 def toDataGenName = "NodesToDataGen"
330 def toGathererName = "NodesToGatherer"

331 def pList = [ new Type1Process(), new Type2Process(), new Type3Process() ]
332 def vList = [ new Type1Process(), new Type2Process(), new Type3Process() ]

333 def processList = new NodeProcess ( nodeId: 100000,
334                                     toGathererName: toGathererName,
335                                     toDataGenName: toDataGenName,
336                                     processList: pList,
337                                     vanillaList: vList
338                                   )

339 new PAR ([ processList]).run()
```

Listing 21-12 The Script to Run a Node

Listing 21-13 shows the script that `run`s an instance of the `DataGenerator` process. The effect of having an `interval` of 500 milliseconds {344} is that once the system is running normally, that is, after all `NodeProcesses` have acquired all the data type processing processes a data value will be generated every half-second. Reducing this value and running the system on a real network gives some indication of how long it takes for an agent to travel around the network to find an instance of a particular data type processing process. This can be determined by the number of data objects that are missed by a particular node as it waits for the return of the required process.

```
340 Node.info.setDevice ( null )
341 Node.getInstance().init(new TCPIPNodeFactory())

342 def toDataGen = "NodesToDataGen"
343 def fromNodes = CNS.createNet2One(toDataGen)

344 def processList = new DataGenerator ( fromNodes: fromNodes, interval: 500 )

345 new PAR ([ processList]).run()
```

Listing 21-13 The Script to Run the Data Generator Process

The Gather process script is shown in Listing 21-14.

```
346 Node.info.setDevice ( null )
347 Node.getInstance().init(new TCPIPNodeFactory())

348 def toGatherer = "NodesToGatherer"
349 def fromNodes = CNS.createNet2One(toGatherer)

350 def processList = new Gatherer ( fromNodes: fromNodes )

351 new PAR ([ processList]).run()
```

Listing 21-14 The Script to Run The Gatherer Process

21.7 Typical Output From the Gatherer Process

Output 21-1 shows the output when three nodes with nodeId 10000, 80000 and 90000 have been started where the first of these nodes contains all the data type processes as in Listing 21-12 and the others none. It can be observed immediately that the data objects with the `instanceValues` 0, 1 and 2 have been lost. We can deduce that the first instance of `Type3` data {352} was sent to node 10000 because the `typeInstance` ends in 000. Recall that the `modify` method adds the `nodeId` to the `typeInstance` value. We also observe that data with instance values 6 and 9 have also been lost implying they contained data that was sent to a node that did not have the required data processing process. The versions of the processes on the accompanying web page allow printing of created objects which makes it easier to determine exactly what is happening.

```
352 Gathered Instance of Type3 instance 103000 and value 3
353 Gathered Instance of Type2 instance 802002 and value 4
354 Gathered Instance of Type2 instance 102003 and value 5
355 Gathered Instance of Type1 instance 801001 and value 7
356 Gathered Instance of Type2 instance 902004 and value 8
357 Gathered Instance of Type1 instance 101002 and value 10
358 Gathered Instance of Type3 instance 903003 and value 11
359 Gathered Instance of Type3 instance 103004 and value 12

360 Gathered Instance of Type3 instance 103024 and value 64
361 Gathered Instance of Type3 instance 103025 and value 66
362 Gathered Instance of Type2 instance 102022 and value 67
363 Gathered Instance of Type3 instance 903026 and value 68
364 Gathered Instance of Type2 instance 802023 and value 70
365 Gathered Instance of Type2 instance 802024 and value 71
366 Gathered Instance of Type3 instance 803028 and value 72
367 Gathered Instance of Type3 instance 403029 and value 73
368 Gathered Instance of Type3 instance 403030 and value 74
```

Output 21-1Typical Output From the Gatherer Process

Subsequently a node with the identity 40000 was started {360 onwards} and we observe that data objects with instance values 65 and 69 were lost. When the dynamics of the system are observed in real-time we see the interactions with the `CNS` as net channels are created dynamically to permit the movement of the agent around the system. Furthermore the system can cope with the initiation of a node that also has instances of data type processing processes in its `vanillaList` that are already available and provided all instances are identical there is no problem. The system does not deal with the termination of a `NodeProcess`.

21.8 Summary

The primary goal of this chapter was to demonstrate that a system could be built in which nodes were initiated dynamically and that if they did not have all the required data processing processes then they could initiate an agent to go and find the required process. Further, that such returned processes could be integrated into an existing network of channels and operate as if they had been there from the outset.

Fundamentally, the system implements a `GET` operation on a network resource. The CNS implements the same functionality as an internet-based DNS, in that given a string representation of a net resource it returns an IP address and port number. Thus the nodes and any associated net input channel can all be resolved to an IP address. The location of any data processing process can then be referenced relative to that IP address via the normal folder structure. Provided the folder and all its precedents are publicly available then the process is publicly available. The `DataGenerator` process in this example acts as a repository of the IP addresses and resource location of the other nodes that are known, because it records the location of the agent visit channels of all the registered nodes. In this manner the `DataGenerator` acts as a repository of resources that can be interrogated if a registered node needs to acquire some data or process from another node without each node having to know all possible nodes that are connected. In this example we have distributed this information but it was not strictly necessary.

21.9 Challenge

Modify the example so that when a node discovers that the required data processing process is not available the agent is first sent to the `DataGenerator` process to discover all the nodes to which it can go in order to find the resource it requires. In this manner the system does not have to distribute the location of all the nodes to every node as currently happens.

22 Concluding Remarks: Why Use Groovy Parallel and JCSP?

At the end of this journey we are able to reflect on the capabilities that have been described and considered. We started with four very simple concepts; process, channel, alternative and timer and from these we have been able to construct a wide variety of systems with very different operational requirements and functionality. In general, this has been achieved with one fundamental design pattern, the client-server, together with a small number of programming idioms that facilitate its use; such as the prompted buffer. The intellectual challenge is realised by understanding how to use this pattern and idioms in an effective manner.

In this book I have purposely avoided the use of any formal representation of the process and how networks of such processes can be analysed using formal tools. I believe that the engineering approach based upon the reuse of a single design pattern, which has its basis in the underlying formalism, is the best way of making software engineers appreciate what can be achieved provided the capability we are using has a formal basis. The real world is not populated with sufficient software engineers who have the mathematical skill to be able to undertake the formal analysis of their systems, even with the tools currently available [FDR, Spin].

The increasing availability of multi-core processor based systems is inevitable and the desire to make more effective use of this technology will be an increasing challenge. If the engineers use currently available models and methods then this technology will be increasingly difficult to use. Software engineers therefore require a better model with which they can program such systems. But why leave it to just multi-core systems? Why not better and more effective use of network based workstation systems? We might then be able to move to processing systems that make more effective use of grid-computing because we have a viable model that allows us to interact over any size of network.

The content of this book started at a basic undergraduate level and ended with examples of mobile systems that are still the subject of intense research activity [occam-pi, mobile]. All the examples presented are demonstrable and where necessary operate over a network and employ aspects of mobility. Yet this is achieved in manner that does not require a detailed understanding of the operation of networked systems and in which the definition of a process is not reliant upon whether it is to execute as one among many on a single processor or over a network.

The underlying support is provided by JCSP [] and it has been made easier to assimilate by use of Groovy because it helps to reduce the amount of code that needs writing. These are of relatively little importance of themselves but it is important that they both utilise the underlying JVM. What is really crucial is the JCSP provides an implementation of CSP []. CSP provides the formal framework upon which JCSP is implemented and thereby the engineering support for programmers. The programmers are not concerned with the inner workings of the underlying JCSP package because they can reason about their systems at a much higher level. The programmer is no longer concerned with the detailed workings of a poorly implemented underlying thread model, in effect writing machine code. They can now concentrate on

high-level design of the processes at the application layer; confident, that if they use the building blocks correctly and apply one pattern effectively then the resulting system will operate as expected.

This does not remove the need for testing, which exposes the frailties of a system when exposed to a real operating environment. In this book we have shown how systems can be tested, albeit in a cumbersome manner but which with further research and the development of support tools will make it easier to achieve.

The final chapters have shown how we can exploit the process concept in a more up-to-date setting and how it may address the problems that the software industry is starting to deal with in terms of how to exploit mobility, network connectivity and parallelism effectively. Previously, parallelism has been thought of as providing a solution to the needs of high performance computing, where ultimate speed was the only driving force. Hopefully, with some of the later examples, in particular, the reader will have been convinced that approaching a solution to a problem from the parallel point of view actually makes it easier to achieve a working and effective solution.

The book ends at the point where the examples have to become real problems and which of course tend to be too large to explain within the confines of such a book. Hopefully, however, the book contains sufficient ideas, concepts and capabilities that the solution to larger problems can be broken down into sufficiently small processes that it becomes manageable.

As a final design consideration I offer the advice that if you are having problems with a design and cannot get the design right then the solution is usually to add one or more processes. If a designer tries to restrict the number of processes then that is usually followed by problems. In the future perhaps we will get to the situation where team leaders will ask why a serial solution has been adopted rather than one that relies on parallel design methods!