

Using Concurrency and Parallelism Effectively – II

Jon Kerridge



Jon Kerridge

Using Concurrency and Parallelism Effectively – II

Using Concurrency and Parallelism Effectively – II

2nd edition

© 2015 Jon Kerridge & bookboon.com

ISBN 978-87-403-1039-9

Contents

To view part I, download Using Concurrency and Parallelism Effectively – I

Preface	Part I
Background	Part I
Why Java and Groovy and Eclipse?	Part I
Example Presentation	Part I
Organisation of the Book	Part I
Supporting Materials	Part I
Acknowledgements	Part I
1 A Challenge – Thinking Parallel	Part I
1.1 Concurrency and Parallelism	Part I
1.2 Why Parallel?	Part I
1.3 A Multi-player Game Scenario	Part I
1.4 The Basic Concepts	Part I
1.5 Summary	Part I

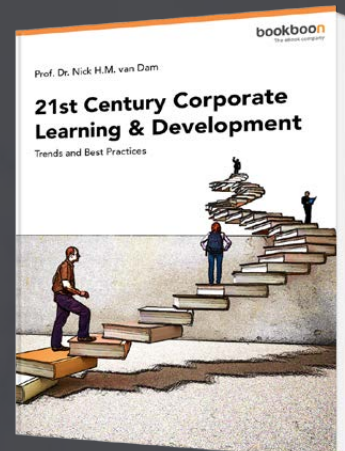


2	Producer Consumer: A Fundamental Design Pattern	Part I
2.1	A Parallel Hello World	Part I
2.2	Hello Name	Part I
2.3	Processing Simple Streams of Data	Part I
2.4	Summary	Part I
2.5	Exercises	Part I
3	Process Networks: Build It Like Lego	Part I
3.1	Prefix Process	Part I
3.2	Successor Process	Part I
3.3	Parallel Copy	Part I
3.4	Generating a Sequence of Integers	Part I
3.5	Testing GNumbers	Part I
3.6	Creating a Running Sum	Part I
3.7	Generating the Fibonacci Sequence	Part I
3.8	Generating Squares of Numbers	Part I
3.9	Printing in Parallel	Part I
3.10	Summary	Part I
3.11	Exercises	Part I

Free eBook on Learning & Development

By the Chief Learning Officer of McKinsey

[Download Now](#)



4	Parallel Processes: Non Deterministic Input	Part I
4.1	Reset Numbers	Part I
4.2	Exercising ResetNumbers	Part I
4.3	Summary	Part I
4.4	Exercises	Part I
5	Extending the Alternative: A Scaling Device and Queues	Part I
5.1	The Scaling Device Definition	Part I
5.2	Managing A Circular Queue Using Alternative Pre-conditions	Part I
5.3	Summary	Part I
5.4	Exercises	Part I
6	Testing Parallel Systems: First Steps	Part I
6.1	Testing Hello World	Part I
6.2	Testing the Queue Process	Part I
6.3	The Queue Test Script	Part I
6.4	Summary	Part I
6.5	Exercises	Part I



Discover the truth at www.deloitte.ca/careers

Deloitte.

© Deloitte & Touche LLP and affiliated entities.



Click on the ad to read more

7	Deadlock: An Introduction	Part I
7.1	Deadlocking Producer and Consumer	Part I
7.2	Multiple Network Servers	Part I
7.3	Summary	Part I
7.4	Exercises	Part I
8	Client-Server: Deadlock Avoidance by Design	Part I
8.1	Analysing the Queue Accessing System	Part I
8.2	Client and Server Design Patterns	Part I
8.3	Analysing the Crossed Servers Network	Part I
8.4	Deadlock Free Multi-Client and Servers Interactions	Part I
8.5	Summary	Part I
8.6	Exercises	Part I
9	External Events: Handling Data Sources	Part I
9.1	An Event Handling Design Pattern	Part I
9.2	Utilising the Event Handling Pattern	Part I
9.3	Analysing Performance Bounds	Part I

© 2013 Accenture. All rights reserved.

be > your degree

Bring your talent and passion to a global organization at the forefront of business, technology and innovation. Discover how great you can be.

Visit accenture.com/bookboon

Be greater than.
consulting | technology | outsourcing

accenture
High performance. Delivered.



9.4	Simple Demonstration of the Event Handling System	Part I
9.5	Processing Multiple Event Streams	Part I
9.6	Summary	Part I
9.7	Exercises	Part I
10	Deadlock Revisited: Circular Structures	Part I
10.1	A First Sensible Attempt	Part I
10.2	An Improvement	Part I
10.3	A Final Resolution	Part I
10.4	Summary	Part I
11	Graphical User Interfaces: Brownian Motion	Part I
11.1	Active AWT Widgets	Part I
11.2	The Particle System – Brownian Motion	Part I
11.3	Summary	Part I
11.4	Exercises	Part I
12	Dining Philosophers: A Classic Problem	Part I
12.1	Naïve Management	Part I
12.2	Proactive Management	Part I
12.3	A More Sophisticated Canteen	Part I
12.4	Summary	Part I
13	Accessing Shared Resources: CREW	Part I
13.1	CrewMap	Part I
13.2	The DataBase Process	Part I
13.3	The Read Clerk Process	Part I
13.4	The Write Clerk Process	Part I
13.5	The Read Process	Part I
13.6	The Write Process	Part I
13.7	Creating the System	Part I
13.8	Summary	Part I
13.9	Challenge	Part I
14	Barriers and Buckets: Hand-Eye Co-ordination Test	Part I
14.1	Barrier Manager	Part I
14.2	Target Controller	Part I
14.3	Target Manager	Part I
14.4	Target Flusher	Part I

14.5	Display Controller	Part I
14.6	Gallery	Part I
14.7	Mouse Buffer	Part I
14.8	Mouse Buffer Prompt	Part I
14.9	Target Process	Part I
14.10	Running the System	Part I
14.11	Summary	Part I
	Index	Part I
	Preface	12
	Organisation of the Book	12
	Supporting Materials	12
15	Communication over Networks: Process Parallelism	14
15.1	Network Nodes and Channel Numbers	15
15.2	Multiple Writers to One Reader	16
15.3	A Single Writer Connected to Multiple Readers	19
15.4	Networked Dining Philosophers	23
15.5	Running the CREW Database in a Network	27
15.6	Summary	33
16	Dynamic Process Networks: A Print Server	34
16.1	Print Spooler Data Objects	35
16.2	The PrintUser Process	37
16.3	The PrintSpooler Process	38
16.4	Invoking The PrintSpooler Node	43
16.5	Invoking A PrintUser Node	44
16.6	Summary	44
17	More Testing: Non-terminating Processes	45
17.1	The Test-Network	47
17.2	The Process Network Under Test	50
17.3	Running The Test	52
17.4	Summary	52
18	Mobile Agents: Going for a Trip	53
18.1	Mobile Agent Interface	53
18.2	A First Parallel Agent System	54
18.3	Running the Agent on a Network of Nodes	59

18.4	Result Returning Agent	62
18.5	An Agent with Forward and Back Channels	68
18.6	Let's Go On A trip	71
18.7	Summary	78
19	Mobile Processes: Ubiquitous Access	79
19.1	The Travellers' Meeting System	80
19.2	The Service Architecture	81
19.3	Universal Client	82
19.4	The Access Server	85
19.5	Group Location Service	89
19.6	Running the System	93
19.7	Commentary	93
20	Redirecting Channels: A Self-Monitoring Process Ring	94
20.1	Architectural Overview	94
20.2	The Receiver process	96
20.3	The Prompter Process	97
20.4	The Queue Process	98
20.5	The State Manager Process	99
20.6	The Stop Agent	100
20.7	The Restart Agent	103
20.8	The Ring Agent Element Process	104
20.9	Running A Node	115
20.10	Observing The System's Operation	116
20.11	Summary	117
20.12	Challenges	117
21	Mobility: Process Discovery	118
21.1	The Adaptive Agent	122
21.2	The Node Process	127
21.3	The Data Generator Process	135
21.4	The Gatherer Process	139
21.5	Definition of the Data Processing Processes	139
21.6	Running the System	142
21.7	Typical Output From the Gatherer Process	144
21.8	Summary	145
21.9	Challenge	145

22	Automatic Class Loading – Process Farms	146
22.1	Data Parallel Architectures	147
22.2	Task Parallel Architectures	151
22.3	Generic Architectures	151
22.4	Architectural Implementation	152
22.5	Summary	165
23	Programming High Performance Clusters	166
23.1	Architectural Overview	167
23.2	The Host and Node Scripts	169
23.3	An Application – Montecarlo Pi	178
23.4	Summary	189
24	Big Data – Solution Scaling	190
24.1	Concordance – A Typical Problem	190
24.2	Concordance Data Structures	191
24.3	The Algorithm	192
24.4	Analysis of Total Time Results	197
24.5	Analysis of Algorithm Phases	198
24.6	Dealing with Larger Data Sets	200
24.7	Implementation of the Scalable Architecture	204
24.8	Performance Analysis of the Distributed System	218
24.9	Summary	222
25	Concluding Remarks	224
25.1	The Initial Challenge – A Review	225
25.2	Final Thoughts	228
26	References	229
	Index	233

Preface

In the second part of *Using Concurrency and Parallelism Effectively* we look at how parallelism can be exploited in a variety of modern computing system environments. These include networked and distributed systems, clusters of workstations and, of course multi-core processors. Multi-core processors have now become ubiquitous and it is nigh on impossible to buy any form of off-the-shelf computing system that does not contain a multi-core processor. One key advantage of using a system based upon the Java Virtual Machine is that we can make use of multi-core processors without any additional effort because the parallel constructs used in this book are able to make immediate and effective use of multi-core processors without any additional effort on the part of the programmer other than ensuring that their code contains sufficient parallel components to utilise the cores. The crucial advantage of the underlying JCSP package is that the definition of the processes does not change as we move from an interleaved concurrent implementation to truly parallel system in which processes are run on different cores or processors over a network. All that changes is the manner in which the processes are invoked.

This capability is more fully exploited as we introduce the capability of using distributed systems that execute on Ethernet based networks and workstation clusters. One of the more challenging aspects of using such clusters and networks is the ability to load the process network over the network from a single 'host' workstation. This challenge is addressed in Chapters 22 and 23 by means of a generic architecture that enables process and class loading over a network in a transparent manner. The only requirement of the programmer is the need to describe the channel connections that connect the processes. This is achieved using a knowledge of the architecture of the network in terms of its IP addresses.

Organisation of the Book

This book assumes that the reader is fully familiar with the material presented in the first book entitled *Using Concurrency and Parallelism Effectively – I*, which is available from the same web site.

This second book presents material that explains how systems can be constructed which run on multiple processing nodes. Chapter 15 provides the basic introduction to multiprocessor systems by taking some of the examples developed in this book and running them on a multi-processor system. Thereafter more complex examples are created that solve a variety of parallel programming problems.

Supporting Materials

The necessary libraries for the packages used are available from the same web site as the book's text. This comprises jar files for the JCSP and Groovy Parallel packages. Documentation for these packages is also provided. A readme text file that describes the library folder structure required is also provided so readers can create the required learning environment quickly and easily.

The source of all the examples is provided in the form of an archived Eclipse project, which can be imported directly into Eclipse, once the required Eclipse environment has been constructed as described in the readme file.

The source code comprises two projects; one called `ChapterExamples` that contains the source of all the examples used throughout the text. The other, `ChapterExercises`, contains supporting material for the Exercises given at the end of some of the Chapters. In these classes there are comments placed where the reader is asked to supply the required coding.

The original version of JCSP included a package called `net`. This enabled the construction of TCP/IP connected process networks in a somewhat cumbersome manner. These versions of the process codes are still available in `ChapterExamples`. Each example has its own PDF file containing the description relating to that example. The text of the book uses a more recent version of the networking package called `net2`. The material presented in the book is based upon the `net2` version of the process networks.

The source of the supporting material is also available in a zip file for non-Eclipse users.

My thanks to the many people who have suggested corrections and other alterations to the text. In particular Joe Bowbeer, who has read both parts in detail and suggested many very helpful corrections.

15 Communication over Networks: Process Parallelism

This chapter begins the second part of the book where we consider networks of processes that execute over TCP/IP networks and thus the systems run in parallel and as such defines some basic concepts:

- net channels
- net channel locations and addresses
- network nodes
- networked versions of the Dining Philosophers and CREW database from previous chapters 12 and 13 are presented to aid understanding

JCSP provides a transparent means of connecting processes, whether they are on the same processor, running concurrently, or if they are executing on separate processes in parallel. Further, we can simulate processes executing in parallel on a single processor by running each process network in a different Java Virtual Machine (JVM). In this latter case there will be no performance gain but the designer is able to assure themselves that the system is functioning as expected. Thus we can test a multi-processor system in a single processor environment knowing that, at the level of inter-process communication the system functions as expected but that it will perform in a different manner when placed on the multi-processor system. If the multi-processor system uses specific hardware then that functional test will have to either be simulated or wait until the system can be tested on the actual hardware.

The definition of the processes that make up the process network, regardless of the processor upon which it will execute and whether or not it is connected to an internal or network channel, remains the same. Thus once a process has been tested, as described in Chapters 6 and 17, we can be assured that it will behave as demonstrated by that testing process. The design of the JCSPNet package architecture means that the actual physical layer used to implement the underlying communication network is completely hidden from the application and in particular the processes and their connection over that network. The only requirement is that the script used to invoke a network of processes on a specific processor utilises a specific network node factory compatible with the underlying communications architecture. Each process network running on a processor has to initialise itself as a node within the communications system.

JCSPNet utilises the underlying Java socket mechanism and thus inherently there are some limitations on the nature of networks that can be constructed. In this chapter we explore these limitations. Initially, the means by which nodes identify themselves to the network and then define network communication channels is described. The JCSP library contains two network architecture packages, `net` and `net2`. The former creates a mechanism similar to a domain name server and requires the creation of names for each network channel. As such it is a little cumbersome to use. The second package, `net2` (Chalmers, 2009) (Chalmers, 2008), utilises the underlying TCP/IP address mechanisms and is thus somewhat more flexible to use. The presented examples use the `net2` package.

15.1 Network Nodes and Channel Numbers

A node in a network is uniquely identified by its TCP/IP address and a port number. The channels connected to a node are each given a unique number, called the Channel Number, to distinguish each of the channels. A processor with a single TCP/IP address can support many distinct network nodes provided they have unique port addresses. A net channel is fully defined by its input channel location which comprises: TCP/IP address: Port / Channel Number.

In the same way that there are `any` and `one` channel connections for ordinary channels then the same applies to net channels as shown in Figure 15-1. Essentially, an output process writes to the net. An input process reads from the net. A `one2net` connection provides a connection between one node and the net. An `any2net` connection provides a connection between multiple nodes and the net. An `any2net` connection has the same properties as in non-network channels in that at one time only one node can write to the `any` end. The output end of a channel, either `one` or `any` is defined by the input channel location.

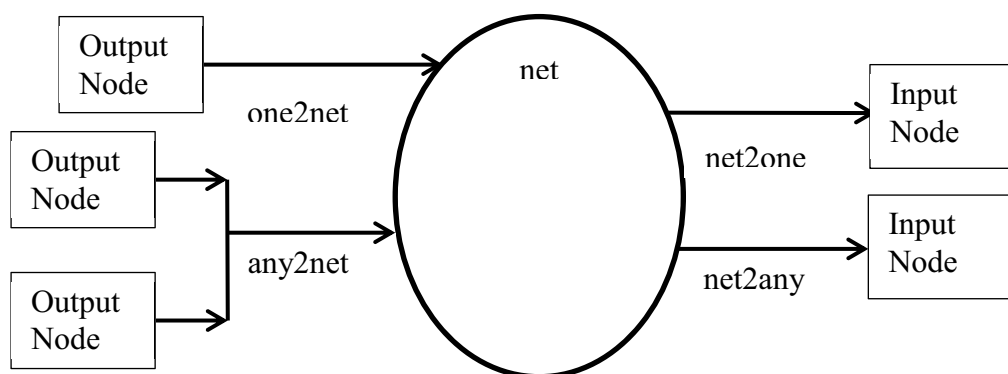


Figure 15-1 The Network Channel Types

A `net2one` channel provides an input connection between the net and a single node. A `net2any` channel provides a connection between the net and any number of processes on the Input Node. A corollary of this definition is that it is not possible to create an `any` to `any` connection between multiple output nodes and multiple input nodes. These aspects are explored in the following sections.

The creation of the communication structure does require that the input end of a net channel is created BEFORE it is referred to by one or more output ends. This means that care has to be taken in the order in which nodes and net channels are created. In some cases it may be necessary to create additional communications to ensure the channels are created in the correct sequence. As a general rule a node should create all its input channels before creating any output channels.

15.2 Multiple Writers to One Reader

Figure 15-2 shows a network comprising many `Sender`, or writing processes, each connected using an `any2net` channel to a network, shown as a heavy dashed line. A single `Receiver`, or reading process is connected to the network by means of a `net2one` channel.

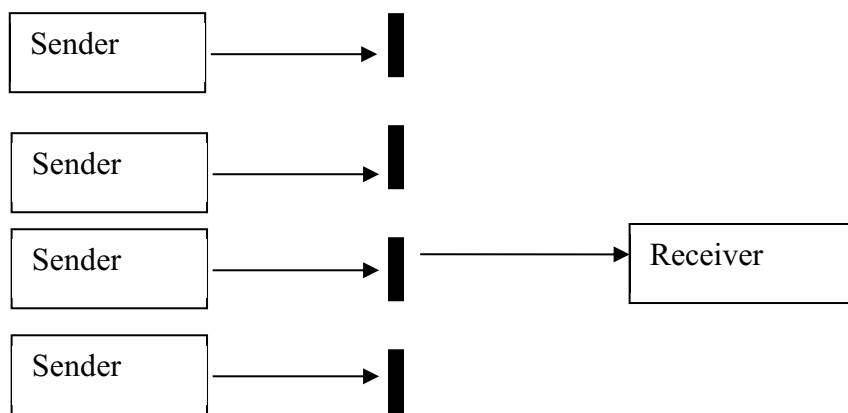


Figure 15-2 The Multi-Sender Single Receiver System

Each process defines its channel property as either a `ChannelInput` in the `Receiver` process or `ChannelOutput` in the `Sender` process. This emphasises the fact that a process definition is totally unconcerned with the nature of the channel used to connect the process to another.

The `Receiver` process repeatedly reads in a value `v` {16} from any of the `Sender` processes and prints it on the console window {17} as shown in Listing 15-1.

```
10 class Receiver implements CProcess {
11
12     def ChannelInput inChannel
13
14     void run() {
15         while (true) {
16             def v = inChannel.read()
17             println "$v"
18         }
19     }
20 }
```

Listing 15-1 The Receiver Process

A Sender process simply waits for 10 seconds {18} and then outputs its identity `String id` {19}, and then repeats the sequence forever, as shown in Listing 15-2.

```
10 class Sender implements CProcess {
11
12     def ChannelOutput outChannel
13     def String id
14
15     void run() {
16         def timer = new CTimer()
17         while (true) {
18             timer.sleep(10000)
19             outChannel.write ( id )
20         }
21     }
22 }
```

Listing 15-2 The Sender Process

The system shown in Figure 15-2 is invoked by initially running the Receiver node as this provides the input end of the channel connecting the nodes. The required script is shown in Listing 15-3.

What if you could build your future and create the future?

The innovation accelerator

One generation's transformation is the next's status quo. In the near future, people may soon think it's strange that devices ever had to be "plugged in." To obtain that status, there needs to be "The Shift".

.....Alcatel-Lucent 

www.alcatel-lucent.com/careers


Click on the ad to read more

In all the following examples we shall use the loop-back TCP/IP address “127.0.0.x” because this makes it easier to run in a single Eclipse environment. Each node of the network of processes is executed in its own JVM within the Eclipse environment, simply by executing the script that creates the node. To run any of the networked systems, on a real network all that is required is to modify the node’s TCP/IP address appropriately.

The node’s address is specified {10} after which a node address is created at that IP address listening on port 3000. The node is then initialised {12}. This operation is common to all node instances and must be completed for each node in the network. A `NetChannel` is now created called `comms` {14}. This uses a call to the method `numberedNet2One()`, which creates an input channel from the net with channel number 100. All the Sender processes must then generate a connection to this input channel. A list comprising one `Receiver` process is then created {15} and the process is run {17}.

```
10 def receiverNodeIP = "127.0.0.1"
11 def receiverNode = new TCPIPNodeAddress(receiverNodeIP, 3000)
12 Node.getInstance().init (receiverNode)
13
14 def comms = NetChannel.numberedNet2One(100)
15 def pList = [ new Receiver ( inChannel: comms ) ]
16
17 new PAR ( pList ).run()
```

Listing 15-3 The Receiver Node

The creation of a Sender node is shown in Listing 15-4. The user is asked to supply the Sender’s identity {21}, which is then used to form the last part of its node’s IP address {10–13}. The Sender node is then created as described previously {14–15}. The sender process has no input channels so we can proceed directly to creating the output channel connection to the Receiver node. The Receiver’s node address is created {17–18} as `receiverNode`. We can then use that node address to create a net output channel {20}. As we want to run many Sender nodes the net output channel is created as `any2net`. The channel number to which the output channel is connected must be the same as the corresponding input channel specified in the `Receiver` node. A list comprising a single process is created {21} and the process run {23}.

```
10 def v= Ask.Int ("Sender identity number 2-9 ? ", 2, 9)
11
12 def senderNodeIPbase = "127.0.0."
13 def senderNodeIP = senderNodeIPbase + v
14 def senderNode = new TCPIPNodeAddress(senderNodeIP, 3000)
15 Node.getInstance().init (senderNode)
16
17 def receiverNodeIP = "127.0.0.1"
18 def receiverNode = new TCPIPNodeAddress(receiverNodeIP, 3000)
19
```

```

20 def comms = NetChannel.any2net(receiverNode, 100)
21 def pList = [ new Sender ( outChannel: comms, id: v ) ]
22
23 new PAR ( pList ).run()

```

Listing 15-4 A Sender Node

Implicitly, an `any2net` connected to a `net2one` channel implements a fair strategy because multiple write requests for the channel are queued in the order they arrive. This can be observed in the output from the system because the order in which messages appear is always the same. The `Receiver` process could incorporate the `net2one` channel in an alternative. In this case the input of a message from the `any2net` channel would be governed by the operation of the associated `ALT`. Thus a queue of messages would build up if the `ALT` did not service the `any2net` channel sufficiently quickly but they would be processed in the order in which they arrived on the channel.

While the network is running further `Sender` nodes can be created, provided they each have a unique identity. The output from `Receiver` will reflect that addition as they occur.

15.3 A Single Writer Connected to Multiple Readers

A single writer process `Put`s data to one of many reader processes that `Get` data from the network. The network structure is shown in Figure 15-3. Each `write` operation undertaken by the `Put` process will be accepted by only one of the `Get` processes. It has to be recalled that each communication over a shared `any` channel always results in a single communication between a pair of processes. Thus in order to show this it will be necessary to make each `Get` process sleep for a short period so that each communication from `Put` has a chance of being read by a different `Get` process. All the `Get` processes have to be created on the same node.

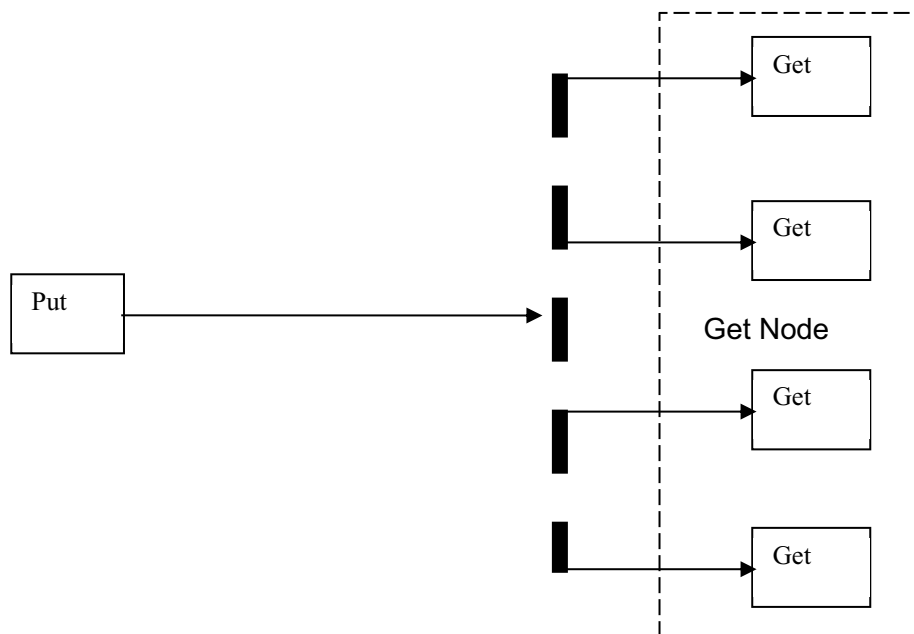
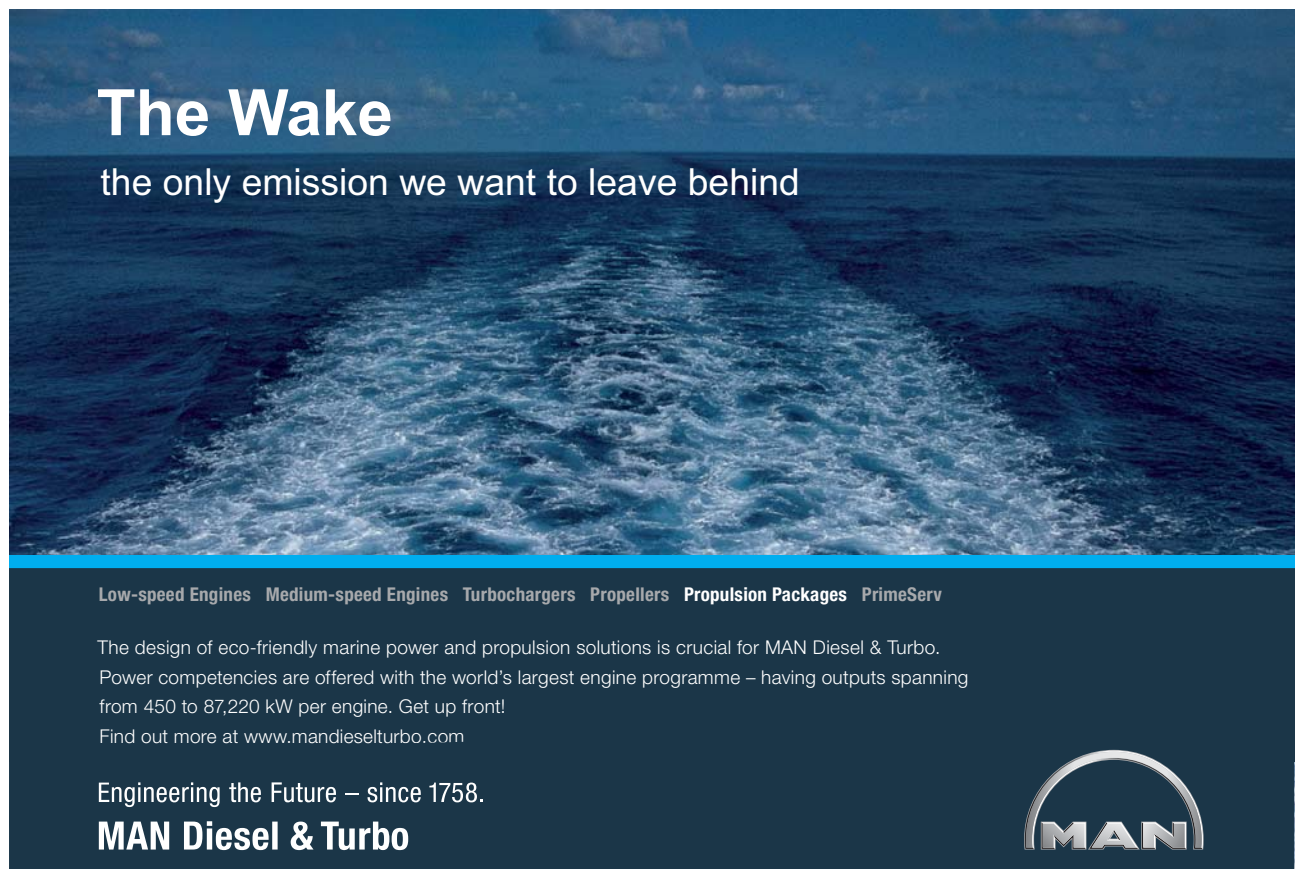


Figure 15-3 One Writer Multiple Reader Process Network

The `Put` process shown in Listing 15-5 shows a process that simply writes a sequence of increasing integers to its `outChannel` {17}. This is done as quickly as possible with no delay between each write operation, though the output will be delayed if there is no input channel ready to read the data.

```
10 class Put implements CProcess {
11
12     def ChannelOutput outChannel
13
14     void run() {
15         def i = 1
16         while (true) {
17             outChannel.write ( i )
18             i = i + 1
19         }
20     }
21 }
```

Listing 15-3 The Put Process



The Wake


the only emission we want to leave behind

Low-speed Engines Medium-speed Engines Turbochargers Propellers Propulsion Packages PrimeServ

The design of eco-friendly marine power and propulsion solutions is crucial for MAN Diesel & Turbo. Power competencies are offered with the world's largest engine programme – having outputs spanning from 450 to 87,220 kW per engine. Get up front! Find out more at www.mandieselturbo.com

Engineering the Future – since 1758.

MAN Diesel & Turbo



Click on the ad to read more

The `Get` process, Listing 15-6, has two properties, its `inChannel` {12} and the identity, `id` {13}, of the `Get` process instance.

```
10 class Get implements CProcess {
11
12     def ChannelInput inChannel
13     def int id = 0
14
15     void run() {
16         def timer = new CTimer()
17         while (true) {
18             def v = inChannel.read()
19             println "$id .. $v"
20             timer.sleep(200 * v)
21         }
22     }
23 }
```

Listing 15-4 The Get Process

A `timer` is defined {16} and within the processes' main loop a value `v` {18} is read from the `inChannel` and then printed {19} after which the process sleeps for a time proportional to the value read {20}. This means that increasingly the output from the `Get` processes should be slowed down as increasing values of `v` are read.

Listing 15-7 shows the way in which multiple copies of the `Get` process can be created, within the same node.

```
10 def numberOfGets = Ask.Int("How many get processes (2..9)?", 2, 9)
11
12 def manyGetNodeIP = "127.0.0.2"
13 def manyGetAddr = new TCPIPNodeAddress(manyGetNodeIP, 3000)
14 Node.getInstance().init (manyGetAddr)
15
16 def comms = NetChannel.net2any()
17 def pList = (0 ..< numberOfGets).collect{
18     i -> new Get ( inChannel: comms, id: i )
19 }
20
21 new PAR ( pList ).run()
```

Listing 15-7 The Creation of Many Get processes Within a Single Node

A shared `net2any` channel can only be accessed by processes on the same node; hence we need to create multiple `Get` processes on the same node. The number of `Get` processes is obtained {10}. The `get` node is then created {12–14}. The input end of the `net` channel, `comms`, is then created as a `net2any()` channel {16}. In this case the numbered version of channel creation has not been used; this means that channels are allocated a number by default starting at 50. A list, `pList`, is then created {17–19} by collecting the required number of `Get` process instances, each with a unique `id` and all accessing the single input channel `comms`. The list of processes is then `run` {21}

```
10 def putNodeIP = "127.0.0.1"
11 def getNodeIP = "127.0.0.2"
12
13 def nodeAddr = new TCPIPNodeAddress(putNodeIP, 3000)
14 Node.getInstance().init (nodeAddr)
15
16 def getNode = new TCPIPNodeAddress(getNodeIP, 3000)
17 def comms = NetChannel.one2net(getNode, 50)
18
19 def pList = [ new Put ( outChannel: comms ) ]
20
21 new PAR ( pList ).run()
```

Listing 15-8 The Node Running the Put Process

Listing 15-8 shows how the node that runs the `Put` process is created. Both node IP addresses are defined {10, 11}, after which the `Put` node instance is created {13, 14}. The `Put` node has no input channels and so the node address of the `Get` node can be created as `getNode` {16}. The `net` output channel `comms` can now be created as a `one2net` channel {17} connected to the default channel numbered 50. The list `pList` is then defined {19} containing a single `Put` process instance, which is then `run` {21}.

On the accompanying web site there is a version of the script used to invoke a single `Get` process, which the interested reader can use to convince themselves that it is not possible to create multiple copies of a single `Get` process accessing a shared `net2any` channel. Effectively, an attempt to create a second `Socket` with the same address is being undertaken and not surprisingly this causes an error.

Execution of the `Put` and many `Get` processes produces an output stream that over a period runs ever slower as the time delay increases in each `Get` process instance. Further inspection of the output shows that the order in which the values are read by the `Get` processes remains unaltered as would be expected and that the values are read by the different `Get` processes in turn.

15.4 Networked Dining Philosophers

As another demonstration of shared networked channels we implement the canteen based version of the dining philosophers discussed previously in Chapter 12. The process definition for the `Canteen`, `Philosophers` and `Kitchen` are taken directly from those presented in Chapter 12. All that has changed is the manner of their invocation.

Inspection of Listing 12-9 will show that there is a shared channel to which the `Philosopher` processes write to the `Canteen` and that there is another by which the `Canteen` writes data back to the `Philosophers`. This means the `Philosopher` processes all have to be executed on the same `Node`. The networked structure of the system is shown in Figure 15-4. The network is represented by the central ellipse. The figure shows the IP-addresses of the nodes, together with their ports and the channel numbers allocated to each channel at each node.

The advertisement features a central graphic with three stylized human figures in blue, surrounded by four interlocking gears and four curved arrows forming a circular path. To the right, the text 'UNLEASHING CHANGE MANAGEMENT' is written in large, bold, blue capital letters. Below this, the dates 'OCTOBER 18 & 19, 2018' and the location 'DE RODE HOED AMSTERDAM' are listed in smaller blue text. At the bottom, there is a silhouette of an Amsterdam cityscape including a windmill, a bridge, and several buildings. In the bottom left corner, the text 'Global Executive Events' is displayed. A hand cursor icon is positioned over a green oval at the bottom right of the ad, which contains the text 'Click on the ad to read more'.

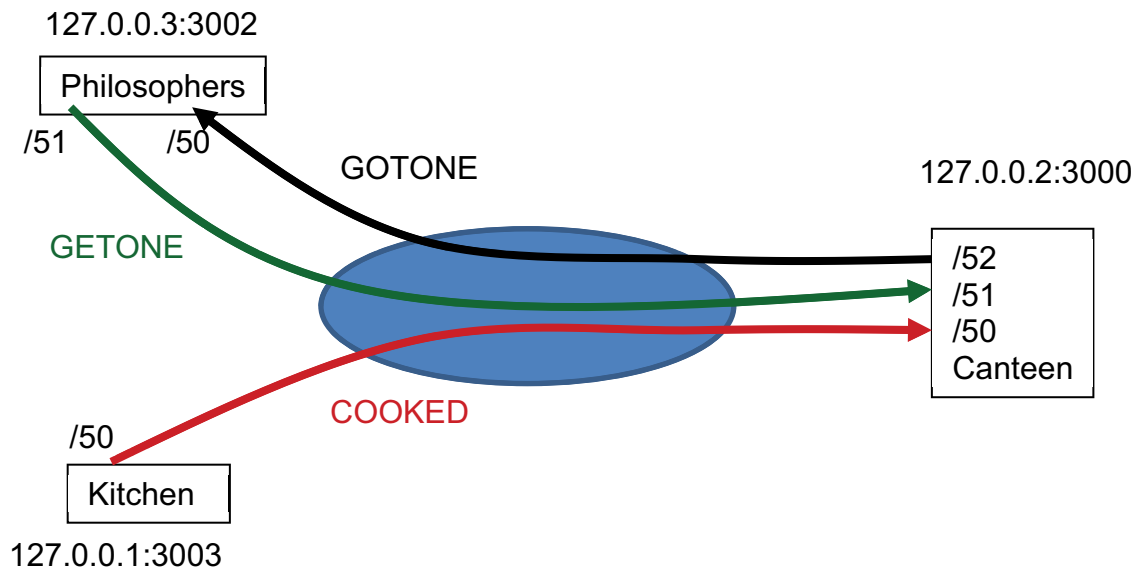


Figure 15-4 The Networked Structure for the Dining Philosophers Canteen Based Solution

The Canteen is run on a node as shown in Listing 15-9. The `import {10}` explicitly imports the process definition from package `c12` reinforcing the fact that we are reusing a process, originally executed on a single processor for execution on a network of processors. The IP address of each node is defined {12–14}

```

10 import c12.canteen.*
11
12 def chefNodeIP = "127.0.0.1"
13 def canteenNodeIP = "127.0.0.2"
14 def philosopherNodeIP = "127.0.0.3"
15
16 def canteenNodeAddr = new TCPIPNodeAddress(canteenNodeIP, 3000)
17 Node.getInstance().init (canteenNodeAddr)
18 def cooked = NetChannel.net2one()
19 println "cooked location = ${cooked.getLocation()}"
20
21 def getOne = NetChannel.net2one()
22 println "getOne location = ${getOne.getLocation()}"
23
24 getOne.read() // signal from the philosophers
25 def philosopherAddr = new TCPIPNodeAddress (philosopherNodeIP, 3002)
26 def gotOne = NetChannel.one2net(philosopherAddr, 50)
27
28 def processList = [
29   new ClockedQueuingServery(service:getOne, deliver:gotOne, supply:cooked)
30 ]
31 new PAR ( processList ).run()

```

Listing 15-9 The Invocation of the Canteen Node

The Canteen node instance is created, after which the input end of the channel from the Chef, called `cooked`, can be created {18}. The location of this channel is printed, confirming that it has the default channel number 50 {19}. The input end of the channel from the Philosophers is created called `getOne` and its location is printed {21, 22}. This shows that, by default, it has been allocated to channel number 51. The Canteen process cannot create the remaining output channel from the Canteen to the Philosophers because we need to be assured that the Philosopher node has been created. The process therefore waits until a signal is received from the Philosopher node {24}. Once the Philosopher node is created, it will send the required signal which allows the Canteen node to make the output connection to the Philosopher node {25, 26}. The `getOne` channel is assumed to be on channel number 50 on the Philosopher node. The Canteen is the one end of all the networked channels and this can be observed in the definition of each of the networked channels {18, 21, 26}. The `ClockedQueuingServer` version of the canteen is invoked {28–30} and run {31}. The Canteen node must be created first, otherwise the other nodes will attempt to connect to input channel ends that do not yet exist. The order in which the Kitchen and Philosopher nodes are created is immaterial.

The `Kitchen` is invoked as shown in Listing 15-10, within the `Kitchen` process, which like all the other nodes will also create a console by which the operation of the system can be observed. The `Kitchen` node instance is created {16, 17}. The node address for the Canteen is created {18} and then an output channel end is created connecting to the `cooked` channel of the Canteen. The channel location is printed for confirmation {20} and then the imported process {10} is run {22, 23}.

```
10 import c12.canteen.*
11
12 def chefNodeIP = "127.0.0.1"
13 def canteenNodeIP = "127.0.0.2"
14 def philosopherNodeIP = "127.0.0.3"
15
16 def TCPIPNodeAddress chefNodeAddr = new TCPIPNodeAddress(chefNodeIP, 3003)
17 Node.getInstance().init (chefNodeAddr)
18 def canteenAddress = new TCPIPNodeAddress(canteenNodeIP, 3000)
19 def cooked = NetChannel.one2net(canteenAddress, 50)
20 println "cooked location = ${cooked.getLocation()}"
21
22 def processList = [ new Kitchen ( supply: cooked) ]
23 new PAR ( processList ).run()
```

Listing 15-10 The Kitchen Node

The collection of `Philosopher` processes is invoked by means of the script shown in Listing 15-11.

```
10 import c12.canteen.*
11
12 def chefNodeIP = "127.0.0.1"
13 def canteenNodeIP = "127.0.0.2"
14 def philosopherNodeIP = "127.0.0.3"
15
16 def philosopherNodeAddr = new TCPIPNodeAddress(philosopherNodeIP, 3002)
17 Node.getInstance().init (philosopherNodeAddr)
18 def gotOne = NetChannel.net2any()
19 println "gotOne location = ${gotOne.getLocation()}"
20
21 def canteenAddress = new TCPIPNodeAddress(canteenNodeIP, 3000)
22 def getOne = NetChannel.any2net(canteenAddress, 51)
23 println "getOne location = ${getOne.getLocation()}"
24
25 getOne.write(0)
26 def philList = ( 0 .. 4 ).collect{
27   i -> return new Philosopher(philosopherId:i, service:getOne, deliver:gotOne)
28 }
29 new PAR ( philList ).run()
```

Listing 15-11 The Node Running the Collection of the Philosopher Processes

The Philosopher node instance is created {16, 17} and then the input channel from the Canteen is created as the channel `gotOne` {18} and its location printed {19}. The channel `gotOne` is shared among all the Philosophers and so is defined to be `net2any`. The Canteen node has to be created before the Philosophers node and so we can create the required output channel connection to the Canteen called `gotOne` {21–23}. It is important to note here that the channel numbers allocated at each node have to be carefully managed so that the correct connections are made. Here for example we know that the input channel in the Canteen node was the second one created and hence will be numbered 51. The Philosopher node can now write a signal on the `gotOne` channel {25}, which allows the Canteen node to complete its creation. The collection of Philosopher processes can now be created and run {26–29}.

15.5 Running the CREW Database in a Network

The simplest way to run the CREW Database example of Chapter 13 (see Figure 13-1) is to execute the Database process on one node and each of the external Read and Write processes on their own node. Each process is allocated to its own node with its own TCP/IP address. The scripts that run each of the required node processes are described in the following sections.

15.5.1 Read Process Node

The script to create a Read process node is shown in Listing 15-12. The first aspect to note is that the script imports the Read process from package `c13` {10} ensuring that the process previously described, that ran in a single node system is exactly the same as that being used in the multi-node version.

```
10 import c13.Read
11
12 def dbIp = "127.0.0.1"
13 def readBase = 100
14 def readerBaseIP = "127.0.0."
15 def readerId = Ask.Int ("Reader process ID (0..4)? ", 0, 4)
16 def readerIndex = readBase + readerId
17 def readerIP = readerBaseIP + readerIndex
18 def readerAddress = new TCPIPNodeAddress(readerIP, 1000)
19 Node.getInstance().init(readerAddress)
20
21 println "Read Process $readerId, $readerIP is creating its Net channels "
22
23 //NetChannelInput
24 def fromDB = NetChannel.numberedNet2One(75) // the net channel from the database
25 println "fromDB location = ${fromDB.getLocation()}"
26
27 //NetChannelOutput
28 def dbAddress = new TCPIPNodeAddress(dbIp, 3000)
29 def toDB = NetChannel.one2net(dbAddress, readerIndex) // the net
    channel to the database
```

```
30 println "toDB location = ${toDB.getLocation()}"
31 toDB.write(0)
32 fromDB.read()
33
34 println "Read Process $readerId has created its Net channels "
35 def consoleChannel = Channel.one2one()
36 def pList = [
37     new Read ( id:readerId, r2db: toDB, db2r: fromDB, toConsole:
        consoleChannel.out() ),
38     new GConsole(toConsole:consoleChannel.in(), frameLabel: "Reader
        $readerId" )
39 ]
40 new PAR (pList).run()
```

Listing 15-12 CREW Database Read Process Node Creation Script

The database IP-address is defined as the variable `dbIp` {12}. The system assumes that all Read processes have a base IP-address that starts from 100 {13}. The user is asked for the identification of the Read process as a number in the range 0 to 4 {15}. Each Read process must be allocated to a node that has a different IP-address. The `readerIP` address is then created from the constants and variable previously defined {17} and this is then used to create the node address for this reader as `readAddress` {18}. The node can then be initialised using that `readAddress` {19}. A message is then printed confirming the creation of the node {21}. The script now continues with the creation of the required communication channels.

The script assumes that the Database node has been created before any of the Read and Write processes are created. First the input channel from the database, `fromDB`, is created {24} as a `numberedNet2One` channel. The input channel is allocated the index 75, which is confirmed by printing out the connection data {25}. The output channel from the Read node to the Database node can now be created. The IP-address of the Database node is created as `dbAddress` {28}, after which the channel `toDB` is created {29}. The channel is created with index number `readerIndex`, which is unique to this node. It assumes that channels with this index number have been created in the Database node. The location of the `toDB` channel is printed to confirm its creation {30}.

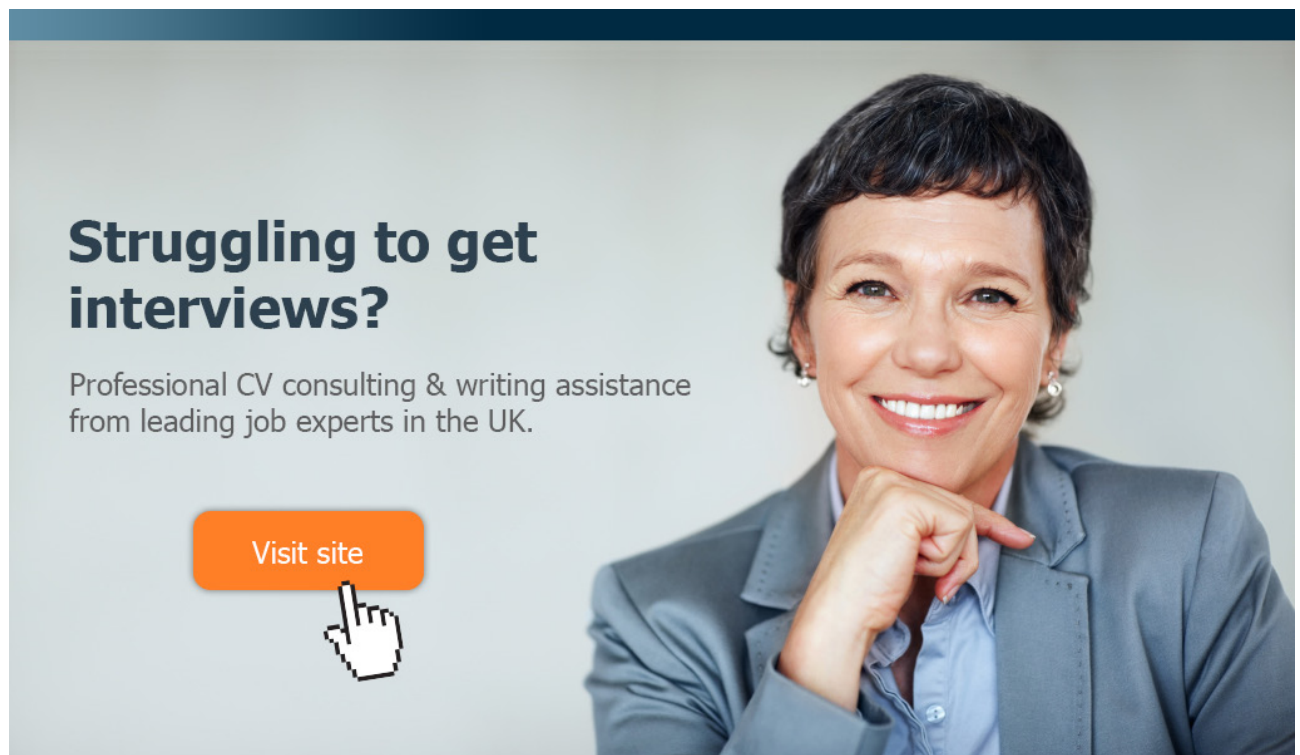
At this point the script causes a signal to be sent to the Database node {31}. Each Read or Write node will send an equivalent signal, so that the database can ensure that all channels have been created before any of the actual node processes are started. Thus the script then reads a signal from the Database node {32}, which will only be sent once all the scripts have reached the same point. A message is then printed confirming this has happened {34}.

The script now creates an internal channel, `consoleChannel`, {35} that will be used to connect the Read process to its associated `GConsole` process. The process list `pList` is created {36–39} and then the processes are `run` {40}.

15.5.2 Write Process Node

The script to create a Write process node is shown in Listing 15-13. This has the same structure as the Read process, previously described.

```
10 import c13.Write
11
12 def dbIp = "127.0.0.1"
13 def writeBase = 200
14 def writerBaseIP = "127.0.0."
15 def writerId = Ask.Int ("Writer process ID (0..4)? ", 0, 4)
16 def writerIndex = writeBase + writerId
17 def writerIP = writerBaseIP + writerIndex
18 def writerAddress = new TCIPNodeAddress(writerIP, 2000)
19 Node.getInstance().init(writerAddress)
20
21 println "Write Process $writerId, $writerIP is creating its Net channels "
22
23 //NetChannelInput
24 def fromDB = NetChannel.numberedNet2One(150) // the net channel from
    the database
25 println "fromDB location = ${fromDB.getLocation()}"
26
27 //NetChannelOutput
```



Struggling to get interviews?

Professional CV consulting & writing assistance from leading job experts in the UK.

Visit site



Take a short-cut to your next job!
Improve your interview success rate by 70%.



TheCVagency
Visit theagency.co.uk for more info.



Click on the ad to read more

```
28 def dbAddress = new TCPIPNodeAddress(dbIp, 3000)
29 def toDB = NetChannel.one2net(dbAddress, writerIndex) // the net
    channel to the database
30 println "toDB location = ${toDB.getLocation()}"
31 toDB.write(0)
32 fromDB.read()
33
34 println "Write Process $writerId has created its Net channels "
35 def consoleChannel = Channel.one2one()
36
37 def pList = [
38     new Write ( id:writerId, w2db:toDB, db2w:fromDB, toConsole:
        consoleChannel.out() ),
39     new GConsole(toConsole:consoleChannel.in(), frameLabel: "Writer $writerId" )
40 ]
41 new PAR (pList).run()
```

Listing 15 – 13 CREW Database Write Process Node Creation Script

The IP-address base for Write nodes is 200 {13} and the input channels to the node a numbered at 150 {24}.

15.5.3 Database Process Node

The script to create the Database process node is shown in Listing 15-14. Lines {12–14} create a node instance for the Database node. The number of Reader and Writer nodes is then obtained {16, 17}.

```
10 import c13.DataBase
11
12 def dbIp = "127.0.0.1"
13 def dbAddress = new TCPIPNodeAddress(dbIp, 3000)
14 Node.getInstance().init(dbAddress)
15
16 int nReaders = Ask.Int ( "Number of Readers ? ", 1, 5)
17 int nWriters = Ask.Int ( "Number of Writers ? ", 1, 5)
18
19 def readerAddresses = []
20 def writerAddresses = []
21 def toDB = new ChannelInputList()
22 def fromDB = new ChannelOutputList()
23
24 println "Creating reader network channels"
25 def readBase = 100
26 def readerBaseIP = "127.0.0."
27
28 for ( readerId in 0 ..< nReaders ) {
29     def readerIndex = readBase + readerId
30     def readerIP = readerBaseIP + readerIndex
```

```
31 readerAddresses << new TCPIPNodeAddress(readerIP, 1000)
32 toDB.append ( NetChannel.numberedNet2One(readerIndex) )
33 println "Reader: $readerId, $readerIndex, $readerIP - " +
34     "toDB location = ${toDB[readerId].getLocation()}"
35 }
36 println "Creating writer network channels"
37 def writeBase = 200
38 def writerBaseIP = "127.0.0."
39
40 for ( writerId in 0 ..< nWriters ) {
41     def writerIndex = writeBase + writerId
42     def writerIP = writerBaseIP + writerIndex
43     writerAddresses << new TCPIPNodeAddress(writerIP, 2000)
44     toDB.append ( NetChannel.numberedNet2One(writerIndex) )
45     println "Writer: $writerId, $writerIndex, $writerIP - " +
46         " toDB location = ${toDB[writerId+nReaders].getLocation()}"
47 }
48
49 for ( r in 0 ..< nReaders){
50     toDB[r].read()
51     fromDB.append ( NetChannel.one2net ( readerAddresses[r], 75) )
52     println "Reader $r fromDB location = ${fromDB[r].getLocation()}"
53 }
54
```



e-learning for kids

- The number 1 MOOC for Primary Education
- Free Digital Learning for Children 5-12
- 15 Million Children Reached

About e-Learning for Kids Established in 2004, e-Learning for Kids is a global nonprofit foundation dedicated to fun and free learning on the Internet for children ages 5 - 12 with courses in math, science, language arts, computers, health and environmental skills. Since 2005, more than 15 million children in over 190 countries have benefitted from eLessons provided by EFK! An all-volunteer staff consists of education and e-learning experts and business professionals from around the world committed to making difference. eLearning for Kids is actively seeking funding, volunteers, sponsors and courseware developers; get involved! For more information, please visit www.e-learningforkids.org.

```
55 for ( w in 0..< nWriters){
56   toDB[w + nReaders].read()
57   fromDB.append ( NetChannel.one2net ( writerAddresses[w], 150) )
58   println "Writer $w fromDB location = ${fromDB[w + nReaders].
           getLocation()}"
59 }
60
61 for ( c in 0 ..< (nReaders + nWriters)){
62   fromDB[c].write(0)
63 }
64 println "DBM: Creating database process list"
65
66 def pList = [ new DataBase ( inChannels: toDB,
67                             outChannels: fromDB,
68                             readers: nReaders,
69                             writers: nWriters ) ]
70 println "DBM: Running Database"
71
72 new PAR (pList).run()
```

Listing 15-14 CREW Database Node Creation Script

The lists `readerAddresses` and `writerAddresses` {19, 20} are used to hold the node addresses of each of the other nodes. The `ChannelInputList` `toDB` {21} is used to hold all the input channels from the other nodes. Similarly, the `ChannelOutputList` `fromDB` {22} is used to hold all the output channels to the other nodes. The coding shown from lines {24–35} is used to create the node address of each of the Read nodes. The `readerIP` is created {30} and then used to create a node address that is appended to `readerAddresses` {31}. A net input channel is then created, at the required index number and appended to the `toDB` channel input list {44}. The channel index number is the same as that used in the creation of the Read node. This coding is then repeated for the equivalent channels for each Write node {36–47}.

The next phase causes the signal to be read from each Read node {50}, after which the output channel to that node can be created and appended to the `fromDB` channel output list {51}. The index number used in the channel creation is the same as that specified when the channel was created in the Read process. The output channel cannot be created until the script knows that the Read process at the input end of the channel exists, hence the requirement for the signal on the `toDB` channel. Lines {55–59} repeat the coding for each Write node. Once all the input signals have been read and the corresponding output channel created, the matching signal telling the other nodes they can continue with their creation can be sent {61–64}.

The Database, imported from package `c13` {10} can now be created and run {66–72}.

15.6 Summary

In this chapter we have seen how to create networks of nodes executing on a TCP/IP based network. For ease of implementation, the nodes have been created using the 'loop-back' IP-address 127.0.0.n. The scripts can be easily modified to run on a real network simply by changing the base IP-address in each script. Each of the examples has been based on one that ran on a single processor and demonstrates that transforming an application merely requires the scripts to invoke the nodes but no change to any of the application's processes.

FACTCARDS

Are you working in academia, research or science? And have you ever thought about working and moving to the Netherlands?

Arriving 33

Living 50

Studying 51

Working 101

Research 50

Factcards.nl offers all the **information** that you need if you wish to proceed your **career** in the **Netherlands**.

The information is ordered in the categories arriving, living, studying, working and research in the Netherlands and it is freely and easily accessible from your smartphone or desktop.

VISIT FACTCARDS.NL



16 Dynamic Process Networks: A Print Server

Dyanmic process networks require a few simple concepts which are explained:

- creation of net channels at run-time
- communicating net channel locations over network channels
- dynamic creation and destruction of network topologies on as needed basis

A print server is probably the simplest service used by users of a networked service. It provides a means whereby a user can send a file for printing using a printer shared among the users of the network. In this implementation a print service will be constructed that accepts print lines, a line at a time, from a user. The print service will accept print lines from a number of users, in parallel, up to some limit set when the print server is installed. Once the user has sent all the lines of text to be printed; the print server will then output those lines in a single printed output. The printed output will be preceded by a job number that can be recognised by the user of the service. The user of the service will be informed both when their job has been accepted and when it has completed. The user will be unaware that the print service is dealing with other user requests. The print service should run in the background and always be ready to accept requests from a user, that is, the user processes should start asynchronously with the print service process. The order in which the respective processes start should have no bearing on the operation of the system, apart from the fact that the print service must commence before any of the user processes.

From the foregoing it is obvious that users need to request that their lines of output are sent to the print service and subsequently on completion of their output the user needs to indicate that the lines of text can be printed. To this end the print service provides two named channels by which the user can request and subsequently release their use of the print service.

In addition, if the print service is going to manage print operations from more than one user in parallel then some means of telling the user which of the services to use will be required. The user also needs to be able to send lines to be printed to the print service. These connections will change with each print job and thus the corresponding network channels will be created dynamically.

The architecture of the system is shown in Figure 16-1. The `PrintSpooler` process provides the print service using two named `net2one` channels called `request` and `release`. The network connections are indicated by the dashed lines, one for each channel. Each `PrintUser` process can dynamically connect to the `request` and `release` channels by defining them as `any2net` when their node is created. In order to avoid multiple communications on the `request` and `release` channels only one communication will be permitted on each channel for each print job.

The diagram shows the state when the `PrintSpooler` is willing to accept print lines from up to two `PrintUser` processes in parallel. These have been given names for clarity but in reality are anonymous. The `useChannel` is used by `PrintSpooler` to tell the `PrintUser` the location of the `printChannel` it is to use to send the lines to the `PrintSpooler`. The `printChannel` is used to send the lines to be printed to the `PrintSpooler`.

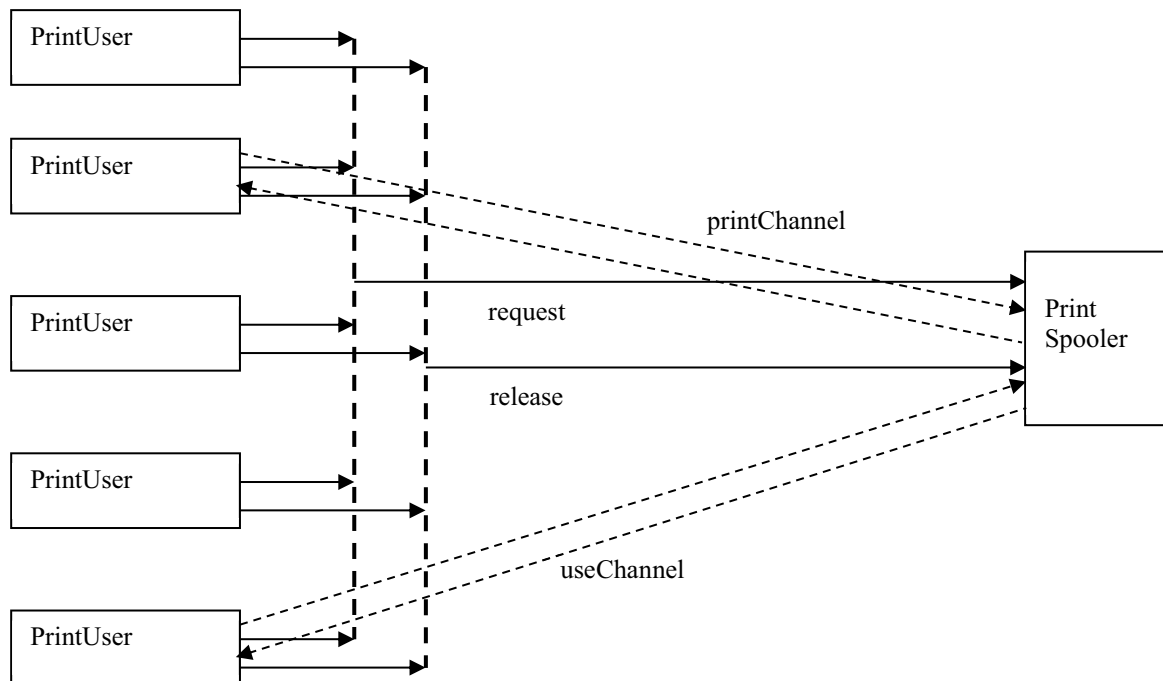


Figure 16-1 The Print Service Architecture

16.1 Print Spooler Data Objects

Two data objects are required, which both are used to transfer information from `PrintUser` processes to the `PrintSpooler` process. These objects must implement the `Serializable` interface as the objects are to be communicated over the network and thus will be transferred from one address space to another. The first, `PrintJob`, shown in Listing 16-1 is used to make an initial request for service. It comprises two properties, the identity of the user {2} and the net channel location to be used by `PrintSpooler` as the `useChannel` {3}. The manner of its creation and its use will be described later.

```
10 class PrintJob implements Serializable{
11
12     def int userId
13     def NetChannelLocation useLocation
14 }
```

Listing 16-1 The PrintJob Data Object

The other data object, `PrintLine`, shown in Listing 16-2, is used to transfer the lines to be printed from a `PrintUser` process to the `PrintSpooler` process. It is written by the `PrintUser` to a `printChannel`. The property `printKey {6}` indicates to which, of the possibly several internal concurrent spoolers within `PrintSpooler`, this line of text is intended. The `String line {7}` is the text to be added to the output.

```
10 class Printline implements Serializable {
11
12     def int printKey
13     def String line
14 }
```

Listing 16-2 The PrintLine Data Object

Brain power

By 2020, wind could provide one-tenth of our planet's electricity needs. Already today, SKF's innovative know-how is crucial to running a large proportion of the world's wind turbines.

Up to 25 % of the generating costs relate to maintenance. These can be reduced dramatically thanks to our systems for on-line condition monitoring and automatic lubrication. We help make it more economical to create cleaner, cheaper energy out of thin air.

By sharing our experience, expertise, and creativity, industries can boost performance beyond expectations. Therefore we need the best employees who can meet this challenge!

The Power of Knowledge Engineering

Plug into The Power of Knowledge Engineering.
Visit us at www.skf.com/knowledge

SKF

16.2 The PrintUser Process

Listing 16-3 shows the coding of the `PrintUser` process. This process has three properties; `printerRequest` {12}, the network channel used to make requests to the `PrintSpooler`, `printerRelease` {13}, the network channel used to release the `PrintSpooler` at the end of printing and the identity of the user, `userId`{14}.

```
10 class PrintUser implements CProcess {
11
12     def ChannelOutput printerRequest
13     def ChannelOutput printerRelease
14     def int userId
15
16     void run() {
17         def printList = [ "line 1 for user " + userId,
18                         "line 2 for user " + userId,
19                         "last line for user " + userId
20         ]
21         def useChannel = NetChannel.net2one()
22         printerRequest.write(new PrintJob ( userId: userId,
23                                           useLocation: useChannel.
24                                           getLocation() ) )
25
26         def printChannelLocation = useChannel.read()
27         def useKey = useChannel.read()
28         println "Print for user ${userId} accepted using Spooler $useKey"
29         def printerChannel = NetChannel.one2net ( printChannelLocation)
30         printList.each { printerChannel.write (new Printline ( printKey:
31         useKey, line: it) )}
32         printerRelease.write ( useKey )
33         println "Print for user ${userId} completed"
34     }
35 }
```

Listing 16-3 The `PrintUser` Process Definition

The List `printList` {17–20} holds the lines of text that are to be printed. Each user prints the same output, only differentiated by their `userId`.

A `net2one NetChannel` is created at the node with a default index number {21}. This input network channel is assigned to variable `useChannel`. A `PrintJob` object is constructed from `userId` and the location of `useChannel` {22–23}. The location of a network channel is obtained by calling the method `getChannelLocation()` {23}, which returns the IP address, port and unique channel number of the channel. The `PrintJob` object is then written to the `printerRequest` network channel. This write {22} may be delayed if the `PrintSpooler` is already dealing with the maximum number of print requests, but the user will be unaware of this, in the sense that they do not have to undertake any additional processing. This is also the first part of a client behaviour with its corresponding server response being the read of a `printerChannelLocation` on the `useChannel` {24}. The `useChannel` is also used to read `useKey` from the `PrintSpooler` {25}. The `useKey` is the means by which the `PrintUser` process identifies which of the concurrent spoolers maintained by `PrintSpooler` it is to use. A message is then printed indicating that the request has been accepted {26}.

The process now creates the channel `printerChannel` on which it is to send lines to the `PrintSpooler`. The location of this channel has been read as `printerChannelLocation`. A network channel can be created from this location by a call to the `NetChannel` factory to create a `one2net` channel with the location specified in `printerChannelLocation` {27}. This channel is then used to write each of the `printList` elements to `printerChannel` using a constructed `PrintLine` object for each element {28}. The Groovy operator `each` iterates through the elements of a `List` and the associated closure can refer to the specific element using the `it` keyword.

Once all the elements of `PrintList` have been written to the `PrintSpooler` it can be released and this is simply achieved by writing the `useKey` to the `printerRelease` channel {29}, after which a message can be printed {30}.

16.3 The PrintSpooler Process

Listings 16-4 and 16-5 show the coding of the `PrintSpooler` process. Properties `printerRequest` {12} and `printerRelease` {13} are the request and release networked channels `PrintUsers` use to indicate their desire to access the `PrintSpooler`. The `PrintSpooler` will create two concurrent spoolers by default {14} but this can be changed when the process is invoked.

```
10 class PrintSpooler implements CProcess {
11
12     def ChannelInput printerRequest
13     def ChannelInput printerRelease
14     def int spoolers = 2
15
16     void run() {
17         def timer = new CTimer()
18         def spooling = 0
19         def spoolChannels = []
```

```
20     def spoolChannelLocations = [:]
21     def unusedSpoolers = []
22     def preCon = new boolean[spoolers + 2 ]
23     def printMap = [:]
24     def jobMap = [:]
25     preCon[0] = true
26     0.upto(spoolers - 1) { i -> def c = NetChannel.net2one()
27                               spoolChannels << c
28                               spoolChannelLocations.put(i,
29                                                         c.getLocation() )
30                               unusedSpoolers << i
31                               preCon[i+2] = false
32 }
33     def altChans = [ printerRelease, printerRequest ]
34     altChans = altChans + spoolChannels
35     def psAlt = new ALT ( altChans )
```

Listing 16-4 PrintSpooler Initialisation

Cynthia | AXA Graduate

**AXA Global
Graduate Program**

Find out more and apply

redefining / standards AXA



The variable `spooling` {18} is used to count how many users are currently sending output to the `PrintSpooler` and initially this is none. `SpoolChannels` will hold a `List` of the channels {19} that will become the `printChannels` of Figure 16-1. These will be created at the outset rather than recreating them on each occasion individually for each print request. The `Map` `spoolChannelLocations` {20} will hold the `NetChannelLocation` of each spool channel used by each spooler; its key is the index of the spooler. The `List` `unusedSpoolers` {21} holds the index of the spoolers that are currently not being used. The `Map` `printMap` {23} is used to hold the lines for each concurrent print request and its key is the associated spooler index. The `Map` `jobMap` {24} is used to maintain the connection between spooler index and user requesting the print job.

The process is going to use an alternative to determine which input channels it will receive input from. These are going to be further managed using a precondition array and this is defined as `preCon` {22}. There is an input channel for each spooler plus the two named request and release channels, giving the number of elements in the array.

Lines {26–31} initialise these data structures as follows by iterating over the number of spoolers {14}, with `i` indexed from 0. A `NetChannelInput` `c` is created {26} and this will be subsequently used as the `printChannel`. It is then appended to `spoolChannels` {27}. Its net channel location is then put in the `i`'th element of `spoolChannelLocations` {28}. The value of `i` is then appended to the `List` of `unusedSpoolers` {29}. Finally, `preCon[i+2]` {30} is set `false` indicating that the processes cannot accept input on any of its spool channels. The `List` `altChans` {32} is initialised with the `printerRelease` and `printerRequest` channels to which is added the `List` of `spoolChannels` {33}. The process is always willing to accept an input on its release channel and so `preCon[0]` is set `true` {25}. The reason for the offset of `i+2` {30} is to take account of the fact that the printer release and request channels appear first in the alternative `psAlt` {34}.

Listing 16-5 shows the main body loop of the `PrintSpooler` process. At the start of each iteration a test is made to determine the state of `preCon[1]` {36} which ensures that a request for service will only be accepted if at least one of the available spoolers is free. The `index` of the enabled alternative is selected {37} and used to determine which case is processed.

```
35     while (true) {
36         preCon[1] = (spooling < spoolers)
37         def index = psAlt.select(preCon)
38         switch (index) {
39             case 0:
40                 //user releasing a print channel
41                 def usedKey = printerRelease.read()
42                 unusedSpoolers.add(usedKey)
43                 preCon[usedKey + 2] = false
44                 spooling = spooling - 1
```



```
45         // now print the spooled lines
46         def lines = printMap.get(usedKey)
47         print "\n\nOutputFor User ${jobMap.get(usedKey)}\n"
48         println "Produced using spooler $usedKey \n\n"
49         lines.each{ println "$it" }
50         println "\n\n=====\n\n"
51         printMap.remove(usedKey)
52         jobMap.remove(usedKey)
53         break
54     case 1:
55         // user requested a print channel
56         def job = printerRequest.read()
57         def useChannelLocation = job.useLocation
58         def userId = job.userId
59         def useChannel = NetChannel.one2net(useChannelLocation)
60         spooling = spooling + 1
61         def useKey = unusedSpoolers.pop()
62         preCon[useKey+2] = true
63         printMap[useKey] = [] // initialise the printlist for this user
64         jobMap[useKey] = userId
65         useChannel.write(spoolChannelLocations.get(useKey) )
66         useChannel.write( useKey )
67         break
68     default :
69         // prntline being received from a user
70         def pLine = spoolChannels[ index - 2].read()
71         printMap[pLine.printKey] << pLine.line
72         timer.sleep(5000)
73     } //switch
74 } //while
75 } // run
76 } // class
```

Listing 16-5 The PrintSpooler Process Loop

Case 0 represents an input on the release channel, which means that the lines can be printed and the associated spooler released for another user. The input `usedKey` from the `printerRelease` channel identifies the spooler allocated to the user {41}. This spooler can then be added to the List of `unusedSpoolers` {42}. The process is now unwilling to accept any more inputs from the user and thus sets the corresponding `preCon` element `false` {43}. Similarly, the number of `spooling` spoolers can be decremented {44}. The List `lines` comprises the Map entry for `usedKey` {46} obtained using the `Map.get()` method. The printed output banner lines can now be printed {47–48}, after which the `lines` can themselves be printed {49} followed by a terminating banner {50}. The Map entry for `usedKey` can now be removed {51}. Similarly the entry relating job and user from `jobMap` can be removed {52}. This code sequence recovers the printing resources, prints the lines and ensures that the associated data structure have been updated accordingly.

Case 1 pertains to a request for printing by a user process and will only be accepted if at least one of the spoolers is available. The print job details are read from the `printerRequest` channel {56} and the `PrintJob` properties are extracted into variables `useChannelLocation` {57} and `userId` {58}. The channel by which the `PrintSpooler` process sends data to the `PrintUser` process, `useChannel`, is created by taking the value of `useChannelLocation` {59} as a parameter to a call of `NetChannel.one2net()`. In the `PrintSpooler` process we are creating the output end of the channel to be connected to the input end that was created in the `PrintUser` process. The number of spoolers that are spooling can be incremented {60} and the index of an unused spooler can be pop'ed from the List of `unusedSpoolers` {61} and assigned to `useKey`. The pre-condition element of the array `preCon` associated with this spooler can be set true because the process is now willing to accept inputs on the related spool channel {62}. A Map entry that uses `useKey` as its key can be initialised to an empty List {63}. An entry can be placed in `jobMap` that relates `useKey` to the `userId` of the job being processed by this spooler {64}.

The `PrintSpooler` process acts as a server to the `PrintUser` processes and a request for service expects a response in finite time. The generated response is the location of the spool channel that the `PrintUser` process is to use for writing lines of text to the `PrintSpooler` {65}. Secondly, the value of `useKey` which is used by the `PrintUser` process to identify which spooler is being used to form the lines of text to be output {66}.

TURN TO THE EXPERTS FOR SUBSCRIPTION CONSULTANCY

Subscribe is one of the leading companies in Europe when it comes to innovation and business development within subscription businesses.

We innovate new subscription business models or improve existing ones. We do business reviews of existing subscription businesses and we develop acquisition and retention strategies.

Learn more at [linkedin.com/company/subscribe](https://www.linkedin.com/company/subscribe) or contact
Managing Director Morten Suhr Hansen at mha@subscribe.dk

SUBSCR✓**BE** - to the future



The default case {68} is perhaps the simplest and reflects the input of a line of text from a `PrintUser` process to the `PrintSpooler`. The line of text is read into `pLine` from the element of `spoolChannels` indexed by `index - 2` {70}. The value of `index` is obtained from the `select` method call on the alternative `psAlt` {37}. However, `psAlt` precedes the `spoolChannels` with the `printerRelease` and `printerRequest` channels and thus it is necessary to subtract 2 from the `index` value to access the correct element of `spoolChannels`. The variable `pLine` is of type `PrintLine` and the properties `printKey` and `line` are used to add the line to the `printMap` entry for the `printKey` {71}. In this manner each of the active `PrintUser` processes can send lines to the `PrintSpooler` adding lines to the `List` contained in the `printMap`. All that is required by a `PrintUser` process is the key of the `printMap` used to add lines to the `List` of lines. Thus, each spooler is in fact just represented by an entry in the `printMap` structure. Once a line has been processed a delay is introduced {72} so that it is easier for exploration of the system behaviour enabling the invoking of more `PrintUser` easily.

16.4 Invoking The PrintSpooler Node

Listing 16-6 shows the script used to invoke the `PrintSpooler` process. The initial interaction {10} determines the number of spoolers maintained by the `PrintSpooler` and thus the maximum number of concurrent spoolers active at any time. Next the `PrintSpooler` node is created as the IP-address '127.0.0.1' {12–14}. The net request and release channels are created using the default channel index numbers, with `pRequest` allocated to index 50 {15} and `pRelease` allocated to index 51 {16}. The locations are then printed to confirm their creation {17–18}. The process can then be invoked {20–24}. The `PrintSpooler` process must be invoked before any of the `PrintUser` processes are invoked.

```
10 def spoolers = Ask.Int ("Number of spoolers ? ", 1, 9)
11
12 def printSpoolerIP = "127.0.0.1"
13 def psAddress = new TCPIPNodeAddress(printSpoolerIP, 2000)
14 Node.getInstance().init(psAddress)
15 def pRequest = NetChannel.net2one() // cn = 50
16 def pRelease = NetChannel.net2one() // cn = 51
17 println "pRequest location = ${pRequest.getLocation()}"
18 println "pRelease location = ${pRelease.getLocation()}"
19
20 new PAR ( [ new PrintSpooler ( printerRequest: pRequest,
21                               printerRelease: pRelease,
22                               spoolers : spoolers
23                               )
24           ] ).run()
```

Listing 16-6 The Script to Invoke the `PrintSpooler` Process

16.5 Invoking A PrintUser Node

Listing 16-7 shows the script that invokes a `PrintUser` node. Any number of `PrintUser` nodes can be created. All ‘printed’ output appears in the `PrintSpooler` console window.

The IP-address of the `PrintSpooler` is defined as `printSpoolerIP {10}`. The IP-address of the `PrintUser` is created as `printUserIP {11–13}`. The print user node is then created {15–16} and then the node address of the `PrintSpooler` {17}. The net channel connections to the `PrintSpooler` can now be created using the default allocated channel index numbers {19–20}. These net output channels are created as `any2net` so that there can be many processes that write to the channels. These details are printed {22–23} after which the `PrintUser` process is executed. {25–29}.

```
10 def printSpoolerIP = "127.0.0.1"
11 def printUserIPmask = "127.0.0."
12 def user = Ask.Int ("User Number (2 to 254) ? ", 2, 255)
13 def printUserIP = printUserIPmask + user
14
15 def printUserAddr = new TCPIPNodeAddress(printUserIP, 3000)
16 Node.getInstance().init(printUserAddr)
17 def printSpoolerAddr = new TCPIPNodeAddress(printSpoolerIP, 2000)
18
19 def pRequest = NetChannel.any2net (printSpoolerAddr, 50)
20 def pRelease = NetChannel.any2net (printSpoolerAddr, 51)
21
22 println "pRequest location = ${pRequest.getLocation()}"
23 println "pRelease location = ${pRelease.getLocation()}"
24
25 new PAR ( [ new PrintUser ( printerRequest: pRequest,
26                           printerRelease: pRelease,
27                           userId : user
28 )
29 ] ).run()
```

Listing 16-7 The Script to Invoke a `PrintUser` Process.

16.6 Summary

This chapter has shown how net channel locations can be transferred from one process to another so that communication links between the processes can be created directly at run time. This means that dynamic process architectures can be created.

17 More Testing: Non-terminating Processes

This chapter explores the testing of non-terminating process networks by:

- defining a generic architecture
- separating the test network from the network under test using a TCP/IP network
- ensuring the test network terminates to enable assertion testing
- requiring no change to the network under test

Chapter 6 showed it is possible to use the `GroovyTestCase` capability to test networks of processes, provided each of the processes in the network terminates. Most of the processes used in this book do not terminate and so a means of testing such non-terminating process networks has to be developed.



Losing track of your leads?
Bookboon leads the way
Get help to increase the lead generation on your own website. Ask the experts.

bookboon.com

Interested in how we can help you?
email ban@bookboon.com 



First, however, we need to reflect on the operation of `PAR`. A `PAR` only terminates when all the process in the list of processes passed to it terminate. Thus, if only one of the processes does not terminate then a `PAR` will never terminate. However, if the assertion testing commonly used in `JUnit` and `GroovyTestCase` is to be undertaken then at least some of the test environment has to terminate. Figure 17-1 shows a generic architecture that allows a process network under test (PNUT) to run without terminating, while the Test-Network does terminate, which then allows the assertion testing to take place in the normal manner (Kerridge, 2007).

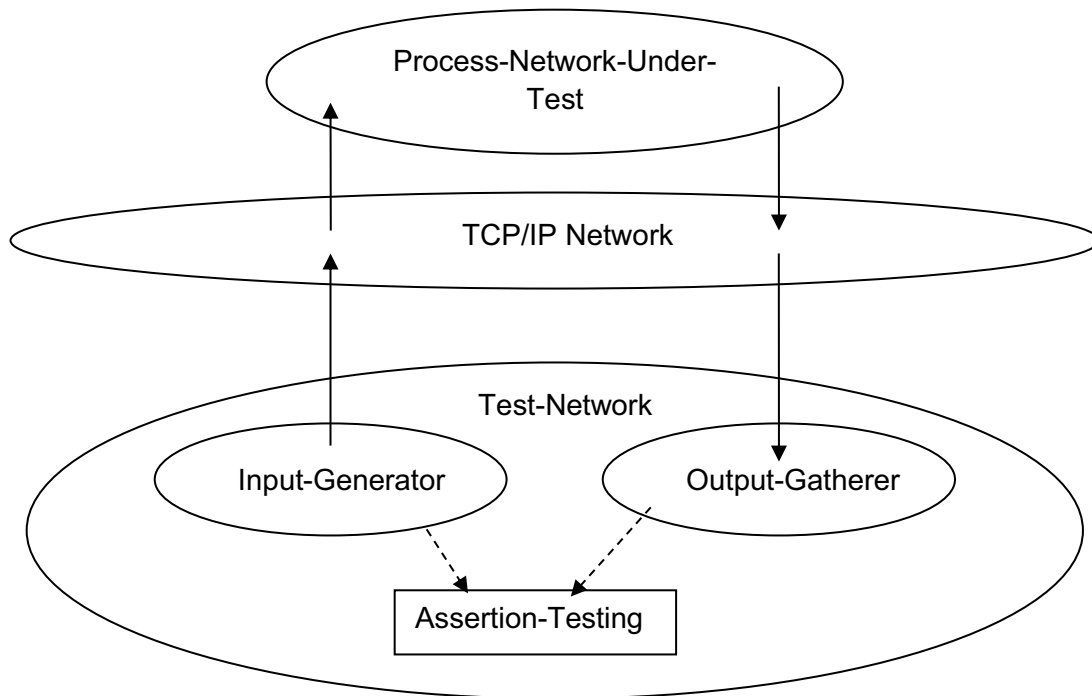


Figure 17-1 Generic Testing Architecture

The separation of the PNUT from the Test-Network by means of a TCP/IP communications network means that the two process networks run independently of each other and it does not matter if the PNUT does not terminate, provided the Test-Network does. We can assume that the PNUT requires input and also that it outputs results in some format. This data is communicated by means of the network channels shown. Both the Input-Generator and Output-Gatherer processes must run as a `PAR` within the process Test-Network, they then terminate; after which their internal data structures can be tested within Assertion-Testing. This demonstrates the generic nature of the architecture in that the only parts that have to be specifically written are the processes that implement the Input-Generator and Output-Gatherer respectively. The architecture will now be demonstrated using the Scaling Device example described previously in Chapter 5. The Scaling Device takes a stream of input numbers and outputs an equivalent stream of scaled numbers, while monitoring the operation of a Scale process by modifying the applied scaling factor.

17.1 The Test-Network

The class `RunTestPart`, shown in Listing 17-1, implements the Test-Network {10} and simply extends the class `GroovyTestCase`. The method `testSomething` {12} creates the Test-Network as a process running in a node on a network. The node is initialized in the normal manner within the JCSP framework {13–17}. Two `NetChannels`, `ordinaryOutput` {19} and `scaledInput` {21} are defined and their locations printed.

The test mechanism assumes that the PNUT node is created before the `RunTestPart` node. To ensure this happens the `RunTestPart` process writes a signal to the PNUT {23}. Once this signal has been read by the PNUT, the processes required by `RunTestPart` can be created and invoked.

The processes are created {25, 26} and then invoked {28, 30}. Once the `PAR` has terminated, the properties `generatedList`, `collectedList` and `scaledList` can be obtained from the processes {32–34} using the Groovy dot notation for accessing class properties. In this case we know that the original generated set of values should equal the `unscaled` output from the collector and this is tested in an assertion {35}. In this case we also know that each modified output from the PNUT should be greater than or equal to the corresponding input value. This is implemented by a method contained in a package `TestUtilities` called `list1GEList2`, which is used in a second assertion {36}.

```
10 class RunTestPart extends GroovyTestCase {
11
12     void testSomething() {
13         def testPartIP = "127.0.0.1"
14         def deviceIP = "127.0.0.2"
15         def testPartAddr = new TCPIPNodeAddress(testPartIP, 3000)
16         def deviceAddr = new TCPIPNodeAddress(deviceIP, 3000)
17         Node.getInstance().init(testPartAddr)
18
19         def ordinaryOutput = NetChannel.one2net(deviceAddr, 50)
20         println "ordinaryOutput location = ${ordinaryOutput.getLocation()}"
21         def scaledInput = NetChannel.net2one()
22         println "scaledInput location = ${scaledInput.getLocation()}"
23         ordinaryOutput.write(1)
24
25         def collector = new CollectNumbers ( inChannel: scaledInput)
26         def generator = new GenerateNumbers (outChannel: ordinaryOutput)
27
28         def testList = [ collector, generator]
29
30         new PAR(testList).run()
31
32         def original = generator.generatedList
33         def unscaled = collector.collectedList
```

```
34     def scaled = collector.scaledList
35     assertTrue (original == unscaled)
36     assertTrue (TestUtilities.list1GEList2(scaled, original))
37   }
38 }
```

Listing 17-1 The Extended GroovyTestCase Class to Run The Test Network

The benefit of this approach is that we are guaranteed that the Test-Network will terminate, provided the `CollectNumbers` and `GenerateNumbers` processes terminate and thus values derived from these processes can be tested in assertions. The fact that the PNUT continues running is made disjoint by the use of the network. This could not be achieved if all the processes were run in a single JVM as the assertions could not be tested because the `PAR` would never terminate. The process network comprising the PNUT and the Test-Network can be run on a single processor with each running in a separate JVM. `RunTestPart` will write its output to a console window indicating whether or not the test has passed. The console window associated with PNUT will continue to produce any outputs associated with the network being tested.



"I studied English for 16 years but...
...I finally learned to speak it in just six lessons"

Jane, Chinese architect

ENGLISH OUT THERE

Click to hear me talking before and after my unique course download

17.1.1 The Generate Numbers Process

Listing 17-2 shows the coding of the `GenerateNumbers` process. Recall from Chapter 5 that the Scaling Device expects to receive numbers at regular intervals, which it then processes. The property `delay` {12} is used to specify the time between the generation of an output of numbers to the PNUT. The length of the generated sequence is specified in `iterations` {13}. The channel `outChannel` {14} is used to communicate the generated numbers to the PNUT. The list `generatedList` {15} is used to hold the sequence of generated numbers for subsequent testing in an assertion in the process `RunTestPart`.

```
10 class GenerateNumbers implements CProcess{
11
12     def delay = 1000
13     def iterations = 20
14     def ChannelOutput outChannel
15     def generatedList = []
16
17     void run() {
18         println "Numbers started"
19         def timer = new CTimer()
20         for (i in 1 .. iterations) {
21             outChannel.write(i)
22             generatedList << i
23             timer.sleep(delay)
24         }
25         println "Numbers finished"
26     }
27 }
```

Listing 17-2 The `GenerateNumbers` Process

The `run` method {17} outputs a start message {18} and then defines a `timer` {19}. Each number is then generated using a `for` loop {20}, limited by the value of `iterations`. The next number in sequence is output {21} and then appended (<<) to `generatedList` {22}. The process then sleeps for the defined `delay` period {23}. Finally, a finished message is output {25}.

17.1.2 The Collect Numbers Process

Listing 17-3 shows the `CollectNumbers` process. The `inChannel` {12} is used to input data from the PNUT. The output from the Scaling Device is in the form of objects comprising two properties; the original value and the scaled value (See 5.1.3). The original, unmodified values are appended {21} to the property `collectedList` {13} and the scaled values are appended {22} to the property `scaledList` {14}. The number of `iterations` {15} is required to ensure that the process terminates after it has read the expected number of outputs from the PNUT. The `run` method simply iterates over the expected outputs, inputting `ScaledData` from the PNUT {20} and placing the data into the respective lists {21, 22}. The method also indicates, by console messages, when the process starts {18} and finishes {24}.

```
10 class CollectNumbers implements CSProcess {
11
12     def ChannelInput inChannel
13     def collectedList = []
14     def scaledList = []
15     def iterations = 20
16
17     void run() {
18         println "Collector Started"
19         for ( i in 1 .. iterations) {
20             def result = (ScaledData) inChannel.read()
21             collectedList << result.original
22             scaledList << result.scaled
23         }
24         println "Collector Finished"
25     }
26 }
```

Listing 17-3 The CollectNumbers Process

17.2 The Process Network Under Test

The Process Network Under Test (PNUT) is the `ScalingDevice` described in Chapter 5 and can be represented by the `CSProcess` shown in Listing 17-4.

```
10 import c05.*
11 class ScalingDevice implements CSProcess {
12
13     def ChannelInput inChannel
14     def ChannelOutput outChannel
15
16     void run() {
17         println "scaling device started"
18         def oldScale = Channel.one2one()
19         def newScale = Channel.one2one()
20         def pause = Channel.one2one()
21
22         def scaler = new Scale ( inChannel: inChannel,
23                                 outChannel: outChannel,
24                                 factor: oldScale.out(),
25                                 suspend: pause.in(),
26                                 injector: newScale.in(),
27                                 multiplier: 2,
28                                 scaling: 2 )
```

```
29
30     def control = new Controller ( testInterval: 7000,
31                                   computeInterval: 700,
32                                   addition: 1,
33                                   factor: oldScale.in(),
34                                   suspend: pause.out(),
35                                   injector: newScale.out() )
36
37     def testList = [ scaler, control]
38
39     new PAR(testList).run()
40 }
41 }
```

Listing 17-4 The Scaling Device Process Definition

The `ScalingDevice` has an `inChannel` property {13} from which input numbers are read and an `outChannel` property {14} to which objects of `ScaledData` are written. The `run` method is simply the parallel instantiation {37, 39} of a `Scale` {22} and a `Controller` {30} process. These processes are connected by means of the channels `oldScale` {18}, `newScale` {19} and `pause` {20} as described in 5.1 and 5.1.4.

This e-book
is made with
SetaPDF



SETASIGN



PDF components for **PHP** developers

www.setasign.com



17.3 Running The Test

The test requires that the two parts, PNUT and TestNetwork execute as nodes of a network. The script to run the `ScalingDevice` node is shown in Listing 17-5. This node must be created first. The node is created first {10-14}. The network channels are now created, `ordinaryInput` {16} first, from which the signal generated by `RunTestPart` can be read enabling the rest of the node's creation. The `scaledOutput` channel {19} that connects the `ScalingDevice` to the Test Network can now be created. The signal between the nodes ensures that the necessary input channels are created before they are written to. The `ScalingDevice` process is then invoked within a `PAR` {22}.

```
10 def testPartIP = "127.0.0.1"
11 def deviceIP = "127.0.0.2"
12 def testPartAddr = new TCPIPNodeAddress(testPartIP, 3000)
13 def deviceAddr = new TCPIPNodeAddress(deviceIP, 3000)
14 Node.getInstance().init(deviceAddr)
15
16 def ordinaryInput = NetChannel.net2one()
17 println "ordinaryInput location = ${ordinaryInput.getLocation()}"
18 ordinaryInput.read()
19 def scaledOutput = NetChannel.one2net(testPartAddr, 51)
20 println "scaledOutput location = ${scaledOutput.getLocation()}"
21
22 new PAR(new ScalingDevice (inChannel: ordinaryInput, outChannel:
    scaledOutput) ).run()
```

Listing 17-5 The ScalingDevice Node Script

The node that runs the TestNetwork is created as part of the class `RunTestPart`, Listing 17-1, where it can be seen that the corresponding ends of the channels, `ordinaryInput` {16} and `scaledOutput` {19} are created.

17.4 Summary

This chapter has shown that it is possible to test a system that is intended to run in parallel using an existing technology JUnit and GroovyTestCase formulation. The formulation described is somewhat limited in that only one test can be undertaken against the system under test, which is not the normal mode of operation within the JUnit framework.

The package `c17` contains some further process definitions that can be used to create TestNetworks depending upon the nature of the PNUT.

18 Mobile Agents: Going for a Trip

This chapter introduces the concept of mobility and agents by:

- defining a mobile agent interface
- enabling connection and disconnection of an agent from a host node
- letting agents manage their own transition from one node to another
- using the JCSP `ProcessManager` class to manage agent execution using the `start` and `join` methods
- developing a series of increasingly complex examples to demonstrate the concept

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 each node and 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 (Kosek, et al., 2009) (Kerridge, et al., 2008).

18.1 Mobile Agent Interface

The `MobileAgent` interface is shown in Listing 18-1. It extends `CSProcess` {10} 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. First, `connect` {11}, which is passed a `List` of channels by which the agent is able to communicate with its host and any other initialisation properties. Secondly, `disconnect` {12} 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 and any other properties of the agent that are not serializable.

```
10 interface MobileAgent extends CSProcess, Serializable {  
11     abstract connect(x)  
12     abstract disconnect()  
13 }
```

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 a 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, as many times as required.

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` {12} and `fromLocal` {13} and will collect data in the `List results` {15}. The `connect` method has a `List` parameter, `c` {17}, that contains two channels that are the `toLocal` and `fromLocal` channel ends respectively {18, 19}. The `disconnect` method {22–23} simply sets the local channels to `null`.

The `Agent`'s `run` method, which is required because the `MobileAgent` interface implements the interface `CSPProcess`, simply writes the value of `results` to the `toLocal` channel {26} and then reads the `results` back from the `fromLocal` channel {27}. At which point the `Agent` process will terminate. It is assumed that local process running at the node will modify the `results` object in some way.

gaiteye
Challenge the way we run

**EXPERIENCE THE POWER OF
FULL ENGAGEMENT...**

**RUN FASTER.
RUN LONGER..
RUN EASIER...**

**READ MORE & PRE-ORDER TODAY
WWW.GAITEYE.COM**

The advertisement features a background image of a person running on a path during a sunrise or sunset. The Gaiteye logo is in the top left. The main text is in the middle left. A yellow call-to-action button is in the bottom right, with a hand cursor icon pointing to it.

```
10 class Agent implements MobileAgent {
11
12     def ChannelOutput toLocal
13     def ChannelInput fromLocal
14
15     def results = [ ]
16
17     def connect ( c ) {
18         this.toLocal = c[0]
19         this.fromLocal = c[1]
20     }
21     def disconnect () {
22         toLocal = null
23         fromLocal = null
24     }
25     void run() {
26         toLocal.write (results)
27         results = fromLocal.read()
28     }
29 }
```

Listing 18-2 The Agent Process

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` {12} and `outChannel` {13} connect the Root process to the ring of processes. The property `iterations` {14} indicates how many times the Agent will be sent round the ring of processes. The property `initialValue` {15} is a String that will be placed in the results List as the first element of that list.

```
10 class Root implements CSProcess{
11
12     def ChannelInput inChannel
13     def ChannelOutput outChannel
14     def int iterations
15     def String initialValue
16
17     void run() {
18         def N2A = Channel.one2one()
19         def A2N = Channel.one2one()
20         def ChannelInput toAgentInEnd = N2A.in()
21         def ChannelInput fromAgentInEnd = A2N.in()
```

```
22     def ChannelOutput toAgentOutEnd = N2A.out()
23     def ChannelOutput fromAgentOutEnd = A2N.out()
24
25     def theAgent = new Agent( results: [initialValue])
26
27     for ( i in 1 .. iterations) {
28         outChannel.write(theAgent)
29         theAgent = inChannel.read()
30         theAgent.connect ( [fromAgentOutEnd, toAgentInEnd ] )
31         def agentManager = new ProcessManager (theAgent)
32         agentManager.start()
33         def returnedResults = fromAgentInEnd.read()
34         println "Root: Iteration: $i is $returnedResults "
35         returnedResults << "end of " + i
36         toAgentOutEnd.write (returnedResults)
37         agentManager.join()
38         theAgent.disconnect()
39     }
40 }
41 }
```

Listing 18-3 The Root Process Definition

The channels N2A {18} and A2N {19} provide the local connections between `theAgent` and this node. They cannot be accessed externally from this node and hence are defined within the `run` {17} method. The input and output ends of these local channels are obtained {20–23}. A variable, `theAgent`, is defined {25} 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 {27}. Initially, `theAgent` is written to the `outChannel` {28} and then the `Root` process waits until `theAgent` can be read from its `inChannel` {29}, 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 {30}. An `agentManager` of type `ProcessManager` is defined {31} and is used to manage the operation of the interaction of `theAgent` in parallel with the `Root` node. The `agentManager` is then started {32}. It first reads the `returnedResults` that are written by `theAgent` {33} using the `fromAgentInEnd` input channel. The value of `returnedResults` is printed on the console window {34} and then written back to `theAgent`, modified to indicate the end of an iteration {35}, using the `toAgentOutEnd` output channel {36}. 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 {37}, 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 {38}. The `Root` process will now progress to execute any outstanding iterations.

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 {12, 13} provide the channel connections to the ring of channels connecting all the processes together. The property `nodeId` {14} is just an integer identifier for the node. The mechanism by which the node is connected to the agent {17–22} is identical to that previously described for the `Root` process. The value of `nodeId` is copied into a `localValue` variable {23} and will be used in interactions with the agent.

wethrive.net

How to retain your top staff

FIND OUT NOW FOR FREE

DO YOU WANT TO KNOW:

- What your staff really want?
- The top issues troubling them?
- How to make staff assessments work for you & them, painlessly?

Get your free trial

Because happy staff get more done

Click on the ad to read more

The main body of the process is an infinite loop {25} and is almost identical to that previously described for the `Root` process in that on receipt of `theAgent` {26} they are connected together {27} and started within a `ProcessManager` {28, 29}. The only difference is that the results passed from `theAgent` to this process are read into `currentList` {30}. The value of `localValue` is then appended to `currentList` {31} before it is printed in the process console window {32}. It is then written back to the agent {33}. Once `theAgent` has disconnected {35} from this process it can be written to the `outChannel` for transfer to the next process on the ring {36}. Finally, `localValue` is incremented by 10 {37} as this makes it easier to observe the behaviour after a number of iterations around the ring of processes.

```
10 class ProcessNode implements CSProcess{
11
12     def ChannelInput inChannel
13     def ChannelOutput outChannel
14     def int nodeId
15
16     void run() {
17         def N2A = Channel.one2one()
18         def A2N = Channel.one2one()
19         def ChannelInput toAgentInEnd = N2A.in()
20         def ChannelInput fromAgentInEnd = A2N.in()
21         def ChannelOutput toAgentOutEnd = N2A.out()
22         def ChannelOutput fromAgentOutEnd = A2N.out()
23         def int localValue = nodeId
24
25         while (true) {
26             def theAgent = inChannel.read()
27             theAgent.connect ( [fromAgentOutEnd, toAgentInEnd] )
28             def agentManager = new ProcessManager (theAgent)
29             agentManager.start()
30             def currentList = fromAgentInEnd.read()
31             currentList << localValue
32             println "Node $nodeId: updated list = $currentList"
33             toAgentOutEnd.write (currentList)
34             agentManager.join()
35             theAgent.disconnect()
36             outChannel.write(theAgent)
37             localValue = localValue + 10
38         }
39     }
40 }
```

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 3 and the agent will travel round the ring of processes 3 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 in the folder `ChapterExamples/src/c18/net2`.

```
Number of Nodes ? 3
Number of Iterations ? 3
Initial List Value ? ex1
Node 1: list = [ex1, 1]
Node 2: list = [ex1, 1, 2]
Node 3: list = [ex1, 1, 2, 3]
Root: Iteration: 1 is [ex1, 1, 2, 3]
Node 1: list = [ex1, 1, 2, 3, end of 1, 11]
Node 2: list = [ex1, 1, 2, 3, end of 1, 11, 12]
Node 3: list = [ex1, 1, 2, 3, end of 1, 11, 12, 13]
Root: Iteration: 2 is [ex1, 1, 2, 3, end of 1, 11, 12, 13]
Node 1: list = [ex1, 1, 2, 3, end of 1, 11, 12, 13, end of 2, 21]
Node 2: list = [ex1, 1, 2, 3, end of 1, 11, 12, 13, end of 2, 21, 22]
Node 3: list = [ex1, 1, 2, 3, end of 1, 11, 12, 13, end of 2, 21, 22, 23]
Root: Iteration: 3 is [ex1, 1, 2, 3, end of 1, 11, 12, 13, end of 2, 21, 22, 23]
```

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) 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 that data within its own internal structures. The agent makes these collected data values available 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.

```
10 def int nodeId = Ask.Int ("Node identification (2..9) ? ", 2, 9)
11 def Boolean last = Ask.Boolean ("Is this the last node? - ( y or n):")
12
13 def ipBase = "127.0.0."
14 def nodeIP = ipBase + nodeId
15 def nodeAddress = new TCPIPNodeAddress(nodeIP, 3000)
16 Node.getInstance().init(nodeAddress)
17 def fromRing = NetChannel.net2one()
18 fromRing.read()
19
20 def nextNodeIP = (last) ? "127.0.0.1" : ipBase + (nodeId + 1)
21
22 def nextNodeAddress = new TCPIPNodeAddress(nextNodeIP, 3000)
23 def toRing = NetChannel.one2net(nextNodeAddress, 50)
24 toRing.write(0)
25
26 def processNode = new ProcessNode ( inChannel: fromRing,
27                                     outChannel: toRing,
28                                     nodeId: nodeId)
29
30 new PAR ([ processNode]).run()
```

Listing 18-5 The Run Node Script



The advertisement features a black header with the CMO Inspired Conference logo on the left, which consists of a green speech bubble containing the letters 'CMO'. To the right of the logo, the text 'INSPIRED CONFERENCE' is written in large, white, bold, sans-serif capital letters. Below this, in smaller white capital letters, is the date and location: '25 OCTOBER | DE VERE BEAUMONT ESTATE | OLD WINDSOR UK'. The main body of the ad is a collage of three images: the top image shows a large, white, classical-style building with many windows, surrounded by green trees and a fountain in the foreground; the bottom-left image shows a woman in a black dress speaking into a microphone on a stage with a 'INSPIRED CONFERENCE' backdrop; the bottom-right image shows a man in a light-colored shirt presenting to an audience. At the bottom of the ad, a black banner contains the text 'Join Over 100 Chief Marketing Officers & Digital Innovators' in green.

The identification of a node, `nodeId`, is obtained by means of a simple user interaction {10}. It is required that the processing nodes are numbered consecutively from 2 upwards. It is also necessary to know whether this is the last node in the ring {11}. The `nodeId` is used as the last part of the node's IP-address {13, 14}. A node address is created {15} with the node listening for communications on port 3000, after which a node instance is created {16}. An input channel `fromRing` is now defined as a `net2one` channel {17}. The script now reads an input from the `fromRing` channel {18}. The order of executing the node creation scripts is crucial. All the node processes have to be created before the root node is created. When the root node is created it will send a signal to the first node which will be read on the `fromRing` channel. The first node can then create its output channels, knowing the required node has been created.

The IP-address of the next node can now be determined {20}. If this is not the last node then its IP-address is one more than this node's IP-address, otherwise it is the address of the root node which is assumed to be "127.0.0.1". The address of the next node can be determined and is also assumed to be listening on port 3000. The output channel `toRing` can be created using `nextNodeAddress` and it will be located at the default channel number 50 {23}. The initialisation signal can now be sent to the next node {24} after which the node process can be constructed {26–28} and invoked {30}.

The script to run the root node is very similar (Listing 18-6), except that we need to determine the number of iterations {15} and the `initialValue` of the results list {16}. The script creates the `toRing` {10–13} and `fromRing` {18–20} channels. The script then writes the initialisation signal to the first node in the ring {22} and then waits to receive the returned signal from the ring of nodes {23}.

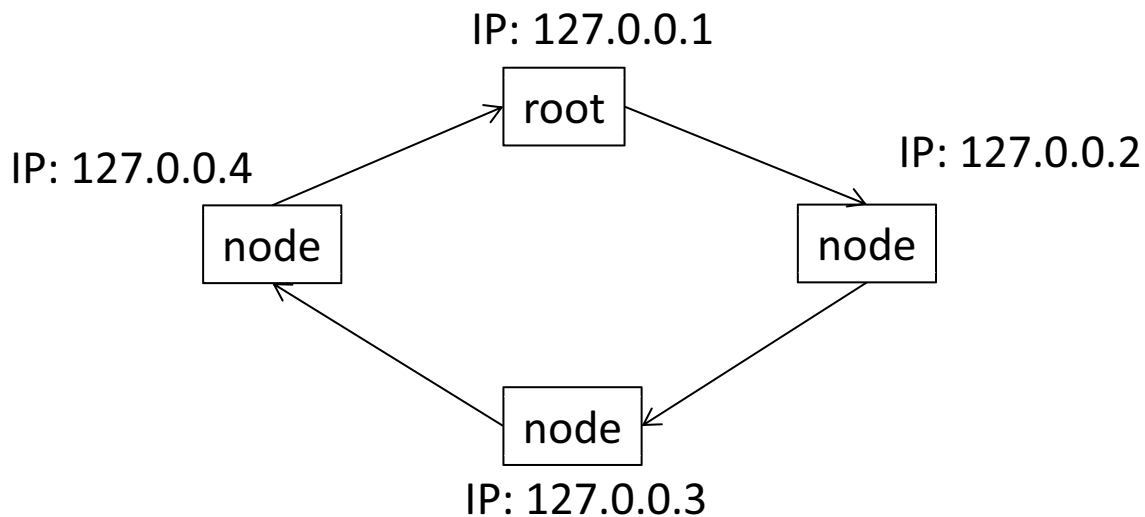
The node is then constructed {25–28} and executed {30}. The output from this set of nodes is similar to that shown above and in particular, the output from the root node is identical for the same number of nodes and iterations. This can be observed by running the required node scripts `RunNode` and `RunRoot` available in the package `ChapterExamples/src/c18/net2`.

```
10 def rootIP = "127.0.0.1"
11 def rootAddress = new TCPIPNodeAddress(rootIP, 3000)
12 Node.getInstance().init(rootAddress)
13 def fromRing = NetChannel.net2one()
14
15 def int iterations = Ask.Int ("Number of Iterations ? ", 1, 9)
16 def String initialValue = Ask.string ( "Initial List Value ? ")
17
18 def nextNodeIP = "127.0.0.2"
19 def nextNodeAddress = new TCPIPNodeAddress(nextNodeIP, 3000)
20 def toRing = NetChannel.one2net(nextNodeAddress, 50)
21
22 toRing.write(0)
23 fromRing.read()
24
```

```
25 def rootNode = new Root ( inChannel: fromRing,  
26                          outChannel: toRing,  
27                          iterations: iterations,  
28                          initialValue: initialValue )  
29  
30 new PAR ( [rootNode] ).run()
```

Listing 18-6 The Run Root Script

Figure 18-1 shows the structure that can be created with a root and three other nodes. Similar structures will be created for the other examples in this chapter. The system will not deadlock because the channels are used to send the agent from one node to the next and not for communication between the nodes. The internode communication structure is neither shown nor created.

**Figure 18-1** Node Structure

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 also 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 all 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 {14} that is the location of a net channel used by the agent to return values back to the root node. The property `backChannel` holds all the data required to create a net channel output.

```
10 class BackAgent implements MobileAgent {
11
12     def ChannelOutput toLocal
13     def ChannelInput fromLocal
14     def NetChannelLocation backChannel
15
16     def results = [ ]
17
18     def connect ( c ) {
19         this.toLocal = c[0]
20         this.fromLocal = c[1]
21     }
22
23     def disconnect () {
24         toLocal = null
25         fromLocal = null
26     }
27
28     void run() {
29         def toRoot = NetChannel.one2net (backChannel)
30         toLocal.write (results)
31         results = fromLocal.read()
32         def last = results.size - 1
33         toRoot.write(results[last])
34     }
35 }
```

Listing 18-7 The BackAgent Specification

The `run` method {28–34} is also modified slightly to permit the return value communication. The agent initially makes the connection for the `backChannel` creating a net output channel `toRoot` {29}. The interaction with the node is the same as before {30, 31}. The index of the `last` element in the `results` list is determined {32} and this element is then written to the `toRoot` channel {33} immediately back to the root process.

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` {16} is required to provide the `NetChannelInput` of the channel that connects the `BackAgent` to the `BackRoot`, when it is running in another 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 {19–24}. The net channel location of the `backChannel` is then obtained by a call to `getLocation()` and stored as `backChannelLocation` {26}.

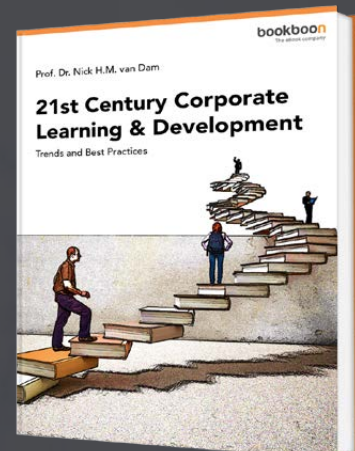
An instance of `BackAgent` is then constructed as `theAgent` {28, 29}, with property values of a list containing the element `initialValue` and the `backChannelLocation`. The `BackRoot` process can receive inputs on its `inChannel`, when the `BackAgent` returns to the `BackRoot` process, or from the `BackAgent` on the `backChannel`, when `BackAgent` is running in another node. The alternative `rootAlt` captures this behaviour {31}. The agent is written to the `outChannel` {32} to start its trip around the nodes. A count variable `i` {33} and a Boolean `running` {34} are defined and initialised. The main loop of the process now commences {36} with the determination of the source of any input communication {37}.

Case 0 {39} relates to return of the agent from an iteration around the other nodes. The agent is read from `inChannel` {40} into `theAgent` and subsequent processing is the same as previously described, except that a returned value still has to be read from `theAgent` on the `backChannel` {48}, which is stored in `backValue` but is effectively 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. If this communication did not take place the `backAgent` would deadlock because it is expecting to output a value on the `backChannel`. The remainder of this alternative's coding {49–58} relates to the management of the number of iterations and the termination of the process and is the same as in the previous agent structure.

Free eBook on Learning & Development

By the Chief Learning Officer of McKinsey

Download Now



Click on the ad to read more

Case 1 deals with an input from the agent when it is running on another node. A variable `backValue` is read from `backChannel` {61} and printed {62}.

```
10 class BackRoot implements CSProcess{
11
12     def ChannelInput inChannel
13     def ChannelOutput outChannel
14     def int iterations
15     def String initialValue
16     def NetChannelInput backChannel
17
18     void run() {
19         def N2A = Channel.one2one()
20         def A2N = Channel.one2one()
21         def ChannelInput toAgentInEnd = N2A.in()
22         def ChannelInput fromAgentInEnd = A2N.in()
23         def ChannelOutput toAgentOutEnd = N2A.out()
24         def ChannelOutput fromAgentOutEnd = A2N.out()
25
26         def backChannelLocation = backChannel.getLocation()
27
28         def theAgent = new BackAgent( results: [initialValue],
29                                     backChannel: backChannelLocation)
30
31         def rootAlt = new ALT ( [inChannel, backChannel])
32         outChannel.write(theAgent)
33         def i = 1
34         def running = true
35
36         while ( running) {
37             def index = rootAlt.select()
38             switch (index) {
39                 case 0: // agent has returned
40                     theAgent = inChannel.read()
41                     theAgent.connect ( [fromAgentOutEnd, toAgentInEnd] )
42                     def agentManager = new ProcessManager (theAgent)
43                     agentManager.start()
44                     def returnedResults = fromAgentInEnd.read()
45                     println "Root: Iteration: $i is $returnedResults "
46                     returnedResults << "end of " + i
47                     toAgentOutEnd.write (returnedResults)
48                     def backValue = backChannel.read()
49                     agentManager.join()
50                     theAgent.disconnect()
51                     i = i + 1
52                     if (i <= iterations) {
53                         outChannel.write(theAgent)
```

```
54         }
55         else {
56             running = false
57         }
58         break
59
60     case 1:
61         def backValue = backChannel.read()
62         println "Root: Iteration $i: received $backValue"
63         break
64     } // end switch
65 } // end while
66 } // end run
67 }
```

Listing 18-8 The Back Root Process

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.

```
10 def rootIP = "127.0.0.1"
11 def rootAddress = new TCPIPNodeAddress(rootIP, 3000)
12 Node.getInstance().init(rootAddress)
13 def fromRing = NetChannel.net2one() // 50
14
15 def int iterations = Ask.Int ("Number of Iterations ? ", 1, 9)
16 def String initialValue = Ask.string ( "Initial List Value ? ")
17
18 def backChannel = NetChannel.net2one() // 51
19 println " BackRoot channel location = ${backChannel.getLocation()} "
20
21 def nextNodeIP = "127.0.0.2"
22 def nextNodeAddress = new TCPIPNodeAddress(nextNodeIP, 3000)
23 def toRing = NetChannel.one2net(nextNodeAddress, 50)
24
25 toRing.write(0)
26 fromRing.read()
27
28 def rootNode = new BackRoot ( inChannel: fromRing,
29                               outChannel: toRing,
30                               iterations: iterations,
31                               initialValue: initialValue,
32                               backChannel: backChannel)
33
34 new PAR ( [rootNode] ).run()
```

Listing 18-9 The Script to Run BackRoot

The only differences are the definition of a `NetChannelInput backChannel {18}` and its inclusion as a property in the construction of the `BackRoot` process {32}. It is noted that the `backChannel` is allocated channel number 51 by default, however, this is never needed by the programmer because the channel is created dynamically by the agent when it arrives at each node.

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 each of 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` (2, 3 and 4) 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 (12, 13, 14) 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 2
Root: Iteration 1: received 3
Root: Iteration 1: received 4
Root: Iteration: 1 is [ex2, 2, 3, 4]
Root: Iteration 2: received 12
```



Discover the truth at www.deloitte.ca/careers

Deloitte.

© Deloitte & Touche LLP and affiliated entities.



Click on the ad to read more

```
Root: Iteration 2: received 13
Root: Iteration 2: received 14
Root: Iteration: 2 is [ex2, 2, 3, 4, end of 1, 12, 13, 14]
Root: Iteration 3: received 22
Root: Iteration 3: received 23
Root: Iteration 3: received 24
Root: Iteration: 3 is [ex2, 2, 3, 4, end of 1, 12, 13, 14, end of 2, 22, 23, 24]
```

Output 18-2 Output From the BackRoot Console Window

The output from each of the node processes can be seen in the console print areas associated with each process.

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 a net input channel, `fromRoot` is created {29} and its net channel location determined {30}. Once the back channel, `toRoot`, has been created {31}, it is used to write the `fromRootLocation` to the root process {32}. A value is then read from the `fromRoot` channel and appended to the `results` list {33}. The agent then writes the augmented `results` object to the node process {34}, which then responds with a further revision to `results` {35}. The index of the last revision is determined {36} and this is sent straight back to the root node {37}.

```
28 void run() {
29     def fromRoot = NetChannel.net2one()
30     def fromRootLocation = fromRoot.getLocation()
31     def toRoot = NetChannel.one2net (backChannel)
32     toRoot.write(fromRootLocation)
33     results << fromRoot.read()
34     toLocal.write (results)
35     results = fromLocal.read()
36     def last = results.size - 1
37     toRoot.write(results[last])
38 }
```

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 because whether the agent is running in a remote node or in the root process the same connections to the agent channels have to be made.

The location of the forward channel is read from `backChannel` {47, 68} into `forwardLocation`. This is then used to create the output end of a net channel `forwardChannel` {48, 69}. A variable `rootValue`, initially -1, is written to the `forwardChannel` {49, 70} and then its value is decremented by 1 {50, 71}. This means that the agent and the root processes have created a pair of 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.

```
39     while ( running) {
40         def index = rootAlt.select()
41         switch (index) {
42             case 0:           // agent has returned
43                 theAgent = inChannel.read()
44                 theAgent.connect ( [fromAgentOutEnd, toAgentInEnd] )
45                 def agentManager = new ProcessManager (theAgent)
46                 agentManager.start()
47                 def forwardLocation = backChannel.read()
48                 def forwardChannel = NetChannel.one2net(forwardLocation)
49                 forwardChannel.write (rootValue)
50                 rootValue = rootValue - 1
51                 def returnedResults = fromAgentInEnd.read()
52                 println "Root: Iteration: $i is $returnedResults "
53                 returnedResults << "end of " + i
54                 toAgentOutEnd.write (returnedResults)
55                 def backValue = backChannel.read()
56                 agentManager.join()
57                 theAgent.disconnect()
58                 i = i + 1
59                 if (i <= iterations) {
60                     outChannel.write(theAgent)
61                 }
62                 else {
63                     running = false
64                 }
65                 break
66             case 1:
67                 def forwardLocation = backChannel.read()
68                 def forwardChannel = NetChannel.one2net(forwardLocation)
```

```

70         forwardChannel.write (rootValue)
71         rootValue = rootValue - 1
72         def backValue = backChannel.read()
73         println "Root: During Iteration $i: received $backValue"
74         break
75     } // end switch
76 } // end while
    
```

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

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 {28} 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. As in the previous examples the output generated by each of the nodes can be seen in their console print windows.

© 2013 Accenture. All rights reserved.

be > your degree

Bring your talent and passion to a global organization at the forefront of business, technology and innovation. Discover how great you can be.

Visit accenture.com/bookboon

Be greater than.
consulting | technology | outsourcing

accenture
High performance. Delivered.



```
Number of Iterations ? 3
Initial List Value ? ex3
Root: During Iteration 1: received 2
Root: During Iteration 1: received 3
Root: During Iteration 1: received 4
Root: Iteration: 1 is [ex3, -1, 2, -2, 3, -3, 4, -4]
Root: During Iteration 2: received 12
Root: During Iteration 2: received 13
Root: During Iteration 2: received 14
Root: Iteration: 2 is [ex3, -1, 2, -2, 3, -3, 4, -4, end of 1,
                    -5, 12, -6, 13, -7, 14, -8]
Root: During Iteration 3: received 22
Root: During Iteration 3: received 23
Root: During Iteration 3: received 24
Root: Iteration: 3 is [ex3, -1, 2, -2, 3, -3, 4, -4, end of 1,
                    -5, 12, -6, 13, -7, 14, -8, end of 2,
                    -9, 22, -10, 23, -11, 24, -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 {12, 13} that enable its connection to the node upon which it is hosted. The property `tripList` {14} will hold the net channel locations that form the trip the agent will travel. The `result` property {15} is a list that will be modified as the agent travels to each node. The `pointer` property {16} indicates the next element in `tripList` that is the location to which the agent will travel. The `connect` and `disconnect` methods are identical to those used in previous agents {18–26}.

```
10 class TripAgent implements MobileAgent {
11
12     def ChannelOutput toLocal
13     def ChannelInput fromLocal
14     def tripList = [ ]
15     def results = [ ]
16     def int pointer
17
18     def connect ( c ) {
19         this.toLocal = c[0]
20         this.fromLocal = c[1]
21     }
22
23     def disconnect () {
24         toLocal = null
25         fromLocal = null
26     }
27
28     void run() {
29         toLocal.write (results)
30         results = fromLocal.read()
31         if (pointer > 0) {
32             pointer = pointer - 1
33             def nextChannel = NetChannel.one2net (tripList.get(pointer))
34             disconnect()
35             nextChannel.write(this)
36         }
37         else {
38             println "Agent has returned to TripRoot"
39         }
40     }
41 }
```

Listing 18-12 The Trip Agent Definition

The run method initially writes the current results list to the node process {29} using the toLocal channel and then reads the modified version of results from the channel fromLocal {30}. This is the same as happened in the previously described agents. 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 {38} because the agent can be sent to no other nodes.

If the value of `pointer` is greater than zero {31} 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 a `one2net` output channel variable called `nextChannel`{33}. 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 usual Java self-reference `this` {35}. Thus in this example the agent itself causes itself to be written to the next node, whereas in previous examples the node process has undertaken this task.

18.6.2 The Trip Node Process

Listing 18-13 shows the coding of the `TripNode` process. The property `toRoot` {12} 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 {13}. Its value will also be used as the last element in the node's IP-address. Within the `run` method {16} channels are created {17, 18} together with their channel ends {19–22} which provide the internal channel mechanism by which the agent communicates with the host node, as described previously.

A net input channel is then defined, `agentInputChannel` {24}, and its net channel location is written to the root process using the `toRoot` net output channel {25}. The node process now waits until it can read `theAgent` from the `agentInputChannel` {26}.



What if you could build your future and create the future?

The innovation accelerator

One generation's transformation is the next's status quo. In the near future, people may soon think it's strange that devices ever had to be "plugged in." To obtain that status, there needs to be "The Shift".

.....Alcatel-Lucent 

www.alcatel-lucent.com/careers



Using the local channels, `theAgent` can be connected to the local node and then executed using a `ProcessManager` {27–29}. The interaction with the agent then takes place {30–32}, after which the `agentManager` can join the node process {33}, so that in this case they can both terminate. In this simple case the node process does not contain a loop so the agent can only be hosted once.

```
10 class TripNode implements CSProcess{
11
12     def ChannelOutput toRoot
13     def int nodeId
14
15     def N2A = Channel.one2one()
16     def A2N = Channel.one2one()
17
18     void run() {
19         def ChannelInput toAgentInEnd = N2A.in()
20         def ChannelInput fromAgentInEnd = A2N.in()
21         def ChannelOutput toAgentOutEnd = N2A.out()
22         def ChannelOutput fromAgentOutEnd = A2N.out()
23
24         def agentInputChannel = NetChannel.net2one()
25         toRoot.write ( agentInputChannel.getLocation() )
26         def theAgent = agentInputChannel.read()
27         theAgent.connect ( [fromAgentOutEnd, toAgentInEnd] )
28         def agentManager = new ProcessManager (theAgent)
29         agentManager.start()
30         def currentList = fromAgentInEnd.read()
31         currentList << nodeId
32         toAgentOutEnd.write (currentList)
33         agentManager.join()
34     }
35 }
```

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 {12} 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 {17–22}, in a similar manner as before.

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 {24}. The `for` loop {26–29} then reads from the `fromNodes` channel the net input channel location of each of the nodes, which are appended to `tripList`. The next section of coding {30–35} gets the last element of `tripList`, which becomes the net location to which the agent will be sent first. A net output channel, `firstNodeChannel`, is created {31} from the location. An instance of the `TripAgent` is then constructed {32–34} as `theAgent` after which it can be written to the `firstNodeChannel` {35}.

The remainder of the coding {36–44} shows the return of `theAgent` after the trip. It will be read from the channel `fromNodes` {36}. The process interaction between `theAgent` and the `TripRoot` process is very similar to other such root nodes {37–42}. In order that the root node terminates correctly it is necessary for the agent to disconnect itself from the `TripRoot` node. Thus the host node has to `join()` the `agentManager` {43} and then `theAgent` must call `disconnect()` {44} so that both `theAgent` and the `TripRoot` processes terminate.

```
10 class TripRoot implements CSProcess{
11
12     def ChannelInput fromNodes
13     def String initialValue
14     def int nodes
15
16     void run() {
17         def N2A = Channel.one2one()
18         def A2N = Channel.one2one()
19         def ChannelInput toAgentInEnd = N2A.in()
20         def ChannelInput fromAgentInEnd = A2N.in()
21         def ChannelOutput toAgentOutEnd = N2A.out()
22         def ChannelOutput fromAgentOutEnd = A2N.out()
23
24         def tripList = [ fromNodes.getLocation() ]
25
26         for ( i in 0 ..< nodes) {
27             def nodeChannelLocation = fromNodes.read()
28             tripList << nodeChannelLocation
29         }
30         def firstNodeLocation = tripList.get(nodes)
31         def firstNodeChannel = NetChannel.one2net(firstNodeLocation)
32         def theAgent = new TripAgent( tripList: tripList,
33                                     results: [initialValue],
34                                     pointer: nodes)
35         firstNodeChannel.write(theAgent)
36         theAgent = fromNodes.read()
37         theAgent.connect ( [fromAgentOutEnd, toAgentInEnd] )
38         def agentManager = new ProcessManager (theAgent)
```

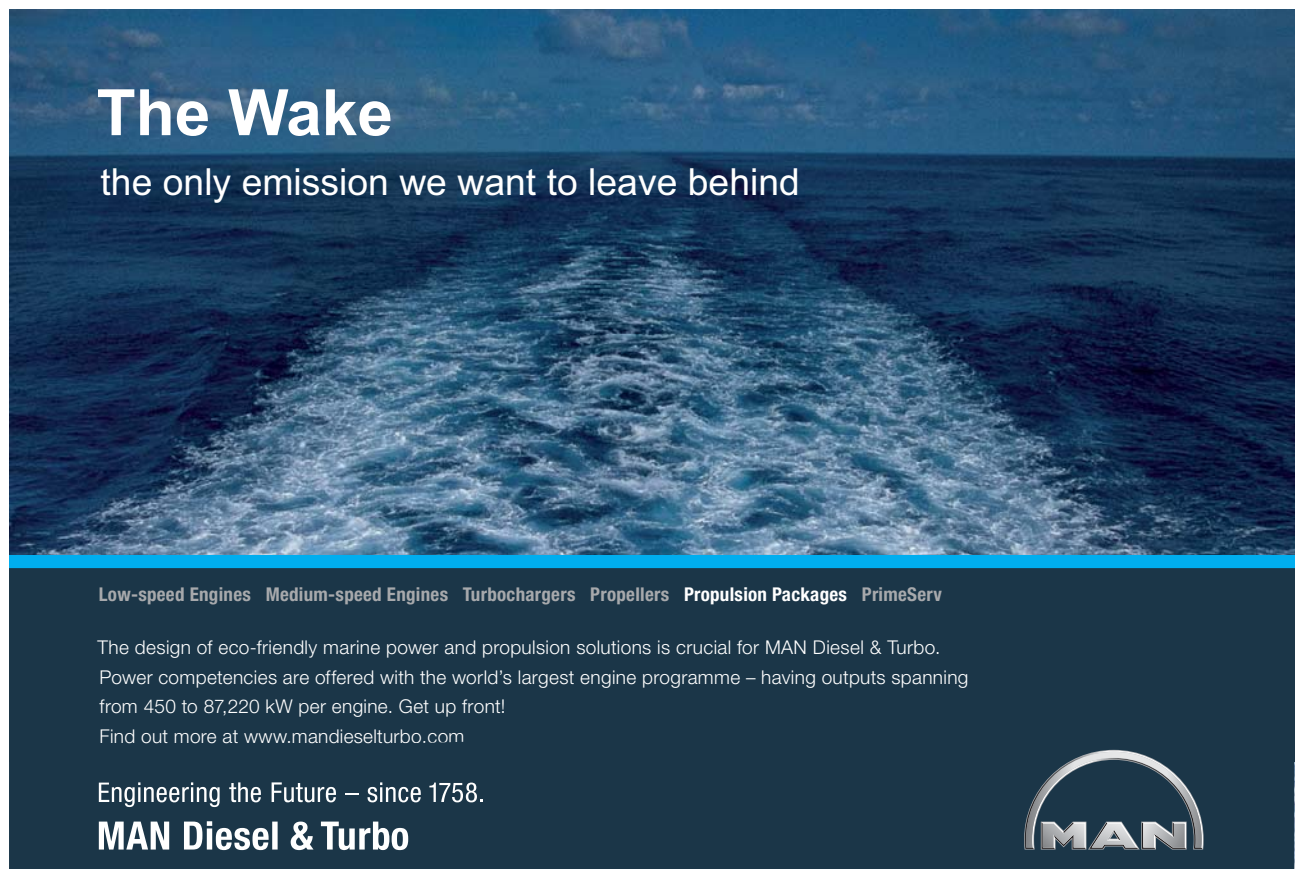
```
39     agentManager.start()
40     def returnedResults = fromAgentInEnd.read()
41     println "TripRoot: has received $returnedResults "
42     toAgentOutEnd.write (returnedResults)
43     agentManager.join()
44     theAgent.disconnect()
45 }
46 }
```

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 {20}` forms the channel between the nodes to the root process.

```
10 def int nodeId = Ask.Int ("Node identification (2..9)? ", 2, 9)
11
12 def ipBase = "127.0.0."
13 def nodeIP = ipBase + nodeId
14 def nodeAddress = new TCPIPNodeAddress(nodeIP, 3000)
15 Node.getInstance().init(nodeAddress)
16
17 def rootNodeIP = "127.0.0.1"
```




The Wake
the only emission we want to leave behind

Low-speed Engines Medium-speed Engines Turbochargers Propellers Propulsion Packages PrimeServ

The design of eco-friendly marine power and propulsion solutions is crucial for MAN Diesel & Turbo. Power competencies are offered with the world's largest engine programme – having outputs spanning from 450 to 87,220 kW per engine. Get up front! Find out more at www.mandieselturbo.com

Engineering the Future – since 1758.
MAN Diesel & Turbo



```
18 def rootNodeAddress = new TCIPNodeAddress (rootNodeIP, 3000)
19
20 def toRoot = NetChannel.any2net (rootNodeAddress, 50)
21
22 def processNode = new TripNode ( toRoot: toRoot,
23                                 nodeId: nodeId)
24
25 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 `net2one` channel `fromNodes` {13} forms the net input channel from the nodes to the root process. In this case the `RunTripRoot` script must be executed first because it creates the net input channel to which all the nodes write a channel location. It therefore holds the input end of the channel. The variable `nodes` {16} is required to ensure that the `TripRoot` process reads all the net channel locations from all the `TripNode` processes.

```
10 def rootIP = "127.0.0.1"
11 def rootAddress = new TCIPNodeAddress (rootIP, 3000)
12 Node.getInstance().init (rootAddress)
13 def fromNodes = NetChannel.net2one()
14
15 def String initialValue = Ask.string ( "Initial List Value ? ")
16 def int nodes = Ask.Int ("Number of nodes (1..8) ? ", 1, 8)
17
18 def rootNode = new TripRoot ( fromNodes: fromNodes,
19                               nodes: nodes,
20                               initialValue: initialValue )
21
22 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 not 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 (1..8) ? 4
TripRoot: has received [ex4, 5, 4, 3, 2]
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, 1996). More prosaically, an agent is that which “denotes something that produces or is capable of producing an effect” (Magedanz, et al., n.d.) 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 & Karmouch, 1998). More recently, it has been argued (Chalmers, et al., 2007) 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. In a later chapter we shall investigate a system in which agents traverse a network to find a process they can load into the node that created the agent, in a network which is created dynamically at run time.



The advertisement features a circular logo on the left with three stylized human figures in the center, surrounded by gears and four arrows pointing clockwise. To the right, the text reads: **UNLEASHING CHANGE MANAGEMENT**, **OCTOBER 18 & 19, 2018**, and **DE RODE HOED AMSTERDAM**. At the bottom, there is a silhouette of an Amsterdam skyline including a windmill and a bridge. The logo for 'Global Executive Events' is in the bottom left corner.

19 Mobile Processes: Ubiquitous Access

The concept of process mobility is introduced by:

- defining a generic system architecture
- communicating process definitions as serializable objects over network channels
- describing a universal capability applicable to mobile devices

The previous chapter showed how it is possible to create network channels over which agents could be transferred. In this chapter we take that concept one stage further and provide a mechanism whereby a process can be communicated from one node to another within the network. The only requirement is that the receiving node has to run a simple process that loads the mobile process. This is further extended to load a process from a server over a wireless network to a mobile device. The mobile device becomes a member of the server's network for the duration of the interaction. The mobile device scans for accessible wireless networks and then is able to download a process from that network with which it can interact with the service provided.

This technology could be used in a retail environment to let stores make offers to customers, as they walk into the store, based upon their previous shopping patterns. In addition, the store could make offers on surplus items to customers they know might be susceptible to the offer. The only requirement is that the customer has a mobile device into which the process loading process has been installed. The customer would also need to store some means of identifying themselves to the store's systems but with loyalty or reward cards this is not a problem.

The technology could be used in a hospital environment to allow access to a hospital information system by registered users, using their own mobile device. The great advantage being that the location of a person can be determined by the wireless access points that are available and this could result in the most appropriate process being downloaded into the mobile device depending upon the user and their role. Obviously, some form of authentication process would be required to ensure authorised access but the advantage of this style of interaction is that no sensitive data is retained in the mobile device.

Finally, it could be used in museums to provide additional resources to visitors about the items on display. In this case, rather than using wi-fi we could use Bluetooth to give more locality of information. The downloaded process could provide additional information in the form of an audio stream giving an aural description of the exhibit, possibly supported by an image that shows the particular part of the object being described. The audio stream could be in any language. The particular advantage for the museum is that visitors can use their own mobile devices, provided they have the process to download other processes.

The capability is provided with the ability to dynamically load classes over the network in an efficient manner that is totally transparent to the programmer and of the underlying network technology. Processes are loaded just like any other object as a `Serializable` data object. The processes will include some of the network channels in their definition that will allow the loaded process to communicate from the mobile device to the server. However, channels that enable communication from the server to the loaded mobile process will need to be created dynamically.

19.1 The Travellers' Meeting System

The meeting system is a service provided by a travel authority such as a railway station or airport that enables people travelling together to find out where other members of a group are located especially in the event of a travel delay (Kerridge & Chalmers, 2006). A member of the group registers the name by which the group recognises itself together with the location of where they are to congregate. Other members may try to create another location but will be informed that the group has already been registered and given that location. Other members will just try to find the meeting location and will be informed of its location. People who try to find a meeting that is not yet registered are informed of this case.

The primary requirement is for an initial channel by which a mobile device can register itself with the server network. This is similar to the `Request` channel used in Chapter 16 to make requests to the printer spooling service. All the `PrintUser` processes knew was that an access channel was available. The situation becomes more complex as we move to a more general environment. If the process loaded into the mobile device is to function with all such publicly available service providers then they are all going to have to use the same mechanism for their access channel. In the case of the hospital environment briefly described above this would not be made publicly available.

Once the initial mobile process has been loaded this can then be used to determine the required service and then further processes can be loaded using private access channels. In the meeting example the initial mobile process will be loaded using the publicly available access channel. The initial mobile process will then determine, by means of a user interaction, whether the user is creating a new meeting location or finding an existing meeting.

19.2 The Service Architecture

The complete architecture is shown in Figure 19-1, in which channel names are shown in *italics* and communicated objects are shown in a regular font. The diagram also shows the IP-addresses, ports and channel numbers used by the different components.

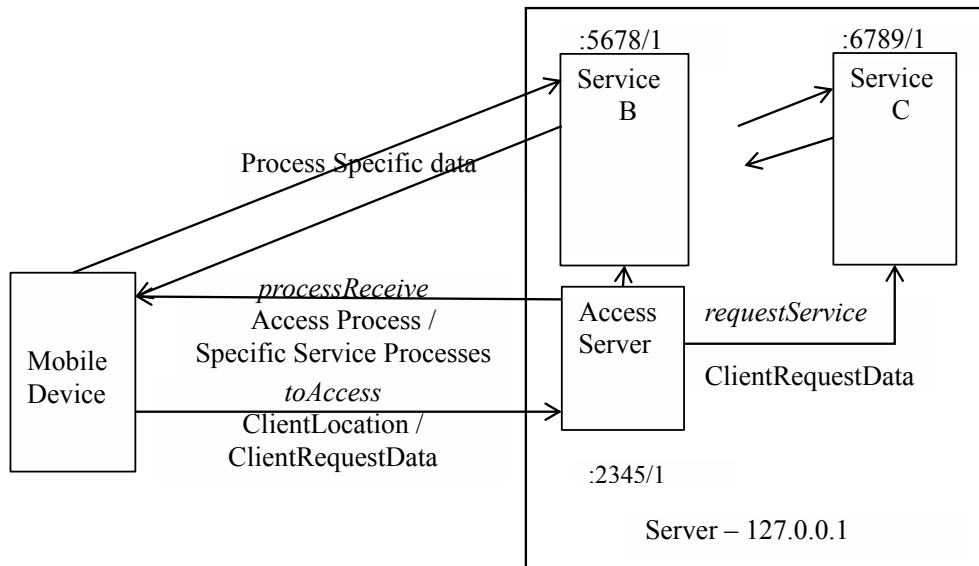


Figure 19-1 The Mobile Device Server Architecture

Initially there are no connections between the Mobile Device and the Server. The mobile device creates the *processReceive* and the *toAccess* any2net channels. Both these channels can be accessed by more than one source process as will be explained later. The location of the *processReceive* channel is sent to the Access Server by means of a Client Location object using the *toAccess* channel. The Access Server then sends the Access Process object to the Mobile Device using the *processReceive* channel. The Mobile Device can now run the Access Process. This creates a user interface with which the user can determine the service they require. The chosen service and the *processReceive* channel location are then sent to the Access Server using a Client Request Data object. The Access Server then determines the required service and passes the object to the required service process together with the *processReceive* channel location. The required Service then sends a specific Service Process client to the Mobile Device using the *processReceive* channel. The Mobile Device can now execute the client service process and interact with the service on the Server. The initial part of the interaction may necessitate the dynamic creation of a net channel with which the service client process running in the Mobile Device can communicate with the service. While the service client and Server are interacting they can communicate service-specific data objects between themselves. The required class definitions for any data will be communicated as part of the service client process when it is first transferred from the Server to the Mobile Device. Each server executes with its own node:port/channel number combination. These are internal and can be created by the service provider. The only port/channel number that has to be fixed for a universal service is that of the Access Server.

19.3 Universal Client

Each mobile device has to execute the same initial process. This is referred to as the Universal Client because it is able to access any such service that follows the same access standard. This is the only process that has to be available in the user's mobile device. All other processes are loaded dynamically once a network connection has been made with the service provider. The Universal Client comprises two processes running in parallel. One provides the functional capability and the other a user interface. This model will be used also in all the processes that are subsequently loaded from the service provider into the user's mobile device. In this demonstration version, we do not model the acquisition of wireless networks but simply assume that such a capability is available.

The script that runs the Universal Client (UC) is shown in Listing 19-1. It simply runs two processes in parallel, `UCInterface` {12} and `UCCapability` {13}, which access a channel `ipChannel` {10} to transfer data between the processes.

```
10 def ipChannel = Channel.one2one(new OverWriteOldestBuffer(5))
11
12 new PAR([ new UCInterface(sendNodeIdentity: ipChannel.out()),
13           new UCCapability(receiveNodeIdentity: ipChannel.in() )]).run()
```

Listing 19-1 The Script use to Run the Universal Client Process in a Simulated Mobile Device

bookboon.com

Corporate eLibrary

See our Business Solutions for employee learning

Click here

Management Time Management

Problem solving Self-Confidence Effectiveness

Project Management Goal setting Motivation Coaching

Click on the ad to read more

The UCInterface process is shown in Listing 19-2. This follows the design principles used in Chapters 11 and 14. The interface simply comprises a set of active components that are placed in the ActiveFrame main {16}. The active components are then executed within a PAR {29}. The resulting graphical interface is shown in Figure 19-2.

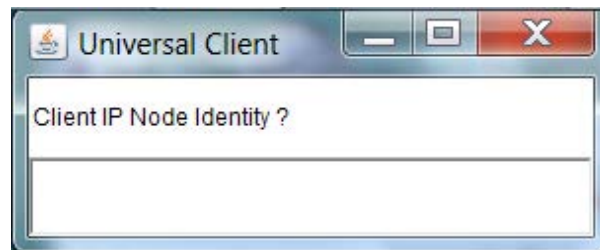


Figure 19-2 Universal Client Interface

The user simply has to type in the last part of the Universal Client's IP-address. The system assumes that the Server processes are located at node "127.0.0.1" and that Universal Client nodes are located at other IP-addresses of the form "127.0.0.?", where the ? will be replaced by whatever the user types in. This interaction is simply to simulate what would happen in a real mobile device that is able to dynamically obtain an IP-address from the server to which it has connected.

```
10 class UCInterface implements CProcess {
11
12     def ChannelOutput sendNodeIdentity
13
14     void run () {
15         def root = new ActiveClosingFrame ("Universal Client")
16         def main = root.getActiveFrame()
17         def requestlabel = new Label("Client IP Node Identity ?")
18         def enterIPfield = new ActiveTextEnterField (null, sendNodeIdentity)
19         def enterIP = enterIPfield.getActiveTextField ()
20         def container = new Container()
21         container.setLayout(new GridLayout(2,1))
22         container.add(requestlabel)
23         container.add(enterIP)
24         main.setLayout(new BorderLayout())
25         main.setSize(480, 640)
26         main.add(container)
27         main.pack()
28         main.setVisible(true)
29         new PAR([root, enterIPfield]).run()
30     }
31 }
```

Listing 19-2 The Universal Client Interface Process

The UCCapability process is shown in Listing 19-3. Initially, the process reads the `clientNodeId` from the user interface using the channel `receiveNodeIdentity` {16}. The `clientIpAddress` is then formed {17} and used to create a node listening on port 1000 {18, 19}. The process then creates a numbered net channel with channel number 2 {21} which will be used to receive processes from the server. The location of this channel has to be sent to the server as `processReceiveLocation` {22, 26}.

```
10 class UCCapability implements CSProcess {
11
12     def ChannelInput receiveNodeIdentity
13     def networkBaseIP = "127.0.0."
14
15     void run(){
16         def clientINodeId = receiveNodeIdentity.read()
17         def clientIpAddress = networkBaseIP + clientINodeId
18         def nodeAddr = new TCPIPNodeAddress(clientIpAddress,1000)
19         Node.getInstance().init(nodeAddr)
20         // create channel on which to receive processes from server
21         def processReceive = NetChannel.numberedNet2One(2)
22         def processReceiveLocation = processReceive.getLocation()
23         //create default channel to access server
24         def accessAddress = new TCPIPNodeAddress("127.0.0.1",2345)
25         def toAccess = NetChannel.any2net(accessAddress, 1)
26         def receiveLocation=new ClientLocation(processReceiveLocation:proces
sReceiveLocation)
27         toAccess.write(receiveLocation)
28         def accessProcess = processReceive.read()
29         def pmA = new ProcessManager(accessProcess)
30         pmA.start()
31         def serviceProcess = processReceive.read()
32         def pmS = new ProcessManager(serviceProcess)
33         pmS.start()
34     }
35 }
```

Listing 19-3 The Universal Client Capability Process

The process now creates the connection to the server. In this example it is assumed that the Access Server is located at node “127.0.0.1” and is listening on port 2345 and this is used to create the `accessAddress` {24}. This is then used to create an `any2net` channel called `toAccess` that uses channel number 1 {25}. An `any2net` channel is used because any number of mobile devices can connect to the server at the same time.

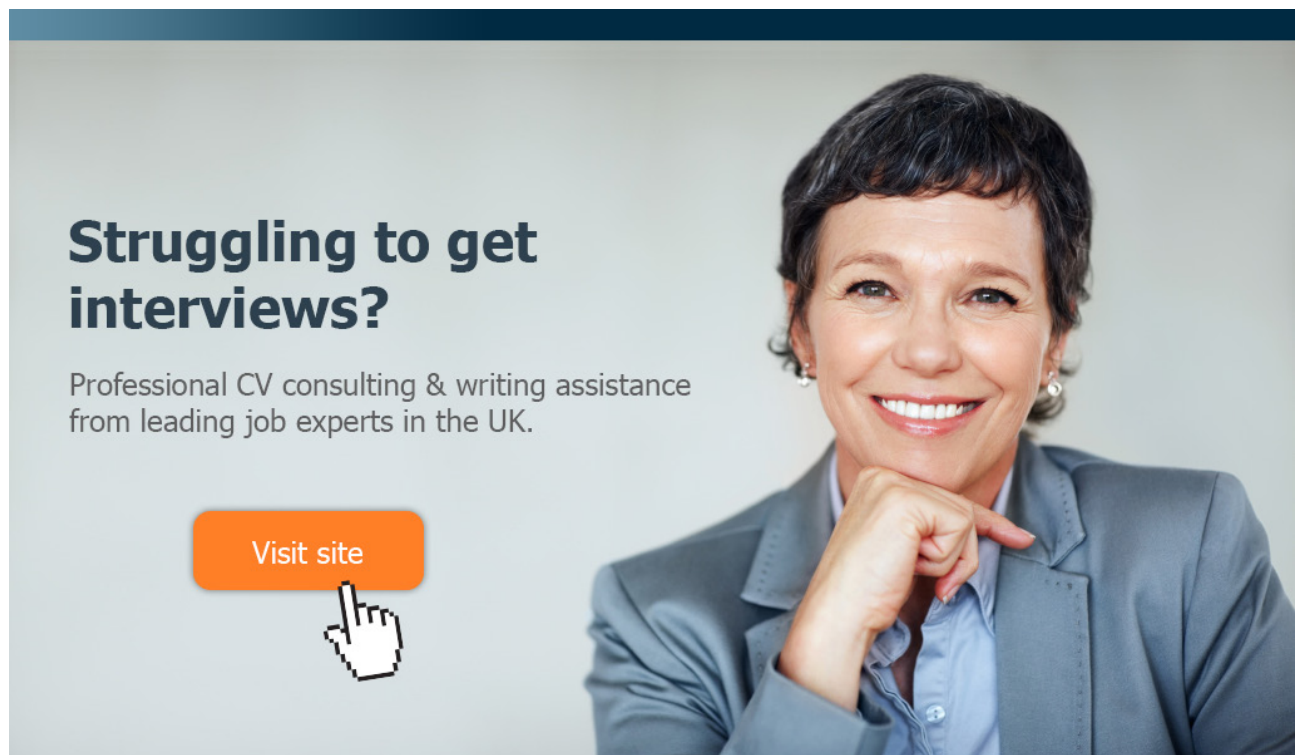
The only parts of this system that have to be standardised for the Universal Client to work anywhere are the port number, 2345, and the access channel number 1. The IP-address of the server can be deduced by the initial wireless interaction between the mobile device and the server.

Once the channel to the Access Server has been created it can be used to send the `processReceiveLocation` variable to the server in an object `ClientLocation` {26, 27}. The process, acting as a client, then reads the `accessProcess` from the `processReceive` channel, created previously {28}. An instance of `ProcessManager` is then started {29, 30}. The `accessProcess` then executes and as a result of another user interaction determines the service required. The `accessProcess` sends that information to the Access Server, which enables the sending of the required service process to the mobile device using the same `processReceive` channel {31}. The `serviceProcess` is then started using another instance of Process Manager {32, 33}. Once the `serviceProcess` terminates the interaction is complete, all the loaded processes can now terminate and the resources can be recovered by the mobile device. Thus the mechanism only uses mobile device resources while the user is interacting with the service.

19.4 The Access Server

The Access Server is a process that simply waits for interactions from Mobile Devices as shown in Listing 19-4.

```
10 def serverIP = "127.0.0.1"
11
12 def accessAddress = new TCPIPNodeAddress(serverIP, 2345)
13 Node.getInstance().init(accessAddress)
14 def accessRequestChannel = NetChannel.numberedNet2One(1)
```



Struggling to get interviews?

Professional CV consulting & writing assistance from leading job experts in the UK.

Visit site



Take a short-cut to your next job!
Improve your interview success rate by 70%.



TheCVagency
Visit theagency.co.uk for more info.



Click on the ad to read more

```
15 def accessRequestLocation = accessRequestChannel.getLocation()
16 println "access request location is $accessRequestLocation"
17 // now create all the request channels to each of the service processes
18 def groupLocationServerAddress = new TCPIPNodeAddress(serverIP, 3456)
19 def requestGLservice = NetChannel.one2net(groupLocationServerAddress, 1)
20 //
21 def AServerAddress = new TCPIPNodeAddress(serverIP, 4567)
22 def requestAService = NetChannel.one2net(AServerAddress, 1)
23 //
24 def BServerAddress = new TCPIPNodeAddress(serverIP, 5678)
25 def requestBService = NetChannel.one2net(BServerAddress, 1)
26 //
27 def CServerAddress = new TCPIPNodeAddress(serverIP, 6789)
28 def requestCService = NetChannel.one2net(CServerAddress, 1)
29
30 while (true) {
31     def clientRequest = accessRequestChannel.read()
32     if (clientRequest instanceof ClientLocation) {
33         def clientProcessLocation = clientRequest.processReceiveLocation
34         def clientProcessChannel = NetChannel.one2net(clientProcessLocation )
35         def accessProcess = new AccessProcess (accessRequestLocation:access
RequestLocation,
36
37                                     processReceiveLocation:
38                                     clientProcessLocation)
39         clientProcessChannel.write (accessProcess)
40     }
41     if (clientRequest instanceof ClientRequestData ) {
42         def serviceRequired = clientRequest.serviceRequired
43         switch (serviceRequired) {
44             case "Service - A" :
45                 requestAService.write(clientRequest)
46                 break
47             case "Service - B" :
48                 requestBService.write(clientRequest)
49                 break
50             case "Service - C" :
51                 requestCService.write(clientRequest)
52                 break
53             case "Group Location Service" :
54                 requestGLservice.write(clientRequest)
55                 break
56         }
57     }
58 }
```

Listing 19-4 The Script for the Access Server

Initially the Access server node is created {12, 13} together with the `accessRequestChannel` and its location {14, 15}. Each of the `requestService` channels for each of the services available on this server are now created {18–28}. The demonstration Server has four services known as A, B, C and Group Location. Only the latter provides a service in this demonstration. It should be noted that each service is located to a different port on the Server node, thereby avoiding communication contention between the services. In larger systems each service could be provided by different nodes on an internal TCP/IP network.

The main body of the Access Server is a loop that repeatedly waits for a `clientRequest`, which it reads on the `accessRequestChannel` {31}. The type of communication is determined {32, 39}. If the request is a `ClientLocation` the Access Server sends a newly constructed instance of an `AccessProcess` to the requesting Mobile Device using the `processReceiveLocation` held in the `clientRequest` {33–38} to create the `clientProcessChannel`.

If the request is an instance of `ClientRequestData`, the Access Server determines the required service and then passes the object to the required service process using the appropriate channel it had previously created {39–54}.

19.4.1 Access Process

The `AccessProcess` is the process that is written to the Mobile Device, which then enables the user to specify the specific service they wish to interact with. The `AccessProcess` comprises capability and user interface processes. The user interface for this process is shown in Figure 19-3 . The user selects a service by clicking the button associated with the required service.



Figure 19-3 The Access Server Interface

Listing 19-5 shows the definition of the `AccessProcess`.

```
10 class AccessProcess implements CSProcess, Serializable {
11
12     def NetChannelLocation processReceiveLocation
13     def NetChannelLocation accessRequestLocation
14
15     void run () {
16         def buttonChannel = Channel.one2one(new OverWriteOldestBuffer(5))
17         new PAR ([new AccessInterface( buttonEvents: buttonChannel.out()),
18                 new AccessCapability( buttonEvents: buttonChannel.in(),
19                                     processReceiveLocation:
20                                     processReceiveLocation,
21                                     accessRequestLocation:accessRequest
22                                     Location)]) .run()
23     }
24 }
```

Listing 19-5 The Access Process definition

The process has two properties that give the location of the process receive and the access request locations respectively {12, 13}. The `buttonChannel` provides a connection between the two internal processes {16}. The process is itself a parallel of the two processes providing the capability and the interface {17–20}. The `AccessCapability` process is shown in Listing 19-6.



e-learning for kids

- The number 1 MOOC for Primary Education
- Free Digital Learning for Children 5-12
- 15 Million Children Reached

About e-Learning for Kids Established in 2004, e-Learning for Kids is a global nonprofit foundation dedicated to fun and free learning on the Internet for children ages 5 - 12 with courses in math, science, language arts, computers, health and environmental skills. Since 2005, more than 15 million children in over 190 countries have benefitted from eLessons provided by EFK! An all-volunteer staff consists of education and e-learning experts and business professionals from around the world committed to making difference. eLearning for Kids is actively seeking funding, volunteers, sponsors and courseware developers; get involved! For more information, please visit www.e-learningforkids.org.


```
10 class AccessCapability implements CSProcess {
11
12     def ChannelInput buttonEvents
13     def NetChannelLocation processReceiveLocation
14     def NetChannelLocation accessRequestLocation
15
16     void run () {
17         def serviceRequired = buttonEvents.read()
18         def clientRequest = new ClientRequestData ( processReceiveLocation:
19                                                     processReceiveLocation,
20                                                     serviceRequired:
21                                                         serviceRequired )
22     }
23 }
```

Listing 19-6 The AccessCapability Process Definition

The process reads the name of the required service from the `buttonEvents` channel {17} and then constructs an instance of `ClientRequestData` which it sends to the `AccessServer` using the `toAccess` channel it has created internally as an `any2net` channel {18–21}. Once the `AccessServer` reads the `clientRequest` it can pass it to the identified service process as described previously.

19.5 Group Location Service

The Group Location Service is the only one that provides a real user interaction. The other services are simply dummies and will not be described further. The Group Location Server is shown in Listing 19-7.

```
10 def serverIP = "127.0.0.1"
11 // each service is located at a different port
12 def groupLocationServerAddress = new TCPIPNodeAddress(serverIP, 3456)
13 Node.getInstance().init(groupLocationServerAddress)
14 def initialChannel = NetChannel.numberedNet2One(1)
15 def requestChannel = NetChannel.numberedNet2One(2)
16 def requestLocation = requestChannel.getLocation()
17 def glAlt = new ALT([initialChannel, requestChannel])
18 def locationMap = [:]
19 while (true) {
20     switch (glAlt.select()){
21         case 0: // initial request
22             def request = initialChannel.read()
23             def processSendChannel = NetChannel.one2net(request.processReceiveLocation)
24             def glProcess = new GLprocess(requestLocation: requestLocation)
25             processSendChannel.write(glProcess)
26             break
27         case 1: // request from user
```

```
28     def requestData = requestChannel.read()
29     def responseChannel = NetChannel.one2net(requestData.responseLocation)
30     def groupName = requestData.groupName
31     if ( locationMap.containsKey(groupName)) {
32         requestData.location = locationMap[groupName]
33         responseChannel.write(requestData)
34     }
35     else {
36         requestData.location == null
37         responseChannel.write(requestData)
38         requestData = requestChannel.read()
39         groupName = requestData.groupName
40         location = requestData.location
41         locationMap.put(groupName, location)
42         println "location map = $locationMap"
43     }
44     break
45 }
46 }
```

Listing 19-7 The Script of the Group Location Server

This service listens on port 3456 {12} for which a node is created {13}. The `initialChannel` is the channel {14} upon which a client request is received from the Access Server. The `requestChannel` is the one from which requests from the client process running on the mobile device will be read. The process alternates over its two input channels {17}. The variable `locationMap` is used to record where each group is located {18}. The server reads inputs from either of its input channels {19, 20}.

In the case of an initial request it reads the `request` {22} and extracts the location of the Mobile Device's `processReceive` channel to create a channel {23} to which it can write {25} a newly constructed instance of `GLprocess` {24}. A `GLprocess` has a single property which is the location of the request channel.

In the case of a request from the Mobile Device {27–44}, the server reads the `requestData` from the `requestChannel` {28}. The `requestData` contains a property that is the location of the channel upon which the server is to respond to the Mobile Device and this is used to create the required `responseChannel` {29}. The other property of the `requestData` is the name of the group. If the `locationMap` contains this `groupName` then the location of the group can be returned to the user of the Mobile Device {31–34}. The server is now interacting with the instance of `GLprocess` that is executing on the Mobile Device. If the `locationMap` does not contain the `groupName` then this is a new group and the user must be asked for the location where the group are to meet. The `location` is set null in the `requestData` object and returned to the `GLprocess` running on the Mobile Device {37}. The server now reads a location and adds it to the `locationMap` {36–42}.

19.5.1 The GLprocess

The GLprocess also comprises a capability process and a user interface process running in parallel as previously described. The GLcapability process is the one which undertakes the interaction with the server and is shown in Listing 19-8.

```
10 class GLcapability implements CProcess, Serializable {
11
12     def ChannelInput nameChannel
13     def ChannelInput locationChannel
14     def ChannelOutput label1Config
15     def ChannelOutput label2Config
16     def NetChannelLocation requestLocation
17
18     void run() {
19         def responseChannel = NetChannel.net2one()
20         def responseLocation = responseChannel.getLocation()
21         def requestChannel = NetChannel.any2net(requestLocation)
22         def groupName = nameChannel.read()
23         def groupData = new GLdata( responseLocation: responseLocation,
24             groupName: groupName)
25         requestChannel.write(groupData)
26         def replyData = responseChannel.read()
27
28         if (replyData.location != null) {
29             label1Config.write("Meeting at")
30             label2Config.write(replyData.location)
31         }
32         else {
33             label1Config.write("Does not yet exist")
34             label2Config.write("Type meeting location")
35             def location = locationChannel.read()
36             def newGroup = new GLdata( responseLocation: responseLocation,
37                 groupName: groupName,
38                 location: location)
39             requestChannel.write(newGroup)
40         }
41     } // end run
42 }
```

Listing 19-8 The GLcapability Process Definition

The channels `nameChannel`, `locationChannel`, `label1Config` and `label2Config` provide connections to the active elements of the user interface process {12–15}. The property `requestLocation` {16} is the location of the net channel by which the process interacts with the Group Location Server when it is running in the user's Mobile Device. The process creates a `responseChannel` that can be used by the server {19}. It creates the channel `requestChannel` from the `requestLocation` property which it will use to send requests to the server {21}. The process now reads the name of the group from the user interface using the `nameChannel` {22}. A new `GLdata` object is created with properties `groupName` and the location of the `responseChannel` {23, 24}. It is then written to the server using the `requestChannel` {25}. It then reads a response as `replyData` {26}, thereby implementing the client behaviour required in an interaction with a server. If the `location` in `replyData` is not `null` then the `ActiveLabels` in the interface can be updated accordingly {28–30}. If the `location` is `null` then no location for the group has yet been recorded by the service, so the interface labels are updated appropriately and the process then reads the `location` from the `locationChannel`, which is connected to an `ActiveTextEnterField` in the interface. A new `GLdata` object is created with all the required property values and written to the server {36–39}, which can now update its `locationMap` as described previously.

FACTCARDS

Are you working in academia, research or science? And have you ever thought about working and moving to the Netherlands?

Arriving 33

Living 50

Studying 51

Working 101

Research 50

Factcards.nl offers all the **information** that you need if you wish to proceed your **career** in the **Netherlands**.

The information is ordered in the categories arriving, living, studying, working and research in the Netherlands and it is freely and easily accessible from your smartphone or desktop.

VISIT FACTCARDS.NL



19.6 Running the System

The packages `c19.net2.*` contain the coding for each of the servers and also for the other processes required by each server. There is also a script `RunUC` that can be used to execute an instance of the Universal Client process. The service servers must be started before the `AccessServer` after which many instances of the User Capability can be executed.

19.7 Commentary

The implementation has some limitations, in that once a Universal Client process has started it has to run to completion before another instance running on a different node can start its interaction even if it has been created. This is not a problem if a location for a group already exists. If the location has not been created then this requires a further interaction which may take some time.

It would be preferable if each user had its own connection to the server. This would necessitate the creation of many request channels in each server, over which they could alternate. This then creates its own problem in that what happens if there are so many requests that there are no request channels left. The server obviously has to manage these channels in a pool but also has to have the ability to inform a user that it does not have sufficient resource to service the request immediately. The interested reader may take this as a challenge and revise the system as described to cater for these additional requirements.

Each of the application areas described in the introduction to this chapter has been implemented as a demonstration. These were implemented using a Personal Data Assistant (PDA) an early form of Smart-phone running a Java Virtual Machine to which the JCSP library had been ported. A more complete version of the Group Meeting system was also implemented in this manner. More recently a proof-of-concept implementation has been undertaken using smart-phone and tablets running various versions of the Android operating system.

20 Redirecting Channels: A Self-Monitoring Process Ring

In this chapter a system is described in which the channels connecting nodes are dynamically changed in response to external stimuli. This is achieved using:

- two agents; one to disconnect and another to reconnect channels
- no centralised control is required; all necessary coding is in the agents
- the agents are created as and when required

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

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

20.1 Architectural Overview

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

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

The `Queue` contains sufficient message slots to hold two messages from each node. The signal to indicate that the receiver has stopped inputting messages is generated by the `StateManager` when the `Queue` is half full. A naive solution would just create an infinite queue to deal with the problem and not worry about the fact that the messages were no longer being processed by the `Receiver`. However, this is not sensible because were the situation to be sustained over a long period the processor would run out of memory and would fail in a disastrous manner. It is thus much better to deal with the situation rather than ignore it. The `Sender` process has been modified in as much that a delay has been introduced so that there is a pause between the sending of one message and the next. This was done so that the operation of the revised system could be more easily observed.



Brain power

By 2020, wind could provide one-tenth of our planet's electricity needs. Already today, SKF's innovative know-how is crucial to running a large proportion of the world's wind turbines.

Up to 25 % of the generating costs relate to maintenance. These can be reduced dramatically thanks to our systems for on-line condition monitoring and automatic lubrication. We help make it more economical to create cleaner, cheaper energy out of thin air.

By sharing our experience, expertise, and creativity, industries can boost performance beyond expectations. Therefore we need the best employees who can meet this challenge!

The Power of Knowledge Engineering

Plug into The Power of Knowledge Engineering.
Visit us at www.skf.com/knowledge

SKF

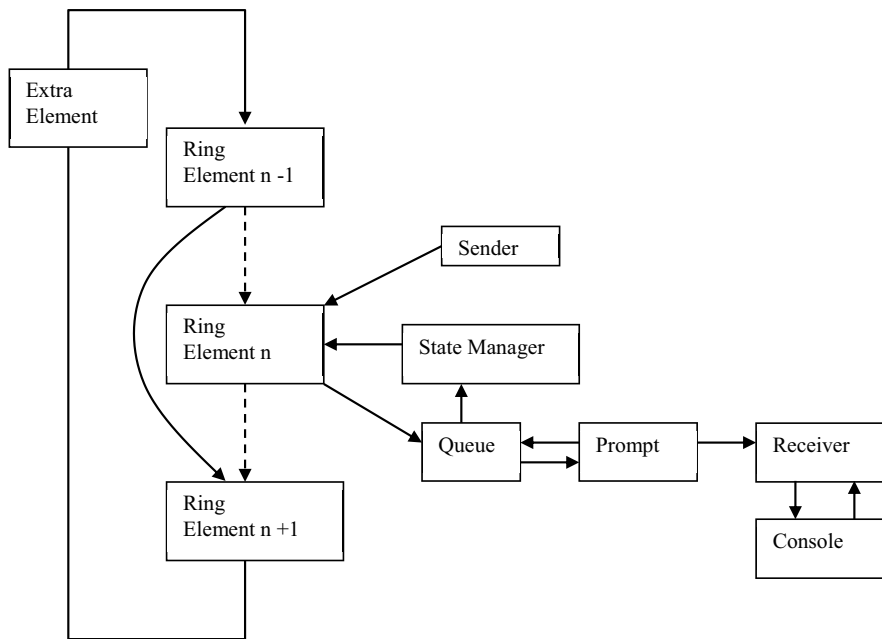


Figure 20-1 The Architecture of the Self Monitoring Ring Structure

20.2 The Receiver process

Listing 20-1 shows the modified `Receiver` process, the behaviour of which is modified by the `Console` process. The channel `fromElement` {12} is the input channel from the `Prompt` process. The remaining channel properties connect the `Receiver` process to the `Console` process. The channel `outChannel` {13} is used to display messages on the `Console` and the channel `fromConsole` {15} is used to input messages from the `Console`. The `clear` channel {14} is used to reset the input area of the `Console`.

The process can receive inputs on either the `fromConsole` or `fromElement` channel and an `ALT` is constructed to this effect {18}. The main loop {21-40} waits for the enabling of an alternative guard {22, 23} and then deals with that input. The expected input from the `Console` {24-34} is either any string to stop the `Receiver` process, typically “stop”, followed by “go” to restart the `Receiver` process. If the input is from the ring element {35-38} then a message is sent to the `Console` writing the content of the data received in the `Console`’s output area. The `Console` process is implemented using the `GConsole` process.

```

10 class Receiver implements CProcess {
11
12     def ChannelInput fromElement
13     def ChannelOutput outChannel
14     def ChannelOutput clear
15     def ChannelInput fromConsole
16

```



```
17 def void run() {
18     def recAlt = new ALT ([ fromConsole, fromElement])
19     def CONSOLE = 0
20     def ELEMENT = 1
21     while (true) {
22         def index = recAlt.priSelect()
23         switch (index) {
24             case CONSOLE:
25                 def state = fromConsole.read()
26                 outChannel.write("\n go to restart")
27                 clear.write("\n")
28                 while (state != "go") {
29                     state = fromConsole.read()
30                     outChannel.write("\n go to restart")
31                     clear.write("\n")
32                 }
33                 outChannel.write("\nresuming ...\n")
34                 break
35             case ELEMENT:
36                 def packet = fromElement.read()
37                 outChannel.write ("Received: " + packet.toString() + "\n")
38                 break
39         }
40     }
41 }
42 }
```

Listing 20-1 The Receiver Process

20.3 The Prompter Process

The Prompter process, shown in Listing 20-2 has channels that communicate with the Queue process as follows. The channel toQueue {12} is used by the Prompter to signal {18} that it is ready to read an item from the Queue. The channel fromQueue {13} is used to input an item from the Queue {19}, which is immediately written to the Receiver process {19} using the toReceiver channel {14}.

```
10 class Prompter implements CSProcess{
11
12     def ChannelOutput toQueue
13     def ChannelInput fromQueue
14     def ChannelOutput toReceiver
15
16     void run() {
17         while (true) {
18             toQueue.write(1)
19             toReceiver.write ( fromQueue.read() )
20         }
21     }
22 }
```

Listing 20-2 The Prompter Process

20.4 The Queue Process

The `Queue` process shown in Listing 20-3, is very similar to that described in Chapter 5.

The main difference is the addition of an output channel `toStateManager` {14} to the properties. The property `slots` {16} is used to specify the number of items that can be held in the queue. The channels `fromPrompter` and `toPrompter` {13, 15} are the respective channel ends of the `Prompter` process' `toQueue` and `fromQueue` channels. The channel `fromElement` {12} is used to receive inputs from the ring element process.

Each time an item is either put into or removed from the queue, depending on the enabled case, the value of the counter {26}, which holds the number of items stored in the `Queue`, is output to the `StateManager` process using the channel `toStateManager` {36, 43}.

```
10 class Queue implements CSPProcess {
11
12     def ChannelInput fromElement
13     def ChannelInput fromPrompter
14     def ChannelOutput toStateManager
15     def ChannelOutput toPrompter
16     def int slots
17
```

Cynthia | AXA Graduate

AXA Global Graduate Program

Find out more and apply

redefining / standards AXA

```
18 void run() {
19     def qAlt = new ALT ( [ fromElement, fromPrompter ] )
20     def preCon = new boolean[2]
21     def ELEMENT = 0
22     def PROMPT = 1
23     preCon[ELEMENT] = true
24     preCon[PROMPT] = false
25     def data = []
26     def counter = 0
27     def front = 0
28     def rear = 0
29     while (true) {
30         def index = qAlt.priSelect(preCon)
31         switch (index) {
32             case ELEMENT:
33                 data[front] = fromElement.read()
34                 front = (front + 1) % slots
35                 counter = counter + 1
36                 toStateManager.write(counter)
37                 break
38             case PROMPT:
39                 fromPrompter.read()
40                 toPrompter.write( data[rear])
41                 rear = (rear + 1) % slots
42                 counter = counter - 1
43                 toStateManager.write(counter)
44                 break
45         }
46         preCon[ELEMENT] = (counter < slots)
47         preCon[PROMPT] = (counter > 0 )
48     } // end while
49 } // end run
50 }
```

Listing 20-3 The Queue Process

20.5 The State Manager Process

The role of the `StateManager` process, shown in Listing 20-4, is to determine how full the `Queue` has become. It inputs the number of items in the `Queue` as the variable `usedSlots` {20} from `Queue` using the channel `fromQueue` {12}. It does this every time the `Queue` process either adds or removes an item. The variable `limit` {17} is half the maximum number of items that can be held in the `Queue`, which is available by means of the property `queueSlots` {14}. The Boolean variable `aboveLimit` determines whether the number of items in the `Queue` is greater than or equal to the `limit` value {21}. The variable `state` {18} indicates the current state of the relationship between `usedSlots` and `limit` and is initially “NORMAL”.

There are two cases of interest; first, when the number of items in the `Queue` is increasing and goes above the `limit` {22–26}, which means the `Queue` is now half full and getting fuller because the `Receiver` has stopped processing items. In this case we need to inform the ring element process of this situation by outputting the “STOP” state on the `toElement` channel {24}. The other case {27–31} occurs when the `Queue` has been above the limit and is now emptying, because the `Receiver` has resumed processing. In this case we reset state to “NORMAL” and write “RESTART” to the `toElement` channel. The coding also includes some print statements {25, 30}, which can be observed in the Eclipse console window for the node as it executes, in addition to the output that appears in the `GConsole` window.

The STOP and RESTART messages have the effect of causing the associated Ring Element process to send a Stop Agent and then a Restart Agent around the network respectively, as described in the next section.

```
10 class StateManager implements CSProcess{
11
12     def ChannelInput fromQueue
13     def ChannelOutput toElement
14     def int queueSlots
15
16     void run() {
17         def limit = queueSlots / 2
18         def state = "NORMAL"
19         while (true) {
20             def usedSlots = fromQueue.read()
21             def aboveLimit = ( usedSlots >= limit)
22             if ((state == "NORMAL") && ( aboveLimit)) {
23                 state = "ABOVE_LIMIT"
24                 toElement.write("STOP")
25                 println "SM: stopping"
26             }
27             if ((state == "ABOVE_LIMIT") && ( !aboveLimit)) {
28                 state = "NORMAL"
29                 toElement.write("RESTART")
30                 println "SM: restarting"
31             }
32         } // end while
33     } // end run
34 }
```

Listing 20-4 The State Manager Process

20.6 The Stop Agent

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

The channel properties `toLocal` {12} and `fromLocal` {13} will be used to connect the `StopAgent` to a host process when it arrives at a new `RingAgentElement` process. These properties are not initialised in its constructor. The property `homeNode` {14} is used to hold the node identity of the `RingAgentElement` that has detected the fault. The property `previousNode` {15} is used to hold the node identity of the `RingAgentElement` that precedes the node that has detected the fault condition. This is the node that will be required to redirect its output channel to the one following the faulty node. The Boolean property `initialised` {16} is used to indicate whether the property `nextNodeInputEnd` {17} has been set. The `nextNodeInputEnd` holds the net channel input end of the channel to which the previous node's net channel output has to be redirected. This can only be obtained once the `StopAgent` has moved from the faulty node to the next node.

The methods `connect` {19–23} and `disconnect` {25–28} are used to initialise and then remove the channel connections between the `StopAgent` and the host process. The `run` method {30–45} initially outputs the properties `homeNode`, `previousNode` and `initialised` to the host process using the channel `toLocal`. If the `StopAgent` has not been initialised it reads the `nextNodeInputEnd` from the host process using the channel `fromLocal`. The host process then sends the `StopAgent` a Boolean `gotThere` indicating whether or not the `StopAgent` has arrived at the node identified by `previousNode`. If the `StopAgent` has arrived then the value of `nextNodeInputEnd` is written to the host process. The `StopAgent` prints messages to the host node's Eclipse console window {31, 38 and 43} to show the state of interactions between the agent and the host node.

TURN TO THE EXPERTS FOR SUBSCRIPTION CONSULTANCY

Subscribe is one of the leading companies in Europe when it comes to innovation and business development within subscription businesses.

We innovate new subscription business models or improve existing ones. We do business reviews of existing subscription businesses and we develop acquisition and retention strategies.

Learn more at [linkedin.com/company/subscribe](https://www.linkedin.com/company/subscribe) or contact
Managing Director Morten Suhr Hansen at mha@subscribe.dk

SUBSCRIBE - to the future



The Stop Agent has to pass through every node. As it does this it informs the Ring Element the identity of the node that has failed [32]. This subsequently has the effect of stopping the node from sending any further messages to the failed node.

An implication of this design is that a network must consist of at least three `RingAgentElement`s and an `AgentExtraElement` process otherwise there would be no possibility of undertaking the redirection. A further implication is that faults cannot be detected in the `AgentExtraElement` process as the system only detects errors once messages have been processed by the `RingAgentElement` process. The system also assumes that the nodes are numbered such that their IP-addresses are numbered consecutively starting with 1 for the `AgentExtraElement` process.

```
10 class StopAgent implements MobileAgent {
11
12     def ChannelOutput toLocal
13     def ChannelInput fromLocal
14     def int homeNode
15     def int previousNode
16     def boolean initialised
17     def NetChannelLocation nextNodeInputEnd
18
19     def connect (c) {
20         this.toLocal = c[0]
21         this.fromLocal = c[1]
22     }
23 }
24
25 def disconnect () {
26     this.toLocal = null
27     this.fromLocal = null
28 }
29
30 void run() {
31     println "SA: running $homeNode, $previousNode, $initialised"
32     toLocal.write(homeNode) // tells node not to send to this node
33     toLocal.write(previousNode) // where we want to get to
34     toLocal.write(initialised)
35     if ( ! initialised) {
36         nextNodeInputEnd = fromLocal.read()
37         initialised = true
38         println "SA: initialised"
39     }
40     def gotThere = fromLocal.read()
41     if ( gotThere ) {
42         toLocal.write(nextNodeInputEnd)
43         println "SA: got to $previousNode"
```

```
44     }  
45 } // end run  
46 }
```

Listing 20-5 The Stop Agent

20.7 The Restart Agent

The `RestartAgent` shown in Listing 20-6 undoes the effect of the `StopAgent` once the faulty node detects that the fault has been removed.

Its properties are very similar and are used in the same way as those with similar names in the `StopAgent`. The `firstHop {16}` property is used to indicate whether the `RestartAgent` has arrived at the node that follows the node that was faulty. This has to be dealt with especially if deadlock is not to occur in the system as will be explained in 20.8.2.2 and 20.8.2.3.

The `run` method of the `RestartAgent` {29–36} simply writes `firstHop` to the host process and then sets `firstHop` `false` if it is `true`. The values of `homeNode` and `previousNode` are then written to the host process, which processes them accordingly. The agent also prints messages {30, 35} to the Eclipse console for the node.

The interactions between the Restart Agent and the node mean that the node can now resume the sending of messages to the previously failed node. This is achieved by the sending of `homeNode` to the local node {33}. The communication of `previousNode` to the local node {34} means that the local node can determine whether it was the previous node and can thus modify its internal structures so that it resumes sending messages to the previously failed node.

```
10 class RestartAgent implements MobileAgent {  
11  
12     def ChannelOutput toLocal  
13     def ChannelInput fromLocal  
14     def int homeNode  
15     def int previousNode  
16     def boolean firstHop  
17  
18     def connect (c) {  
19         this.toLocal = c[0]  
20         this.fromLocal = c[1]  
21  
22     }  
23  
24     def disconnect () {  
25         this.toLocal = null  
26         this.fromLocal = null
```

```
27  }
28
29  void run() {
30      println "RA: running $homeNode, $previousNode"
31      toLocal.write(firstHop)
32      if (firstHop) { firstHop = false }
33      toLocal.write(homeNode) // tells node to resume sending to
    this node
34      toLocal.write(previousNode)
35      println "RA: finished"
36  } // end run
37 }
```

Listing 20-6 The Restart Agent

20.8 The Ring Agent Element Process

The `RingAgentElement` process is somewhat lengthy and thus its description will be subdivided into the sections it comprises. The structure is very similar to that described in Chapter 10 with the modifications required to deal with the agent processing. It should be recalled that the deadlock free architecture developed in Chapter 10 was optimised so that any node that had data to put onto the ring could do so provided an empty packet had been received from the ring. Thus the number of packets circulating around the ring was the same as the number of nodes, excluding the extra element used to overcome one aspect of the deadlock profile of the network.

20.8.1 Properties and Initialisation

The properties and initialisation of the variables used in the operation of the `AgentRingElement` process is shown in Listing 20-7. The channels `fromRing` {12} and `toRing` {13} are net channels used to connect this node to the preceding and following nodes respectively. Messages are received from the `Sender` process on the channel `fromSender` {14}. Similarly outputs of the state of the `Queue` are input from `StateManager` on the `fromStateManager` channel {15}. In this revised version, messages received for the node are now output to the `Queue` process on the `toQueue` {16} channel, rather than directly to the `Receiver` process. The `element` {17} property holds the identity of this node as an integer. The nodes are numbered from 2 upwards, in sequence, with the `AgentExtraElement` given the identity 1.

The `run` method {19}, initially defines the channels and channel ends used to connect this process to any agent {20–25}. The mechanism is exactly the same as that described previously in Chapter 18. The two agents are then constructed {27–32} and the properties that need to be defined are initialised in an obvious manner. The `AgentExtraElement` has also had to be modified to deal with the arrival of agents in a manner very similar to that to be presented. The precise changes for the `AgentExtraElement` will not be described but can be determined from the accompanying folder `ChapterExamples/src/c20/net2`.


```
10 class RingAgentElement implements CSProcess {
11
12     def ChannelInput fromRing
13     def ChannelOutput toRing
14     def ChannelInput fromSender
15     def ChannelInput fromStateManager
16     def ChannelOutput toQueue
17     def int element
18
19     void run() {
20         def N2A = Channel.one2one()
21         def A2N = Channel.one2one()
22         def ChannelInput toAgentInEnd = N2A.in()
23         def ChannelInput fromAgentInEnd = A2N.in()
24         def ChannelOutput toAgentOutEnd = N2A.out()
25         def ChannelOutput fromAgentOutEnd = A2N.out()
26
27         def stopper = new StopAgent ( homeNode: element,
28                                     previousNode: element - 1,
29                                     initialised: false)
30         def restarter = new RestartAgent ( homeNode: element,
31                                           previousNode: element - 1,
32                                           firstHop: true)
33
```

Losing track of your leads?

Bookboon leads the way

Get help to increase the lead generation on your own website. Ask the experts.



Interested in how we can help you?
email ban@bookboon.com



Click on the ad to read more

```
34     def NetChannelLocation originalToRing = toRing.getLocation()
35
36     def failedList = [ ]
37
38     def RING = 0
39     def SENDER= 1
40     def MANAGER = 2
41     def ringAlt = new ALT ( [ fromRing, fromSender, fromStateManager ] )
42     def preCon = new boolean[3]
43     preCon[RING] = true
44     preCon[SENDER] = true
45     preCon[MANAGER] = true // always the case
46     def emptyPacket = new RingPacket(source:-1, destination:-1,
47     value:-1, full: false)
48     def localBuffer = new RingPacket()
49     def localBufferFull = false
50     def restartBuffer = null
51     def restarting = false
52     def stopping = false
53     toRing.write ( emptyPacket )
54     while (true) {
55         def index = ringAlt.select(preCon)
56         switch (index) {
```

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

The variable `originalToRing` {34} is initialised to the `NetChannelLocation` of the `toRing` channel when the process is initialised. If this channel were to be redirected then it is easier to have a record of the original value pre-stored in the process when it comes to reinstating the original connection. The list variable `failedList` {36} will be modified by interaction with the agent to indicate the node(s) that have become faulty. This means that messages destined for a faulty node can be stopped from being sent.

The alternative `ringAlt` {41} has been extended, from that described in Chapter 10, to include the channel `fromStateManager`, so that inputs from the `StateManager` are considered. The related `preCon` element `preCon[MANAGER]` is always `true` {45} as such inputs must always be processed. The variable `restartBuffer` is used when a `RestartAgent` is received {49} and the Boolean variables `restarting` and `stopping` {50, 51} are also used to manage agent processing and their use will be described later.

20.8.2 Dealing With Inputs From The Ring

The main loop of the process essentially deals with incoming packets from the ring, the `Sender` process and the `StateManager` process as indicated by the first part of the loop which selects the enabled alternative from these inputs. The incoming packets from the ring are of three types; data packets, `StopAgents` and `RestartAgents`. Each of these cases will be dealt with separately.

20.8.2.1 Ring Packet Processing

Listing 20-8 shows how RingPackets are dealt with as a result of the selection of the fromRing alternative. The first action is to read the data from the ring {57}. The instanceof operator is then used to determine the type of data that has been read {58}, which in this case is of type RingPacket. It has the same structure as that used in Chapter 10.

```
56     case RING:
57         def ringBuffer = fromRing.read()
58         if ( ringBuffer instanceof RingPacket) {
59             if ( ringBuffer.destination == element ) {
60                 // packet for this node; full should be true
61                 toQueue.write(ringBuffer)
62                 // now write either stopper, restarter, localBuffer or
empty packet to ring
63                 if (stopping) {
64                     stopping = false
65                     toRing.write(stopper)
66                 }
67                 else {
68                     if (restarting) {
69                         restarting = false
70                         toRing.write(restartBuffer)
71                     }
72                     else {
73                         if ( localBufferFull ) {
74                             toRing.write ( localBuffer )
75                             preCon[SENDER] = true // allow another packet
from Sender
76                             localBufferFull = false
77                         }
78                         else {
79                             toRing.write ( emptyPacket )
80                         }
81                     }
82                 }
83             }
84         else {
85             if ( ringBuffer.full ) {
86                 // packet for onward transmission to another element
87                 toRing.write ( ringBuffer )
88             }
89             else {
90                 // have received an empty packet
91                 // write either stopper, restarter, localBuffer or empty
packet to ring
92                 if (stopping) {
93                     stopping = false
```

```
94         toRing.write(stopper)
95     }
96     else {
97         if (restarting) {
98             restarting = false
99             toRing.write(restartBuffer)
100        }
101        else {
102            if ( localBufferFull ) {
103                toRing.write ( localBuffer )
104                preCon[SENDER] = true // allow another
105                packet from Sender
106                localBufferFull = false
107            }
108            else {
109                toRing.write ( emptyPacket )
110            }
111        }
112    }
113 }
114 }
115 else { // dealing with Stop and Restart Agents
```

Listing 20-8 RingPacket Processing



"I studied English for 16 years but...
...I finally learned to speak it in just six lessons"
Jane, Chinese architect

ENGLISH OUT THERE

Click to hear me talking before and after my unique course download



If `ringBuffer {59}` contains a message destined for this node then it is written to the `Queue` process using the `toQueue` channel {61}. The subsequent action taken depends on the state of various Boolean variables. If `stopping` is `true` indicating that the `StateManager` has detected a fault then the `StopAgent stopper` is written to the ring {65}. This action is given the highest priority. If `restarting` is `true` indicating that a `RestartAgent` has been received then the instance saved in `restartBuffer` is written to the ring {70}. If a message has been received from the `Sender` process then it is written to the ring and the `preCon` and variable values are modified so that another message can be received from `Sender` {73–76}. If none of the above conditions are true then an empty packet is written to the ring {79}.

If `ringBuffer {59}` contains a message that is intended for another node then the message is simply written to the ring {87}. The only other possible case is that a message has been received that is the `emptyPacket {89}` and thus the processing required is governed, as before, by the state of the variables associated with, `stopping`, `restarting` and the state of the buffer that holds messages from the `Sender` process. If none of these requires any action then an `emptyPacket` is written to the ring {108}.

20.8.2.2 Stop Agent Processing

The coding associated with `StopAgent` processing is shown in Listing 20-9. The previously read `ringBuffer {57}` is placed in the variable `theAgent {117}`. The agent is then connected to this process and executed {118–120} in the same manner as described in Chapter 18. The interaction with the agent can now commence with the reading of the failed node identity {121}, which can be appended to the `failedList {122}`. A descriptive message is then printed on the node's Eclipse console window {123}. The `targetNode` for the agent can then be read {124} and then the indication of whether the agent has been initialised or not into `alreadyInitialised {125}`. If the agent has not been initialised then the channel location of the input `fromRing` channel can be written to the agent {127}. The `StopAgent` is only initialised when it gets to the node following the node that has failed. The channel location of the `fromRing` channel is required so that it can be sent round the ring to the target node, which is the node preceding the failed node. The target node could be the extra element.

```
116     if (ringBuffer instanceof StopAgent) {
117         def theAgent = ringBuffer
118         theAgent.connect ( [fromAgentOutEnd, toAgentInEnd] )
119         def agentManager = new ProcessManager (theAgent)
120         agentManager.start()
121         def failedNode = fromAgentInEnd.read()
122         failedList << failedNode
123         println "Node $element: stopping failed list now $failedList"
124         def targetNode = fromAgentInEnd.read()
125         def alreadyInitialised = fromAgentInEnd.read()
```

```
126         if ( ! alreadyInitialised ) {
127             toAgentOutEnd.write (fromRing.getLocation())
128         }
129         if (element == targetNode) {
130             // got to node that needs to be changed
131             toAgentOutEnd.write(true)
132             def NetChannelLocation revisedToRing = fromAgentInEnd.read()
133             toRing = NetChannel.any2net(revisedToRing)
134             agentManager.join()
135             theAgent.disconnect()
136             println "Node $element: stopping has redirected toRing"
137             // no need to send agent any further its got to its target
138             // ring has lost a node hence do not send an empty packet
139         }
140         else {
141             toAgentOutEnd.write(false)
142             agentManager.join()
143             theAgent.disconnect()
144             toRing.write(theAgent)
145             println "Node $element: stopping has passed agent on to next node"
146         }
147     }
148     else { // must be instance of RestartAgent
```

Listing 20- 9 Stop Agent Processing

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

If the agent has arrived at the target node {129} then `true` is written to the agent {131} and the channel location to be used for subsequent outputs by this destination node is read from the agent as `revisedToRing` {132}. The channel `toRing` is then assigned the `any2net` channel created from `revisedToRing` {133}. An `any2net` channel has been used so that more than one process can write to the channel, which is required in this situation. When the faulty node detects that it can resume processing because the `Receiver` process has started to accept messages again; a `RestartAgent` will be written on the original `toRing` channel.

The `agentManager` then joins with the agent {134} waiting for the latter to terminate at which point `theAgent` can be disconnected {135}. There is no need to write `theAgent` to the next node as it was the faulty node and we now know that its input, `fromRing`, has been bypassed. More importantly we do not write an `emptyPacket` to the ring even though the agent has taken up an `emptyPacket`, when it was first written to the ring. If we were to write an `emptyPacket` to the ring there would be more packets than nodes and deadlock would ensue.

If `theAgent` has not yet arrived at its destination then processing is much simpler. A false signal is written to the agent {141}. The process then waits for the agent to terminate {142} after which it can disconnect itself from the agent {143}. The process can then write `theAgent` to the ring {144}. A descriptive message is then written {145}.

20.8.2.3 Restart Agent Processing

`RestartAgent` processing is shown in Listing 20-10, which starts with assigning {149} `theAgent` from `ringBuffer` into which it was originally read {57}. In the same manner as before, `theAgent` can be connected to the host process, allocated to an `agentManager` and started {150–152}.



This e-book
is made with
SetaPDF



SETASIGN

PDF components for PHP developers

www.setasign.com



Three values are then read from the agent, `firstHop` {153}, `resumedNode` {154} and `targetNode` {155} using the channel `fromAgent`. As the agent passes through each element it needs to modify the list of failed nodes by removing the identifier of the `resumedNode` from the `failedList` {155}. The target node identity is read {157} into `targetNode` after printing a descriptive message.

If this is the first move made by the agent the `firstHop` will be `true` {158}. In this case the resuming node will have injected the `RestartAgent` into the ring of nodes as an extra packet (see the next section 23.8.3) and this needs to be dealt with specially, if deadlock is not to occur. All the processing between agent and host element has been completed so all that remains is to wait for the agent to terminate and to disconnect it from the host {159, 160}. Instead of writing `theAgent` onto the ring it is placed in a buffer, `restartBuffer` {161} and `restarting` is set `true` {162}. In due course when either an empty packet arrives at the node or a packet is received that is destined for this node the `restartBuffer` will be written to the ring {63–71, 92–100} and `restarting` is reset to `false`.

The remainder of the processing deals with determination of whether or not the agent has arrived at the node, `targetNode`, which needs to have its `toRing` output channel returned to its original setting so that the resumed node is reconnected to the network. This is dealt with by resetting {166} `toRing` to the `originalToRing` value previously saved {34}. The host then waits for the agent to terminate and disconnects itself from the agent. {168–169}. Finally, an `emptyPacket` is written to the ring {172} because the previously faulty node is now operating again. If the agent has not arrived at the `targetNode` it simply waits for `theAgent` to terminate, disconnects itself and then writes `theAgent` to the next node {175–178}.

```
149     def theAgent = ringBuffer
150     theAgent.connect ( [fromAgentOutEnd, toAgentInEnd] )
151     def agentManager = new ProcessManager (theAgent)
152     agentManager.start()
153     def firstHop = fromAgentInEnd.read()
154     def resumedNode = fromAgentInEnd.read()
155     failedList = failedList - [resumedNode]
156     println "Node $element: restarting failed list now $failedList"
157     def targetNode = fromAgentInEnd.read()
158     if (firstHop) {
159         agentManager.join()
160         theAgent.disconnect()
161         restartBuffer = theAgent
162         restarting = true
163     }
164     else {
165         if (element == targetNode) {
166             toRing = NetChannel.any2net (originalToRing)
167             println "Node $element: restarting has redirected toRing"
168             agentManager.join()
```



```
169         theAgent.disconnect()
170         // no need to send agent any further its got to its target
171         // but the node has been reinstated hence need another
           packet on ring
172         toRing.write ( emptyPacket )
173     }
174     else {
175         agentManager.join()
176         theAgent.disconnect()
177         toRing.write(theAgent)
178         println "Node $element: restarting has passed agent on to next node"
179     }
180
181     }
182 }
183 }
184 break
```

Listing 20-10 Restart Agent Processing

The aspect of most importance in the processing of stop and restart agents is to ensure that the correct number of packets remain on the ring. If this is not carefully considered then either the ring will slowly empty of packets as nodes fail and will thus not work optimally, or, more likely, additional packets will be placed on the ring and deadlock will ensue. The design of the ring is inherently prone to deadlock and cannot be analysed using the client server pattern. It is thus up to the system designer to ensure that all possible cases are considered when avoiding the creation of possible deadlocks.

20.8.3 Dealing With Inputs From the Sender Process

Listing 20-11 shows the processing concerned with messages received from the `Sender` process. The `Sender` process writes messages to the ring element node at regular intervals and sends messages to all the other nodes. Messages can only be received when the `localBuffer` is not full and its element in the `preCon` array is true. Once a message packet is received `fromSender` {186}, a test is undertaken to determine whether or not its `destination` is in the `failedList` {188}. If the message is destined for a failed node it is effectively ignored and the control variable `localBufferFull` is not changed. This means that the message will not get written to the ring and will be overwritten by another message. This has the effect that while a node has failed it will receive no messages and for the time that it has failed any messages that it should have received will be lost. This of course is not the most sensible course of action but it does make it much easier to observe the dynamic operation of the system.

```
185     case SENDER:
186         localBuffer = fromSender.read()
187         // test to see if destination is not in failedList
188         if ( ! failedList.contains(localBuffer.destination) ) {
```

```
189         preCon[SENDER] = false // stop any more Sends until buffer
           is emptied
190         localBufferFull = true
191     }
192     // otherwise throw away the message to a stopped node and it
           is lost forever!
193     break
```

Listing 20-11 Sender Process Input Processing

20.8.4 Dealing With Inputs From the StateManager Process

The `StateManager` only generates inputs to the `RingAgentElement` whenever the state of the `Queue` is such that either a ring element needs to be removed from the ring or reinstated. The associated processing is shown in Listing 20-12. If the `fromStateManager` alternative {41} is enabled then the `MANAGER` case is processed {194} in the `switch` specified at {55}. The state is read from `fromStateManager` {196}.

If “STOP” is read then `stopping` is set `true` {199}, which will cause the `StopAgent`, `stopper`, to be output when the next `emptyPacket` is read by the node or a packet arrives that is destined for this node. At this point the node is still operating normally with respect to incoming packets. Its operation will only be modified when it no longer receives packets on its `fromRing` channel because it has been redirected. In fact the faulty node is not modified in any way as its `fromRing` channel is still connected to the previous node. Its `toRing` channel is still connected and will, in due course, be used to output the `RestartAgent`, when it resumes normal operation.

If the value “RESTART” has been read then all that is required is to write {203} the `RestartAgent`, `restarter`, to the ring. This is an additional packet being placed on the ring without there having been a packet received because the node in this case has been disconnected from the ring. This justifies the `firstHop` processing previously described {158} otherwise deadlock would occur.

```
194     case MANAGER:
195         // receiver has restarted
196         def state = fromStateManager.read()
197         if (state == "STOP") {
198             //will write the stopper agent when an empty packet is recieved
199             stopping = true
200             restarting = false
201         }
202         else {
203             toRing.write ( restarter )
204             // this is a stopped node and thus this is the only thing
           it can do
205             // but it will be putting an extra packet onto the ring
```

```
206         // this is why the first Hop processing has to be done if a node
207         // gets a restart agent
208     }
209     break
```

Listing 20-12 StateManager Input Processing

20.9 Running A Node

The easiest way of instantiating the system is to create a script for each node and the extra element that creates the node directly. Such a script for node 2 is shown in Listing 20-13. It assumes that five ordinary nodes will be created, as well as the extra node. The ordinary nodes can be started in any order but the extra element node, which is numbered 1 must be started after the others have been created. The identity of the node is defined {10}, the number of messages that each node is to send {11} and the number of nodes, excluding the extra node, {12} are defined. The required node is created {17-18}, together with the net input channel `fromRing` {19}. The node then waits to input a message on the `fromRing` channel {22}. This can only happen once all the nodes have been created and the Extra Element node has been created last. It is the Extra Element node that initiates the sending of the signal around the network.

```
10 def int nodeId = 2
11 def int sentMessages = 500
12 def int nodes = 5
13
```

The advertisement features a background image of a person running on a path during a sunrise or sunset. The Gaieteye logo is in the top left, with the tagline 'Challenge the way we run'. The main text reads 'EXPERIENCE THE POWER OF FULL ENGAGEMENT...' followed by 'RUN FASTER. RUN LONGER.. RUN EASIER...'. A yellow call-to-action button says 'READ MORE & PRE-ORDER TODAY WWW.GAITEYE.COM'. A hand cursor icon is positioned over the button.

```
14 def nodeIP = "127.0.0.2"
15 def nextNodeIP = "127.0.0.3"
16
17 def nodeAddress = new TCIPNodeAddress(nodeIP, 3000)
18 Node.getInstance().init(nodeAddress)
19 def fromRing = NetChannel.net2one()
20 println "Node $nodeId has been created"
21
22 fromRing.read()
23 def nextNodeAddress = new TCIPNodeAddress(nextNodeIP, 3000)
24 def toRing = NetChannel.any2net(nextNodeAddress, 50)
25 toRing.write(0)
26
27 def processNode = new AgentElement ( fromRing: fromRing,
28                                     toRing: toRing,
29                                     element: nodeId,
30                                     iterations: sentMessages,
31                                     nodes: nodes)
32
33 new PAR ([ processNode]).run()
```

Listing 20–11 Running Node 2

Once the signal has been read the connection to the next node can be created {23–24} and the signal then sent to the next node {25}. Finally, the `AgentElement` process, which creates the network of `RingAgentElement`, `Sender`, `Queue`, `Prompter`, `Receiver` and `StateManager` processes for each node, is constructed {27–31} and run {33}. The ring channels are defined as `any2net` because when a channel is redirected as seen in Figure 20-1 the receiving node has two channel connections on its `fromRing` input.

20.10 Observing The System's Operation

The folder `ChapterExamples/src/c20/net2` contains all the processes and scripts and can be used to create a network of five nodes. Once all the nodes are operating, the console associated with each `Receiver` process will be visible. A `Receiver` process can be made to stop receiving messages by typing “stop” into the input area of the console. It will be observed that the other nodes will continue operation but will not send any messages to the failed node. The failed node can be restarted by typing “go” into the console input area. At which point it will be observed that the messages that have been saved in the `Queue` appear immediately and at some time later messages from the other nodes start to appear. However it can also be observed that the sequence of message values has a gap that corresponds to the time for which the node was not receiving messages.

20.11 Summary

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

20.12 Challenges

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

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

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

The interested reader is invited to address some or all of these challenges.

21 Mobility: Process Discovery

Process discovery is needed in distributed highly parallel systems where each node may not have a complete set of the required processes. This is achieved by:

- defining an agent that can get the required process from another
- being able to incorporate a process dynamically into a node's existing process structure

In this chapter we consider the difference between mobile agents and mobile processes by offering a defining example (Chalmers & Kerridge, 2005). Naming conventions in this area are 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.

wethrive.net

How to retain your top staff
FIND OUT NOW FOR FREE

DO YOU WANT TO KNOW:

- What your staff really want?
- The top issues troubling them?
- How to make staff assessments work for you & them, painlessly?

Get your free trial
Because happy staff get more done



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 data manipulation process, acquire the process, return to its original node and transfer the data manipulation 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 manipulate any further data of the same type.

It can be seen from the above description that the mobile agent has a specific goal that it seeks to achieve. The goal is to find the required data manipulation process. 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 continue to run as part of the node until such time as the node itself is no longer required.

The system architecture is shown in Figure 21-1. The `DataGenerator` process provides a shared 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 shared 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 locations 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.

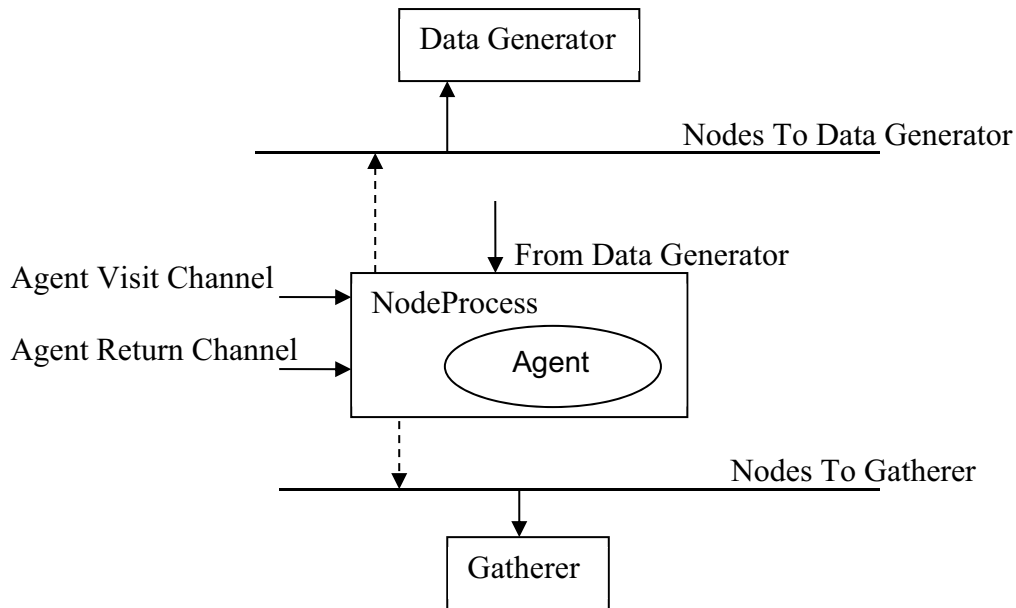


Figure 21-1 Architecture of the Mobile Processes and Agents System

Once these 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 manipulation 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 manipulation process the data is sent to that process where it is modified 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 data manipulation 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 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. A Node also keeps a record on the data manipulation processes for which it has created `Agents`, so that it does not send another `Agent` to search for the same data object type.

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 data manipulation processes in a NodeProcess is invoked using the ProcessManager class. When a process is received by a NodeProcess, from a returning agent, 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.



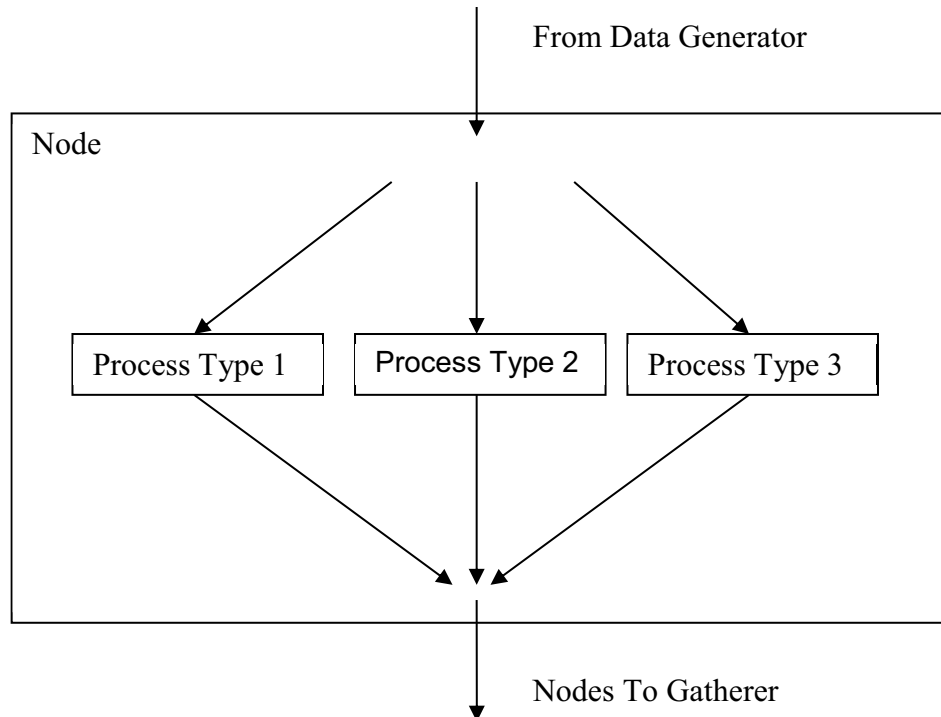


Figure 21-2 The Node Architecture

21.1 The Adaptive Agent

The agent implements the interface described in Chapter 18 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` {12} 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` {13} provides an input from any node the agent visits and similarly `toVisitedNode` {14} provides an output connection to a visited node. Finally, `toReturnedNode` {15} 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` {17–19}. When they are created agents are in the `initial` state, 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 an agent has left the creating node and is attached to another node by the `fromVisitedNode` and `toVisitedNode` channels. Finally, in the `returned` state, the agent has returned to the creating node and is connected to it by the `toReturnedNode` channel.

```
10 class AdaptiveAgent implements MobileAgent, Serializable {
11
12     def ChannelInput fromInitialNode
13     def ChannelInput fromVisitedNode
14     def ChannelOutput toVisitedNode
15     def ChannelOutput toReturnedNode
16
```

```
17  def initial = true
18  def visiting = false
19  def returned = false
20
21  def availableNodes = [ ]
22  def requiredProcess = null
23  def returnLocation
24  def processDefinition = null
25  def homeNode
26
27  def connect ( c) {
28    if (initial) {
29      fromInitialNode = c[0]
30      returnLocation = c[1]
31      homeNode = c[2]
32    }
33    if (visiting) {
34      fromVisitedNode = c[0]
35      toVisitedNode = c[1]
36    }
37    if (returned) {
38      toReturnedNode = c[0]
39    }
40  }
41
42  def disconnect() {
43    fromInitialNode = null
44    fromVisitedNode = null
45    toVisitedNode = null
46    toReturnedNode = null
47  }
48
```

Listing 21-1 The Adaptive Agent Properties and connect Method

The list `availableNodes` {21} 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` {22} will be initialised to the name of the process for which the agent will search. When the agent is initialised the value of `returnLocation` {23} will be set to the net channel location of the node's `agentReturnChannel`. The property `processDefinition` {24} will hold the required data manipulation process' definition once it has been located in a visited node. The value of `homeNode` {25} will be set to the identity of the node from which the agent originates.

The behaviour of the `connect` method {27–40} varies depending upon its state and simply enable the agent to host node local channel connections described previously. 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 {42–47} is always the same and ensures that any channel properties of the agent are set to `null`.

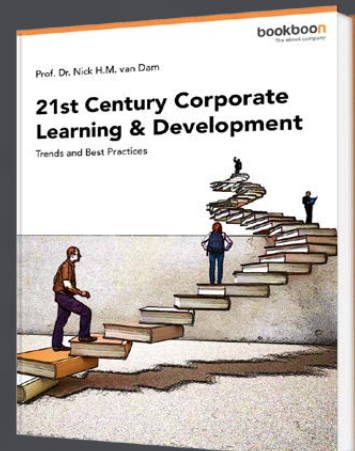
The `run` method of the agent is shown in Listing 21-2 and its behaviour depends upon the state of the agent. It should be noted that in any state the agent always terminates. If the agent is in the `returned` state it writes a list comprising the `processDefinition` and the name of the `requiredProcess` to its `toReturnedNode` channel {50–53}. 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. The Agent prints a message on its home node's console window indicating that it has returned with the required process.

```
49 void run( ) {
50     if (returned) {
51         toReturnedNode.write([processDefinition, requiredProcess])
52         println "AA: returned agent has written $requiredProcess to home node"
53     }
54
55     if (visiting) {
```

Free eBook on Learning & Development

By the Chief Learning Officer of McKinsey

Download Now



Click on the ad to read more

```
56     toVisitedNode.write(requiredProcess)
57     println "AA: visitor wants $requiredProcess"
58     processDefinition = fromVisitedNode.read()
59     println "AA: visitor received $processDefinition"
60     if ( processDefinition != null ) {
61         toVisitedNode.write(homeNode)
62         visiting = false
63         returned = true
64         def nextNodeLocation = returnLocation
65         def nextNodeChannel = NetChannel.any2net(nextNodeLocation)
66         println "AA: visitor being sent home to $nextNodeLocation"
67         disconnect()
68         nextNodeChannel.write(this) // THIS has become NOT serializable!!
69         println "AA: visitor is returning home"
70     }
71     else { //determine next node to visit and go there
72         disconnect()
73         def nextNodeLocation = availableNodes.pop()
74         def nextNodeChannel = NetChannel.any2net(nextNodeLocation)
75         println "AA: visitor continuing journey to $nextNodeLocation"
76         nextNodeChannel.write(this)
77         println "AA: visitor has continued journey"
78     }
79 }
80
81 if (initial) {
82     def awaitingTypeName = true
83     while (awaitingTypeName) {
84         def d = fromInitialNode.read()
85         if ( d instanceof List) {
86             for ( i in 0 ..< d.size) { availableNodes << d[i] }
87         }
88         if ( d instanceof String) {
89             requiredProcess = d
90             awaitingTypeName = false
91             initial = false
92             visiting = true
93             disconnect()
94             //determine next node to visit and go there
95             def nextNodeLocation = availableNodes.pop()
96             def nextNodeChannel = NetChannel.any2net(nextNodeLocation)
97             println "AA: initial going visiting to $nextNodeLocation"
98             nextNodeChannel.write(this)
99             println "AA: initial has been sent to another node"
100         }
101     }
102 }
```

```
103  
104 } // end run  
105 }
```

Listing 21-2 The Adaptive Agent's run Method

If the agent is `visiting` another node {55–79}; it first writes the name of the `requiredProcess` to the visited node {56} and always reads a reply on the `fromVisitedNode` channel {58} into its `processDefinition` property. It also prints an account of the transaction on the visited node's console window {57, 59}. Its subsequent behaviour depends on whether or not the visited node had the required process definition. If the `processDefinition` is not null {60}, the agent writes the identity of the agent's `homeNode` to the visited node {61} 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 {62, 63}. The value of `nextNodeLocation` is set to the agent's `returnLocation` {64} and this is then used to create a `NetChannelOutput` {65}. The agent disconnects from the visited node {67} and then writes itself back to its original node {68}, documenting its progress {66, 69}.

The behaviour of the agent when it has not received a process definition is very similar except that it `pops` the next visiting node location from its list of `availableNodes` {73} and uses that to create a net channel {74} on which it writes itself to the next node on its trip {76}, and again documents its progress {75, 77}.

When the agent is in the `initial` state {81–102} 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 {83–101} by first defining a Boolean `awaitingTypeName` to `true` {82}, which can then be used to control a terminating `while` loop {83–101}. An object is read in from the `fromInitialNode` channel, and its type is determined {85, 88}. If it is a `list` then the agent has received an update to the list of available nodes, which it updates by simply overwriting the previous list {86}. 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 {89} and then the values of `awaitingTypeName` {90}, `initial` {91} and `visiting` {92} are updated. The agent disconnects {93} itself and then causes itself to be written to another node {95–99}, documenting its progress {97, 99}.

21.2 The Node Process

The `NodeProcess`'s properties and definitional part of its `run` method is shown in Listing 21- 3. The `NodeProcess`, unusually, has no channel properties, instead the IP-addresses of the globally available network channels are passed as the parameters, `toGathererIP` {13} and `toDataGenIP` {14}. These are then used to connect to the respective network channels. The `processList` {15} is used to hold any initial data manipulation processes with which the node may be initialised; as further processes are obtained these will be appended to this list. The `vanillaList` {16} contains a possibly empty list of data manipulation 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 manipulation 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` {11} is used to uniquely identify a particular node. The property `nodeIPFinalPart` {12} is used to create the final part of the node's IP-address.

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 manipulation processes. The first requirement is to connect to the `DataGenerator` process. This is achieved by creating the node's IP-address {19} and then its `nodeAddress` and finally, a node instance {21} listening on port 3000. The address of the Data Generator node is created from its IP-address {23}, enabling the creation of an `any2net` channel called `toDataGen` {24}. In a similar manner a connection is made to the `Gatherer` node {25–26}. A net channel to be used by the Data Generator to send data to the node is then created as a `net2one` channel called `fromDataGen` {27}. The next part creates the agent visit and return channels as `net2one` channels {28–29 and 30–31}. The locations of these channels are displayed on the process's console window {32–33}. 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` {35–37}.

```
10 class NodeProcess implements CSProcess {
11     def int nodeId
12     def int nodeIPFinalPart // forms last part of IP address
13     def String toGathererIP
14     def String toDataGenIP
15     def processList = null
16     def vanillaList = null // these must be identical initially
17
18     void run() {
19         def nodeIP = "127.0.0." + nodeIPFinalPart
20         def nodeAddress = new TCPIPNodeAddress(nodeIP, 3000)
21         Node.getInstance().init(nodeAddress)
22     }
```

```
23     def dataGenAddress = new TCPIPNodeAddress(toDataGenIP, 3000)
24     def toDataGen = NetChannel.any2net(dataGenAddress, 50) //50
25     def gathererAddress = new TCPIPNodeAddress(toGathererIP, 3000)
26     def toGatherer = NetChannel.any2net(gathererAddress, 50) //51
27     def fromDataGen = NetChannel.net2one() //52
28     def agentVisitChannel= NetChannel.net2one() //53
29     def agentVisitChannelLocation = agentVisitChannel.getLocation()
30     def agentReturnChannel= NetChannel.net2one() //54
31     def agentReturnChannelLocation = agentReturnChannel.getLocation()
32     println "NP: $nodeId, Visit Channel = $agentVisitChannelLocation"
33     println "NP: $nodeId, Return Channel = $agentReturnChannelLocation"
34
35     toDataGen.write( new DataGenList ( dgl: [ fromDataGen.
36         getLocation(),
37         agentVisitChannelLocation,
38         nodeId] ) )
39
40     def connectChannels = [ ]
41     def typeOrder = [ ]
42     def vanillaOrder = [ ]
43     def currentSearches = [ ]
44     def cp = 0
45
46     if (processList != null) {
47         for ( i in 0 ..< processList.size) {
48             def processType = processList[cp].getClass().getName()
49             def typeName = processType.substring(0, processType.indexOf("Process"))
50             typeOrder << typeName
51             vanillaOrder << typeName
52             connectChannels[cp] = Channel.one2one()
53             def pList = [connectChannels[cp].in(), nodeId, toGatherer.
54                 getLocation()]
55             processList[cp].connect(pList)
56             def pm = new ProcessManager(processList[cp])
57             pm.start()
58             cp = cp + 1
59         }
60     }
61
62     def NodeToInitialAgent = Channel.one2one()
63     def NodeToVisitingAgent = Channel.one2one()
64     def NodeFromVisitingAgent = Channel.one2one()
65     def NodeFromReturningAgent = Channel.one2one()
66
67     def NodeToInitialAgentInEnd = NodeToInitialAgent.in()
68     def NodeToVisitingAgentInEnd = NodeToVisitingAgent.in()
69     def NodeFromVisitingAgentOutEnd = NodeFromVisitingAgent.out()
```



```
68     def NodeFromReturningAgentOutEnd = NodeFromReturningAgent.out()
69
70     def myAgent = new AdaptiveAgent()
71     myAgent.connect([NodeToInitialAgentInEnd,
72                    agentReturnChannelLocation, nodeId])
73     def initialPM = new ProcessManager(myAgent)
74     initialPM.start()
75
76     def nodeAlt = new ALT([fromDataGen, agentVisitChannel, agentReturnChannel])
77     def currentVisitList = [ ]
78
79     while (true) {
80         switch ( nodeAlt.select() ) {
```

Listing 21-3 The Node Process Definitions



Discover the truth at www.deloitte.ca/careers

Deloitte.

© Deloitte & Touche LLP and affiliated entities.



Click on the ad to read more

The list `connectChannels` {39} is used to create a set of internal channels that are used subsequently within the node to connect to the data manipulation processes as they are initialised. The order in which data manipulation 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` {40} and `vanillaOrder` {41} respectively. The list `currentSearches` {42} is used to record the names of the data manipulation processes for which an agent has already been sent. A node can initiate multiple parallel searches for different data manipulation 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` {43} is used to count the number of processes in `processList`.

The next section {45-58} deals with the instantiation of any data manipulation processes in the `processList` that can be ignored if `processList` is `null` {45}. 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 manipulation process {47, 48}. The coding is undertaken for each data process in `processList` {46-57}. The name of a data process is extracted into `processType` {47} using the `getClass` and `getName` methods. The name of the data type is then obtained and placed in `typeName` {48} using the `substring` method and knowledge of the structure of the names as described previously. The `typeName` is then appended to both `typeOrder` {49} and `vanillaOrder` {50}. The channel which connects the `NodeProcess` to the specific data manipulation process now needs to be created and stored in the list `connectChannels` {51}, in the same relative position as `typeName` appears in `typeOrder`. Note that this is an internal `one2one` channel. A list `pList` is now created {52} comprising the `in()` end of this newly created channel, the `nodeId` and the location 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` {53}, which will cause the building of all the required channel connections. An instance of `ProcessManager` is now created {54} with the current element of `processList` as its process and it is then started {55}.

The channels used internally to connect an agent to the `NodeProcess` are now defined {60-63} 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 {65-68}. An instance of the `AdaptiveAgent` is now created as `myAgent` {70} and a call to its `connect` method {71} ensures that `myAgent` can communicate with the `NodeProcess`, and knows its `agentReturnChannelLocation` and its home node identity. The agent is now started using another instance of `ProcessManager` {72-73}.

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` {75}. 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 {76}. The run method enters an infinite loop {78} and switches on the enabled guard of the alternative `nodeAlt` {79}. Each case is dealt with separately.

Listing 21-4 shows the processing when an input from the `DataGenerator` process is received. The data is read into the variable `d` {81} and subsequent processing depends upon the data type. If the input is an instance of `AvailableNodeList` {82} 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` {86} and written to the local agent using the `NodeToInitialAgent` {88} 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 {91} 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` {92} then its position in that list can be determined {93–102}. The data can then be sent to the required data processing process by writing to the corresponding member of the `connectChannels` list {103}.

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 {106}. In this case the type for which the search is being started is appended to `currentSearches` {107} and also written to the agent using the `NodeToInitialAgent` channel {108}. The `join` method is then called {109} 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 {110} and connected {111–112} to the node. It is then passed to a new `ProcessManager` instance {113} and started {114}. The `currentVisitList` is then written to the newly created agent {115}.

```
80         case 0: // data or update to available nodes
81             def d = fromDataGen.read()
82             if ( d instanceof AvailableNodeList ) {
83                 currentVisitList = [ ]
84                 for ( i in 0 ..< d.anl.size) {
85                     if (d.anl[i].toString() != agentVisitChannelLocation.
86                         toString())
87                         currentVisitList << d.anl[i]
88                 }
```

```
88         NodeToInitialAgent.out().write(currentVisitList)
89     }
90     else { // must be a data type name
91         def dType = d.getClass().getName()
92         if ( typeOrder.contains(dType) ) {
93             def i = 0
94             def notFound = true
95             while (notFound) {
96                 if (typeOrder[i] == dType) {
97                     notFound = false
98                 }
99                 else {
100                     i = i + 1
101                 }
102             }
103             connectChannels[i].out().write(d)
104         }
105         else { // do not have process for this data type
106             if ( ! currentSearches.contains(dType)) {
107                 currentSearches << dType
108                 NodeToInitialAgent.out().write(dType)
109                 initialPM.join()
110                 myAgent = new AdaptiveAgent()
111                 myAgent.connect([NodeToInitialAgentInEnd,
112                                 agentReturnChannelLocation,
113                                 nodeId])
114                 initialPM = new ProcessManager(myAgent)
115                 initialPM.start()
116                 NodeToInitialAgent.out().write(currentVisitList)
117             }
118         }
119         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` {121} and then appropriately connected to the node {122–123} by means of the required channel ends. An instance of `ProcessManager` is created {124} to run the visiting agent {125}. The node then reads the type of the data processing process required from the visiting agent into `typeRequired` {126}. If the node contains in the `vanillaOrder` list the `typeRequired` {127} then a search of the `vanillaOrder` is undertaken to find the position of the process {128–140}. The required process can then be obtained from the `vanillaList` of processes and written to the agent using the `NodeToVisitingAgent.out()` channel {138}. The visiting agent then writes its home node identity to the node using the `NodeFromVisitingAgent.in()` channel into `agentHome` {139}. If the node does not have the `typeRequired` in its `vanillaOrder` then a null is written to the visiting agent {142}. The interaction with the visiting agent is now complete so the node process can `join` the agent, waiting for it to terminate {144}, thereby completing the processing of a visiting agent, which either writes itself back to its originating node or the next node in its list of available nodes.

```
120     case 1: // visiting agent has arrived
121         def visitingAgent = agentVisitChannel.read()
122         visitingAgent.connect([NodeToVisitingAgentInEnd,
123                               NodeFromVisitingAgentOutEnd ])
124         def visitPM = new ProcessManager(visitingAgent)
125         visitPM.start()
126         def typeRequired = NodeFromVisitingAgent.in().read()
```



© 2013 Accenture. All rights reserved.

be > your degree

Bring your talent and passion to a global organization at the forefront of business, technology and innovation. Discover how great you can be.

Visit accenture.com/bookboon

Be greater than.
consulting | technology | outsourcing

accenture
High performance. Delivered.

```
127         if ( vanillaOrder.contains(typeRequired) ) {
128             def i = 0
129             def notFound = true
130             while (notFound) {
131                 if (vanillaOrder[i] == typeRequired) {
132                     notFound = false
133                 }
134                 else {
135                     i = i + 1
136                 }
137             }
138             NodeToVisitingAgent.out().write(vanillaList[i])
139             def agentHome = NodeFromVisitingAgent.in().read()
140         }
141         else { // do not have process for this data type
142             NodeToVisitingAgent.out().write(null)
143         }
144         visitPM.join()
145         break
```

Listing 21-5 Node Process: Visiting Agent Processing

The processing associated with a returned agent is shown in Listing 21-6.

```
146         case 2: // agent has returned to originating node
147             def returnAgent = agentReturnChannel.read()
148             returnAgent.connect([NodeFromReturningAgentOutEnd])
149             def returnPM = new ProcessManager (returnAgent)
150             returnPM.start()
151             def returnList = NodeFromReturningAgent.in().read()
152             returnPM.join()
153             def returnedType = returnList[1]
154             currentSearches.remove([returnedType])
155             typeOrder << returnList[1]
156             connectChannels[cp] = Channel.one2one()
157             processList << returnList[0]
158             def pList = [connectChannels[cp].in(), nodeId, toGatherer.
159                 getLocation()]
159             processList[cp].connect(pList)
160             def pm = new ProcessManager(processList[cp])
161             cp = cp + 1
162             pm.start()
163             break
164         } // end switch
165     } // end while true
166 } // end run
```

Listing 21-6 Node Process: Return Agent Processing

The `returnAgent` is read from the `agentReturnChannel` {147} and then connected to the node {148}. An instance of `ProcessManager` is created for the `returnAgent` {149} in which it is started {150}. The `returnList` is obtained from the `returnAgent` using the `NodeFromReturningAgent.in()` channel {151}. This completes the interaction with the return agent and thus the node process can join the `returnPM` awaiting its termination {152}. 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` {154} and appended to the `typeOrder` list {155}. A channel is created that is used to connect the node to the newly acquired data processing process {156} in the `connectChannels` list. The body of the data processing process, returned as `returnList[0]` is appended to the `processList` {157}. A list of properties required for connection is then created as `pList` {158} and connected {159}. An instance of the `ProcessManager` is created {160} and used to start the process {162}. The value of `cp`, which keeps a record of the first empty element in `processList`, is incremented {161}.

21.3 The Data Generator Process

The version of the `DataGenerator` processes presented, in Listing 21-7, 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 also assumes that all the required data processing processes will be available, in some combination, in the first three nodes that are run.

What if you could build your future and create the future?

The innovation accelerator

One generation's transformation is the next's status quo. In the near future, people may soon think it's strange that devices ever had to be "plugged in." To obtain that status, there needs to be "The Shift".

.....Alcatel-Lucent 

www.alcatel-lucent.com/careers



The `ChannelInput fromNodes {11}` is the channel used by all the `NodeProcesses` to communicate with the `DataGenerator` process. The `interval {12}` is used as a timer alarm in an alternative that is selected every cycle to determine whether any new nodes have been registered. This interval subsequently governs the delay between the creation of data objects.

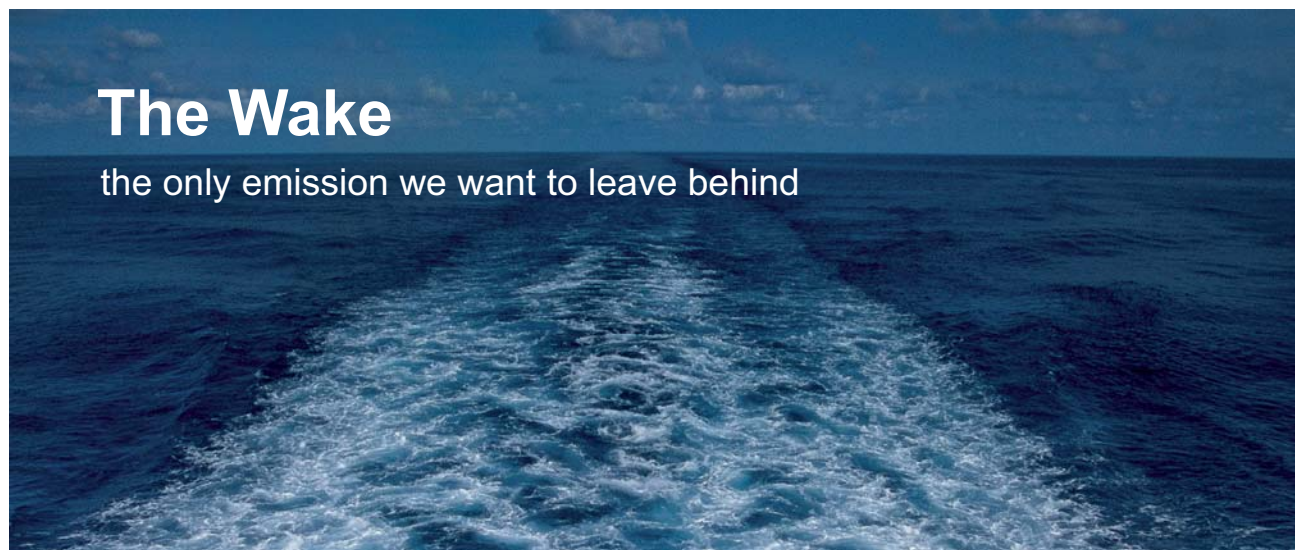
The `ChannelOutputList toNodes {15}` contains the list of net channel output ends used to communicate data to each of the registered `NodeProcesses`. Similarly, the `agentVisitChannelList {16}` maintains a list of all the `agentVisitChannelLocations` sent to the `DataGenerator`. The `allocationList {17}` holds the identity of each registered `NodeProcess`. The variable `rng {18}` is an instance of `Random` and `timer` is an instance of `CSTimer {19}`. The `DataGenerator` process alternates over the `fromNodes` channel and a `timer` alarm as indicated in the `ALT dgAlt {20}`. Each data type has its own associated type instance value {21–23} initialised as shown and there is also a global `instanceValue {24}` 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 {25}`.

The never ending loop {26–69} of the process now starts; by defining and initialising two Boolean variables, `checkingForNewNodes {27}` and `nodeAppended {28}`; and by setting the `timer` alarm {29}. A `while` loop is now entered {30–44} 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. The loop commences with the selection of an enabled guard in the alternative `from dgAlt {31}`.

If a new node is registering itself then data from that node is read from the channel `fromNodes` into `nodeData {33}`. 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 appended to the list `toNodes {34}`. The content of `dgl[1]` is the registering node's `agentVisitChannelLocation` that can be appended to the `agentVisitChannelList {35}`. Finally, `dgl[2]` contains the identity of the registering node and is appended to the `allocationList {36}`. The values of `nodesRegistered` and `nodeAppended` are updated appropriately {37, 38}.

If the alternative selected from `dgAlt` is that corresponding to the `timer` alarm then the value of `checkingForNewNodes` is set `false {41}`. 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 {46–48}`.


```
10 class DataGenerator implements CSProcess {
11     def ChannelInput fromNodes
12     def interval = 1000
13     void run() {
14         def ChannelOutputList toNodes = new ChannelOutputList()
15         def agentVisitChannelList = [ ]
16         def allocationList = [ ]
17         def rng = new Random()
18         def timer = new CTimer()
19         def dgAlt = new ALT ([fromNodes, timer])
20         def type1Instance = 1000
21         def type2Instance = 2000
22         def type3Instance = 3000
23         def instanceValue = 0
24         def nodesRegistered = 0
25         while (true) {
26             def checkingForNewNodes = true
27             def nodeAppended = false
28             timer.setAlarm (timer.read() + interval)
29             while (checkingForNewNodes || (nodesRegistered < 3)){
30                 switch ( dgAlt.select()) {
31                     case 0:
32                         def nodeData = fromNodes.read()
33                         toNodes.append ( NetChannel.one2net ( nodeData.dgl[0] ) )
```



The Wake


the only emission we want to leave behind

Low-speed Engines Medium-speed Engines Turbochargers Propellers Propulsion Packages PrimeServ

The design of eco-friendly marine power and propulsion solutions is crucial for MAN Diesel & Turbo. Power competencies are offered with the world's largest engine programme – having outputs spanning from 450 to 87,220 kW per engine. Get up front! Find out more at www.mandieselturbo.com

Engineering the Future – since 1758.

MAN Diesel & Turbo



```
35         agentVisitChannelList << nodeData.dgl[1]
36         allocationList << nodeData.dgl[2]
37         nodesRegistered = nodesRegistered + 1
38         nodeAppended = true
39         break
40     case 1:
41         checkingForNewNodes = false
42         break
43     }
44 }
45
46 if (nodeAppended) {
47     toNodes.write(new AvailableNodeList ( anl: agentVisitChannelList))
48 }
49 def nNodes = toNodes.size()
50 def nodeId = rng.nextInt(nNodes)
51 switch ( rng.nextInt(3) + 1) {
52     case 1:
53         toNodes[nodeId].write ( new Type1 ( typeInstance:
54                                     type1Instance,
55                                     instanceValue:
56                                         instanceValue ))
57         type1Instance = type1Instance + 1
58         break
59     case 2:
60         toNodes[nodeId].write ( new Type2 ( typeInstance:
61                                     type2Instance,
62                                     instanceValue:
63                                         instanceValue ))
64         type2Instance = type2Instance + 1
65         break
66     case 3:
67         toNodes[nodeId].write ( new Type3 ( typeInstance:
68                                     type3Instance,
69                                     instanceValue:
70                                         instanceValue ))
71         type3Instance = type3Instance + 1
72         break
73     }
74     instanceValue = instanceValue + 1
75 } // end while true
76 } // end run
77 }
```

Listing 21-7 The Data Generator Process

The last part {49–69} 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. In all but the initial phase the value of `interval` {12} will govern the rate at which data objects are generated. The variable `nNodes` {49} is set to the number of nodes in the network. It is then used to choose the `nodeId` {50} to which the data object will be written. The `switch` {51} determines the type identifier of the data type to be generated. It is encoded such 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 {60} as is the global `instanceValue` {68}. 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` {11}, and then prints out the received data `d` {15, 16}. 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.

```
10 class Gatherer implements CSProcess{
11     def ChannelInput fromNodes
12
13     void run() {
14         while (true) {
15             def d = fromNodes.read()
16             println "Gathered from ${d.toString()}"
17         }
18     }
19 }
```

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` {12}. The `modify` {18–21} 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 net would have been removed and that was the purpose of the example.

```
10 class Type1 implements Serializable {
11
12     def typeName = "Type1"
13     def int typeInstance
14     def int instanceValue
15
16     def processedNode
17
18     def modify ( nodeId) {
19         processedNode = nodeId
20         typeInstance = typeInstance + (nodeId *10000)
21     }
22
23     def String toString(){
24         return "Processing Node: $processedNode, Type: $typeName, " +
25             "TypeInstanceValue: $typeInstance, Sequence: $instanceValue"
26     }
27 }
```

Listing 21-9 The Type1 Class Definition

The advertisement features a central graphic on the left with three stylized human figures inside a circular arrangement of four arrows, surrounded by several gears. To the right, the text 'UNLEASHING CHANGE MANAGEMENT' is written in large, bold, blue capital letters. Below this, the dates 'OCTOBER 18 & 19, 2018' and the location 'DE RODE HOED AMSTERDAM' are displayed in smaller blue text. The bottom of the ad shows a silhouette of the Amsterdam skyline, including a windmill and various buildings. In the bottom left corner, the text 'Global Executive Events' is visible.

The data processing processes implements the abstract class `DynamicMobileProcess` 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.

```
10 abstract class DynamicMobileProcess extends MobileProcess
11                                     implements Serializable {
12     abstract connect(x)
13     abstract disconnect()
14 }
```

Listing 21-10 Groovy Mobile Process Abstract Class Definition

Listing 21-11 shows an implementation of `DynamicMobileProcess` for the `Type1` data processing process called `Type1Process`. The `String` property `toGatherer` {12} is the location of the network channel by which this process can connect itself to the net input channel of the `Gatherer` process. The property `inChannel` {13} is used to connect this process to the `NodeProcess`. The property `nodeId` {14} uniquely identifies the node in which this process is executing.

The `connect` {16–20} method is called by the `NodeProcess` 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 `DynamicMobileProcess` `null` {22–24}.

The `run` method {26–33} initially creates the connection to the network channel location in `toGatherer` {27} as `toGathererChannel`. Thereafter the process reads in data from `inChannel` into a variable of defined type {29}, 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 {30}, after which the data is written to the `Gatherer` process {31}. By judicious choice of `nodeId` and `typeInstance` values it is possible to determine the `NodeProcess` to which a particular instance of a data type was sent and processed.

```
10 class Type1Process extends DynamicMobileProcess implements Serializable {
11
12     def toGatherer
13     def ChannelInput inChannel
14     def int nodeId
15
16     def connect (l) {
17         inChannel = l[0]
18         nodeId = l[1]
19         toGatherer = l[2]
20     }
```

```
21
22  def disconnect () {
23    inChannel = null
24  }
25
26  void run() {
27    def toGathererChannel = NetChannel.any2net(toGatherer)
28    while (true) {
29      def Type1 d = inChannel.read()
30      d.modify(nodeId)
31      toGathererChannel.write(d)
32    }
33  }
34
35 }
```

Listing 21-11 The Type1 Process Definition

21.6 Running the System

Three basic 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 IP-addresses of the Data generator and Gatherer node are defined {10, 11}. 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 {13, 14}. An instance of a `NodeProcess` is now defined with all the required property definitions {16–22} and `run` {24}. If a node was being created that had no data type processes available then the lists `pList` and `vList` would be empty.

```
10 def dataGenIP = "127.0.0.1"
11 def gathererIP = "127.0.0.2"
12
13 def pList = [ new Type1Process(), new Type2Process(), new Type3Process() ]
14 def vList = [ new Type1Process(), new Type2Process(), new Type3Process() ]
15
16 def processList = new NodeProcess ( nodeId: 5,
17                                     nodeIPFinalPart: 7,
18                                     toGathererIP: gathererIP,
19                                     toDataGenIP: dataGenIP,
20                                     processList: pList,
21                                     vanillaList: vList
22                                     )
23
24 new PAR ([ processList]).run()
```

Listing 21-12 The Script to Run a Node

Listing 21-13 shows the script that runs an instance of the `DataGenerator` process. The effect of having an interval of 500 milliseconds {16} 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.

```
10 def nodeIP = "127.0.0.1"
11 def nodeAddress = new TCPIPNodeAddress(nodeIP, 3000)
12 Node.getInstance().init(nodeAddress)
13 def fromNodesToGen = NetChannel.net2one() //cn 50
14
15 println "Data Generator Starting"
16 def processList = new DataGenerator ( fromNodes: fromNodesToGen, interval: 500 )
17
18 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.

bookboon.com

Corporate eLibrary

See our Business Solutions for employee learning

[Click here](#)

Management Time Management

Problem solving Self-Confidence Effectiveness

Project Management Goal setting Motivation Coaching

Click on the ad to read more

```
10 def nodeIP = "127.0.0.2"
11 def nodeAddress = new TCPIPNodeAddress(nodeIP, 3000)
12 Node.getInstance().init(nodeAddress)
13 def fromNodesToGatherer = NetChannel.net2one() // cn 50
14
15 println "Gatherer Starting"
16 def processList = new Gatherer ( fromNodes: fromNodesToGatherer )
17
18 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 1, 2 and 3, have been started where the each of these nodes contains the data type processes indicated by their nodeId. It can be observed immediately that the data objects with the instanceValues 0, 2, 3, 4, 6 and 8 have been lost. These have been lost because they contained data that was sent to a node that did not have the required data processing process. The versions of the processes in the accompanying package `ChapterExamples/src/c21/net2` allow printing of created objects which makes it easier to determine exactly what is happening. It also possible to observe agent interactions on the console window of the other processes. The scripts include nodes that have none and all the required type processes.

```
Gatherer Starting
```

```
Gathered from Processing Node: 2, Type: Type2, TypeInstanceValue: 22001, Sequence: 1
Gathered from Processing Node: 2, Type: Type3, TypeInstanceValue: 23001, Sequence: 5
Gathered from Processing Node: 1, Type: Type2, TypeInstanceValue: 12002, Sequence: 7
Gathered from Processing Node: 3, Type: Type3, TypeInstanceValue: 33003, Sequence: 9
Gathered from Processing Node: 3, Type: Type3, TypeInstanceValue: 33004, Sequence: 10
Gathered from Processing Node: 3, Type: Type2, TypeInstanceValue: 32004, Sequence: 11
Gathered from Processing Node: 2, Type: Type2, TypeInstanceValue: 22005, Sequence: 12
Gathered from Processing Node: 3, Type: Type2, TypeInstanceValue: 32006, Sequence: 13
Gathered from Processing Node: 3, Type: Type1, TypeInstanceValue: 31002, Sequence: 14
Gathered from Processing Node: 2, Type: Type3, TypeInstanceValue: 23005, Sequence: 15
Gathered from Processing Node: 2, Type: Type2, TypeInstanceValue: 22007, Sequence: 16
```

Output 21-1 Typical Output From the Gatherer Process

When the dynamics of the system are observed in real-time we see the interactions 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. Thus the nodes and any associated net input channels 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 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

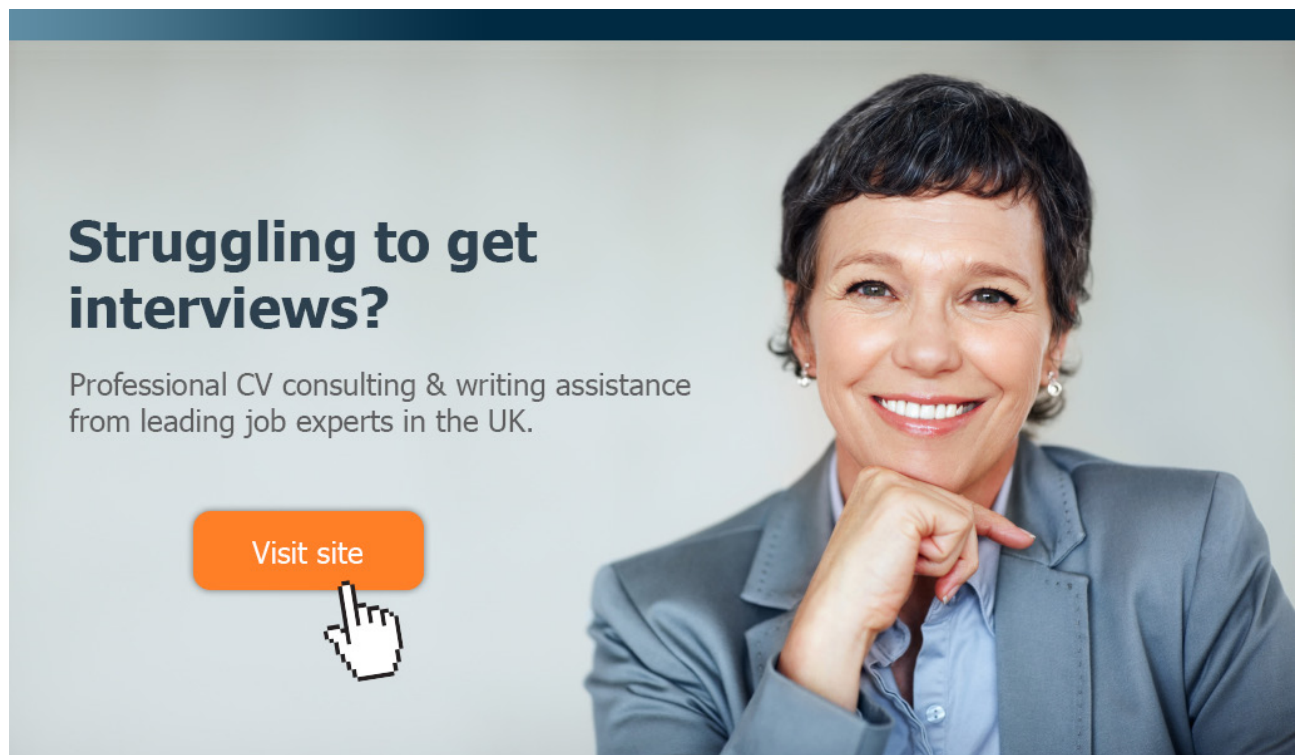
Modify the example so that when a node discovers that the required data processing process is not available the agent is first sent to the `DataGenerator` process to discover all the nodes to which it can go in order to find the resource it requires. In this manner the system does not have to distribute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

22 Automatic Class Loading – Process Farms

Process farms are important because they provide an easy way of generically exploiting parallelism for a wide variety of problems by

- creating Single Instruction Multiple Data (SIMD) data parallel architectures
- creating Multiple Instruction Multiple Data (MIMD) task parallel architectures
- using the ability to dynamically load data object class definitions over a network and
- exploiting the polymorphic capabilities of Java

A common requirement in many parallel processing applications is the need to set up a collection of processes that can subdivide a task into a set of common processes so that work can be distributed over the network so the processing load is shared and thus the overall processing time can be reduced. Such networks are called process farms. Fundamentally, there are two basic forms.



Struggling to get interviews?

Professional CV consulting & writing assistance from leading job experts in the UK.

Visit site



Take a short-cut to your next job!
Improve your interview success rate by 70%.



TheCVagency
Visit theagency.co.uk for more info.



Click on the ad to read more

First, farms where the data can be subdivided into blocks and each block can be independently processed without reference to the other blocks. The farm thus comprises a number of processing nodes each of which undertakes the same process on the block or blocks it is allocated. This is sometimes called data parallelism and yields a processing model referred to as single instruction multiple data (SIMD).

Secondly, farms where the data is subdivided into smaller more manageable blocks but each block has to undergo the same sequence of operations. These individual operations can be captured as a node through which the data is passed. This is referred to as a pipeline of processes because the data passes from one node to the next as if in a pipeline. This is sometimes called task parallelism and yields a processing model referred to as multiple instruction multiple data (MIMD).

Building such farms is a relatively easy task from the architectural point of view as there are a few commonly used architectures. The complexity lies in the fact that the farm has to be specially constructed to deal with the nature of the data being processed. What is required is a mechanism by which the architecture can be built in an application independent manner and then the data and necessary processing can be fed into the network automatically. The JCSP network framework provides precisely this capability. It is possible to load Java classes dynamically over the network of nodes on an as needed basis. Thus we can build a generic architecture into which the specific classes of the application are loaded at run-time.

We shall first describe some common generic farm architectures and then look at one in depth and show how the JCSP class loading capability can be used.

22.1 Data Parallel Architectures

Figure 22-1 shows a basic data parallel architecture in which the Emitter process splits the data into blocks and outputs each block, as a Work Packet, into the stream of Worker processes. Each Worker process outputs its Result packets into a separate channel stream to be recombined by the Collector process.

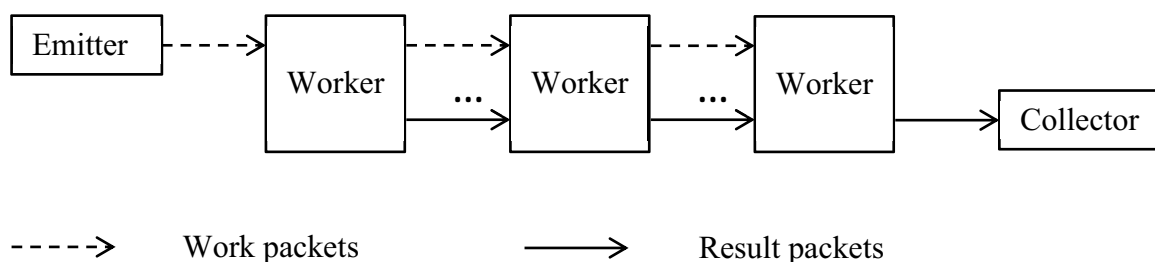


Figure 22-1 A Sequence of Similar Worker Processes

The aim of this architecture is to provide a means whereby a busy Worker process can send a Work packet on to the next Worker immediately. The separation of the Work packets from the Result packets means that the distribution of Work cannot be held up because the Result stream has become blocked by activity downstream or by slowness in the Collector process. In order to achieve this, the Worker process needs an internal architecture that permits this capability. The architecture also needs to overlap Work packet input with task processing and Result packet output. This is a common characteristic of such architectures, regardless of the overall architecture.

Figure 22-2 shows a different architecture in which the Worker processes request Work packets from the Emitter process before sending the Result packets to the Collector process. This architecture will be explored more fully in section 22.3.

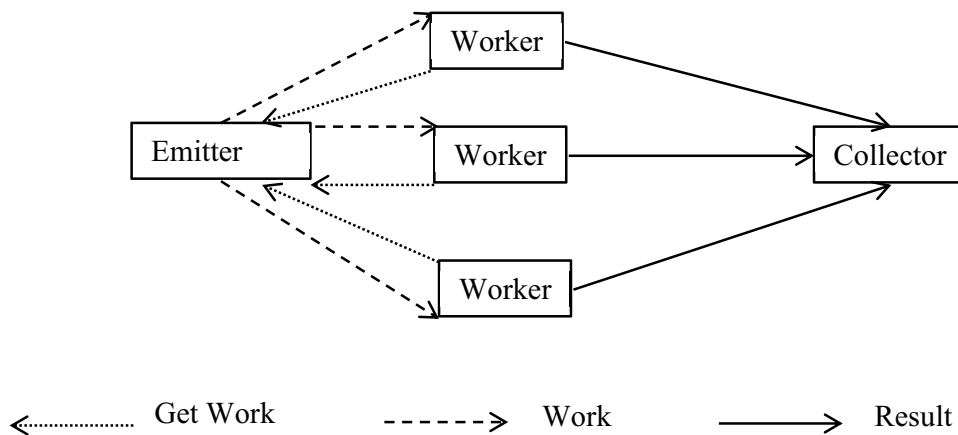


Figure 22-2 An Alternative Data Parallel Architecture

22.1.1 Worker Internal Architecture

The architecture of the Worker process shown in Figure 22-1 is shown in Figure 22-3, which uses the same channel diagramming notation as used in Figure 22-2.

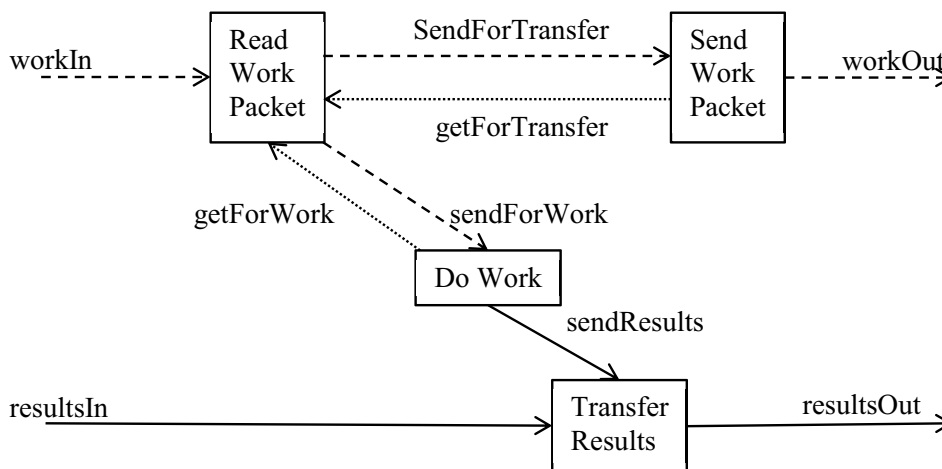


Figure 22-3 Worker Internal Architecture

The processes implement the Client-Server pattern. The process Read Work Packet is a pure server. The processes Send Work Packet and Do Work are pure clients. Only Transfer Results acts as both a Server and a Client and by inspection it can be seen that the architecture contains no cycles even when connected to other Worker processes.

The behaviour of Read Work Packet is shown in Figure 22-4. The process alternates over its two input channels, giving priority to the getForWork channel because we want to ensure the Do Work process is always processing work packets.

```
def rwpAlt = new ALT([getForWork, getForTransfer])
while (true) {
  def packet = workIn.read()
  switch ( rwpAlt.priSelect() ) {
    case 0: //getForWork
      getForWork.read()
      sendForWork.write(packet)
      break
    case 1: //getForTransfer
      getForTransfer.read()
      sendForTransfer.write(packet)
      break
  }
}
```

Figure 22-4 Read Work Packet Behaviour

e-learning for kids

- The number 1 MOOC for Primary Education
- Free Digital Learning for Children 5-12
- 15 Million Children Reached

About e-Learning for Kids Established in 2004, e-Learning for Kids is a global nonprofit foundation dedicated to fun and free learning on the Internet for children ages 5 - 12 with courses in math, science, language arts, computers, health and environmental skills. Since 2005, more than 15 million children in over 190 countries have benefitted from eLessons provided by EFK! An all-volunteer staff consists of education and e-learning experts and business professionals from around the world committed to making difference. eLearning for Kids is actively seeking funding, volunteers, sponsors and courseware developers; get involved! For more information, please visit www.e-learningforkids.org.



The process repeatedly reads in a packet from the workIn channel then depending upon which alternative is selected either sends the packet to the Do Work process (case 0) or to the Send Work Packet process (case 1). The process essentially provides a one place buffer in which to hold the incoming work packet. The Send Work Packet process repeatedly sends a signal on the getForTransfer channel and then reads the work packet on the sendForTransfer channel, which it then outputs on the workOut channel. This interaction will only take place when the Do Work process is busy and has not sent a signal on the getForWork channel.

The behaviour of the Do Work process is shown in Figure 22-5. The process repeatedly sends a signal on the getForWork channel. It then reads the work packet from the sendForWork channel. The packet is then processed, signified by a call to the manipulate() method, after which the produced result packet is written to the sendResults channel.

```
while (true) {
    getForWork.write(1)
    def packet = sendForWork.read()
    def resultsPacket = packet.Manipulate() // process the packet
    sendResults.write(resultsPacket)
}
```

Figure 22-5 Do Work Behaviour

The behaviour of the Transfer Results process is shown in Figure 22-6. The process alternates over its two input channels giving priority to the sendResults channel because we wish to empty the Do Work process as quickly as possible so that it can process another work packet immediately.

```
def trAlt = new Alt ([sendResults, resultsIn])
while (true) {
    switch ( trAlt.priSelect() ){
        case 0: //sendResults
            resultsOut.write( sendResults.read() )
            break
        case 1: //resultsIn
            resultsOut.write( resultsIn.read() )
            break
    }
}
```

Figure 22-6 Behaviour of the Transfer Results Process

22.2 Task Parallel Architectures

The usual architecture adopted by task parallel applications is a pipeline of processes as shown in Figure 22-7. It is assumed that each data block has to be processed by each of the task processes in sequence.



Figure 22-7 Basic Task Parallel Architecture

The Emitter process outputs blocks of data each of which is processed in sequence by the processes Task A, Task B and Task C, finally being output to the Collector process which recombines the data as necessary. For this architecture to be effective it is crucial that each Task in the pipeline overlaps input, work and output as shown in the previous section. For a pipeline to be effective the time to process each task should be similar and there should be no requirement to send any work backwards through the pipeline. If this is achieved then it will take 3 task units for the first data to appear at the input of the Collector but thereafter subsequent data will appear every task unit.

In the architectures described above performance improvement can only occur if all the processes are placed on different cores or nodes in a network of processes. Thus we need to consider how this can be achieved. The JCSP net2 architecture provides support that makes this task easier.

22.3 Generic Architectures

Given the relative simplicity of the architectures previously described it would be beneficial if the designer could implement a set of basic architectural patterns. The Java `interface` mechanism provides a means whereby we can make the processing polymorphic. However that only solves part of the problem. If each process is placed on a node in a network then each node requires knowledge of both the abstract and concrete classes used in the application. Distributing this class information could be problematic over a large network, especially if it is constructed using a network of workstations, normally used as standalone workstations.

The JCSP net2 architecture has two additional capabilities over and above the ability to communicate serialized objects (Chalmers, 2009). The first is the ability to provide a filter mechanism. This mechanism allows the system designer to transform a complex class data structure into a byte array, which can then be communicated over a network channel. Two filters are required one at the transmitting end to undertake the conversion from structure into byte array. The other at the receiving end converts the byte array back into the required structure. The advantage of this filter mechanism is that the serialization of a byte array accounts for very little additional overhead in comparison to the built-in Java serialization mechanism. Thus if large amounts of complex data are being transferred over a network the use of these filters can substantially improve communication performance.

The other mechanism is a class loading capability. If one process in a node of a network sends a class to another node and the receiving node does not have the class definition, then it can dynamically request the class definition from the sending node. If a class references yet other classes within it, then these internal classes are obtained on an as needed basis. Once a node has received a class or classes in this manner it is able to transmit these class definitions to other nodes that also require the class definitions. A node can only obtain a class definition if there is a direct channel connection from the requesting node to a previous node that has the class definition, albeit through other intermediary nodes. Implicitly, such class loading channels create a reverse channel by which requests for class definitions can be communicated. The mechanism fails if the class definition cannot be found on any node for which one of these back channels exists. A general search for a class definition is not implemented.

22.4 Architectural Implementation

The discussion will use the architecture shown in Figure 22-2. In the accompanying project `ChapterExamples/src/` the code for `c22` is split over a number of packages, thereby ensuring that classes from one package are not accessible to other packages. One package contains some Universal Classes that are needed by all nodes. The package `c22.UniversalClasses` contains class definitions for `objects`; `Sentinel`, `Signal` and `InitObject`. The `InitObject` is used to communicate a node identity and a net channel address. The description will present the components of the architecture in the order `Collector`, `Emitter` and finally `Worker`.

FACTCARDS

Are you working in academia, research or science? And have you ever thought about working and moving to the Netherlands?

Arriving 33

Living 50

Studying 51

Working 101

Research 50

Factcards.nl offers all the **information** that you need if you wish to proceed your **career** in the **Netherlands**.

The information is ordered in the categories arriving, living, studying, working and research in the Netherlands and it is freely and easily accessible from your smartphone or desktop.

VISIT FACTCARDS.NL



22.4.1 Collector

The script to run the Collector process is shown in Listing 22-1. The node of the Collector process is fixed as 127.0.0.2 and the node listens on port 3000 {10}. A node instance is created {11} and the IP-address of the node printed using the `getIpAddress()` method {12}. This value will be needed when each of the Worker nodes is created.

The input net channel to the node is then created as the `net2one` channel `fromWorkers` {14}. The channel is created as a code loading channel by means of the creation of the instance of `CodeLoadingChannelFilter.FilterRX()`, which builds the mechanism by which the Collector can dynamically receive class definitions from a previous node. The Collector node only has net input channels so does not have the means to communicate back to a previous (Worker) node. The `CodeLoadingChannelFilter` builds the required channel in the reverse direction automatically and transparently. The Collector process cannot access this back channel. For information, the location of the `fromWorkers` net channel is printed {16,17}. The number of Worker nodes is then obtained by a console interaction as `workers` {19}. This is required because we are going to make the system terminate once all the data has been processed. Finally, an instance of the Collector process is created and run {22–25}.

```
10 def nodeAddr = new TCPIPNodeAddress("127.0.0.2", 3000)
11 Node.getInstance().init (nodeAddr)
12 println "Collector IP address = ${nodeAddr.getIpAddress()}"
13
14 def fromWorkers = NetChannel.net2one(new CodeLoadingChannelFilter.FilterRX())
15
16 def fromWorkersLoc = fromWorkers.getLocation()
17 println "Collector: from Workers channel Location - ${fromWorkersLoc.toString()}"
18
19 def workers = Ask.Int ("Number of workers? ", 1, 20)
20
21
22 def collector = new Collector ( fromWorkers: fromWorkers,
23                               workers: workers)
24
25 new PAR([collector]).run()
```

Listing 22-1 The Script to Run the Collector Node

The structure of the Collector process is shown in Listing 22-1.

```
10 class Collector implements CSProcess {
11
12     def fromWorkers
13     def workers = 2
14
15     void run(){
16         def timer = new CTimer()
```

```
17     def terminated = 0
18     def stopped = false
19     def now = 0
20     def start
21     def first = true
22     def results = []
23     while (!stopped) {
24         def o = fromWorkers.read()
25         if (first) {
26             start = timer.read()
27             first = false
28         }
29         if (o instanceof Sentinel) {
30             terminated = terminated + 1
31             stopped = terminated == workers
32         }
33         else {
34             now = timer.read()
35             results << o.display(now)
36         }
37     }
38     def end = timer.read()
39     def l = 1
40     for ( line in results) {
41         println "line: $l at \t$line"
42         l = l + 1
43     }
44     println "elapsed time: ${end - start} msecs; processed ${results.
45         size()} results"
46 }
```

Listing 22-2 The Collector Process Definition

The property `fromWorkers` {12} is the net input channel to the process and the default number of worker nodes is 2 {13}. The process uses a `timer` {16} to time the arrival of each processed data object and also the overall processing time. The variables `terminated` and `stopped` are used when determining if the process can stop once all the worker nodes have terminated {17, 18}. The list `results` {22} will be used to hold all the processed data objects that are received. The `results` will only be printed once they have all been received otherwise the elapsed processing time will include the printing time, which will therefore not truly reflect the change in processing time by adding additional worker nodes.

The main loop {23–45} is controlled by the value of `stopped`. The process reads objects from the `fromWorkers` channel {24}. If this is the first such object the `start` time is recorded {26}. If the object is an instance of `Sentinel`, then this indicates that the `Worker` that sent it has terminated. In which case the count of `terminated` workers can be incremented and if the number of terminated workers is the same as the number of `workers` then the value of `stopped` can be set true {29–32}. This will cause the process to enter the terminating phase on the next iteration.

If the object is a data object then the process reads the current time into `now` {34} and calls the `display()` method of the object {35}. At the creation of this node the process does not have the class definition for the data object. Thus it will make a request to the first `Worker` that sends it a data object for the required class definition. It is assumed that the data object implements an `interface` that has `display` as one of its `abstract` methods.

Once all the worker nodes have terminated the process reads the end time {38}, prints the results {39–43} and the total elapsed time {44} before itself terminating.



Brain power

By 2020, wind could provide one-tenth of our planet's electricity needs. Already today, SKF's innovative know-how is crucial to running a large proportion of the world's wind turbines.

Up to 25 % of the generating costs relate to maintenance. These can be reduced dramatically thanks to our systems for on-line condition monitoring and automatic lubrication. We help make it more economical to create cleaner, cheaper energy out of thin air.

By sharing our experience, expertise, and creativity, industries can boost performance beyond expectations. Therefore we need the best employees who can meet this challenge!

The Power of Knowledge Engineering

Plug into The Power of Knowledge Engineering.
Visit us at www.skf.com/knowledge

SKF

22.4.2 Emitter

The script to create the `Emitter` node is shown in Listing 22-3. The script initially creates a node instance and prints out its IP-address, as this will be required when each worker node is created {10–12}. A `net2one` input channel is created, `fromWorkers` and its location printed for information {14–17}. Both the `Emitter` and `Collector` nodes behave as pure servers and the worker nodes behave as pure clients.

```
10 def nodeAddr = new TCPIPNodeAddress("127.0.0.1",3000)
11 Node.getInstance().init (nodeAddr)
12 println "Emitter IP address = ${nodeAddr.getIpAddress()}"
13
14 def fromWorkers = NetChannel.net2one()
15
16 def fromWorkersLoc = fromWorkers.getLocation()
17 println "Emitter: from Workers channel Location - ${fromWorkersLoc.toString()}"
18
19 def workers = Ask.Int ("Number of workers? ", 1, 17)
20 def loops = Ask.Int ("Number of data objects to send? ", 1, 1000000)
21 def elements = Ask.Int ("Number of elements in each TestObject? ", 1, 200)
22
23
24 def emit = new EmitterNet ( fromWorkers: fromWorkers,
25                             loops: loops,
26                             workers: workers,
27                             elements: elements )
28 new PAR([emit]).run()
```

Listing 22-3 The Emitter Node Creation Script

The number of workers and the parameters used in the creation of the data objects are then obtained by user interaction {19–21}. An instance of the `Emitter` node is then created and run {24–28}.

The data object used in the example is called `TestObject` and its structure is shown in Listing 22-4. It has four properties {12–15}; `workerId` will indicate the worker node that processed the particular object instance. The property `sum` will be used to add up all the elements in the list `data`. Finally, `dataSize`, is the number of elements in `data`.

The object has an explicit constructor {17–20}, which initialises the list `data` and the property `dataSize`.

```
10 class TestObject implements ManipulateInterface {
11
12     def workerId = -1
13     def sum = 0
14     def data = []
15     def dataSize = 0
16 }
```

```
17  def TestObject (elements, m) {
18      for ( i in 0..<elements) data[i] = (i * (m+1)) + 1
19      dataSize = elements
20  }
21
22  def manipulate (x){
23      for ( i in 0..<dataSize) data[i] = data[i] * (x + 1)
24      for ( i in 0..<dataSize) sum = sum + data[i]
25      workerId = x
26  }
27
28  def String display (now){
29      def s = "$now: from - $workerId data = $data, $sum"
30      return s
31  }
32
33 }
```

Listing 22-4 The TestObject Class Definition

`TestObject` implements the `ManipulateInterface` that has two abstract methods, one used to manipulate the properties of the object {22–26}, which will be called in a worker node and the other `display` {28–31}, used to print the object to the console window, which we have seen already is called in the `Collector` node. The `TestObject` has been designed so that it is relatively easy to create data objects of different sizes. The `manipulate` method iterates through the data twice thereby creating a significant processing requirement if the object has a large number of elements.

Listing 22-5 gives the definition of the `EmitterNet` process. The `EmitterNet` process comprises five separate phases as follows.

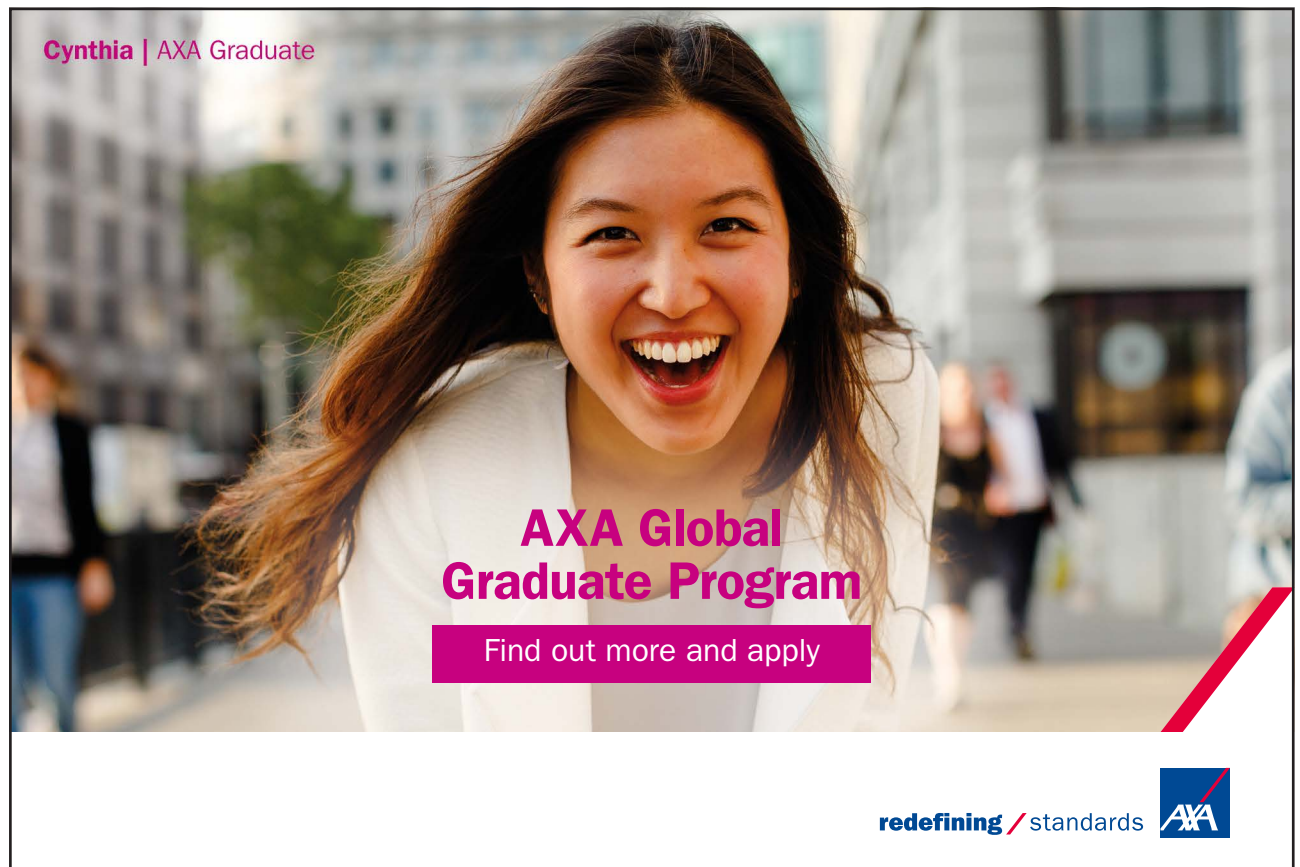
The first phase creates the data object using the properties `loops` and `elements` {23–27}. This will result in the data being created before any subsequent processing. This means that we can evaluate the effect of adding more worker nodes on the actual processing time without the influence of the time it takes to actually create the data objects.

The second phase {29–34} inputs an `InitObject` from each worker node. The `InitObject` contains the `id` of the node which is used as the key in the map `netLocations` {21}. The value stored in the map is a `one2net` channel, which has been created by a worker node, the location of which has been sent in the `InitObject` as the property `channelAddress`. This must be created as a `CodeLoadingChannelFilter` {32} channel because the `Emitter` will use this channel to send the `TestObject` class definition to each `Worker` node.

During the third phase the Emitter node sends a synchronisation signal to each worker node informing them that they can now start requesting data objects from the emitter node {35–41}. The net output channel to be used to access each worker is obtained from the `netLocations` map {38}.

The fourth phase results in the data objects being sent to the Worker nodes as each makes a request for a data object to be sent {42–47}. A Worker requests work by sending a `Signal` object, which contains the identity of the requesting node. The `workerId` {44} is used to access the `netLocations` map {45} to obtain the required output channel location, which is then used to write the next data object to the Worker node {46}.

```
10 class EmitterNet implements CProcess {
11
12     def fromWorkers
13     def toWorkers
14     def loops = 10
15     def workers = 2
16     def elements = 5
17
18     void run(){
19         def data = []
20         def workerId = 0
21         def netLocations = [:]
```



Cynthia | AXA Graduate

AXA Global Graduate Program

Find out more and apply

redefining / standards AXA

```
22     // create the data
23     for ( i in 0 ..< loops) {
24         data[i] = new TestObject (elements, i)
25     }
26     println "Emitter: Data Generated, Loops: $loops, " +
27             "Elements per object: $elements for $workers Workers"
28     // receive an InitObject from each worker
29     for ( i in 1..workers) {
30         def initLoc = (InitObject)fromWorkers.read()
31         netLocations.put (initLoc.id, NetChannel.one2net(initLoc.
32             channelAddress, 50,
33                 new CodeLoadingChannelFilter.FilterTX()))
34     }
35     println "Emitter: $workers Workers have registered"
36     // send each worker a synchronisation signal
37     def channelLoc = null
38     for ( i in 0 ..< workers) {
39         channelLoc = netLocations.get (i)
40         channelLoc.write(new Signal())
41     }
42     println "Emitter: $workers Workers have synchronised"
43     // wait for a request from a worker and then send them
44     for ( i in 0 ..< loops) {
45         workerId = ((Signal)fromWorkers.read()).signal
46         channelLoc = netLocations.get (workerId)
47         channelLoc.write(data[i])
48     }
49     // terminate each of the workers
50     for ( i in 1..workers) {
51         workerId = ((Signal)fromWorkers.read()).signal
52         channelLoc = netLocations.get (workerId)
53         channelLoc.write(new Sentinel())
54     }
55 }
```

Listing 22-5 The EmitterNet Process Definition

The final phase {49–53} occurs once all the data has been processed. Each worker will send a request for another data object but instead a `Sentinel` object is returned, thereby informing the worker node that it should itself terminate.

22.4.3 Worker

The behaviour presented in 22.1.1 would result in the copying of possibly large amounts of data from one process to another within a worker node. This could result in an excessive processing overhead for large complex data objects. To avoid this we shall adopt a strategy of using three data buffers that are shared between the internal processes that make up a worker node. The internal structure of a worker node is shown in Figure 22-8.

The sharedData buffers will be accessed in rotation by each of the processes. Thus as SendOutput is writing a data object to the toCollector channel from buffer 2, the Worker process will be processing the data object in buffer 1 and the GetInput process will be reading a data object into buffer 0. The value that will be passed from one process to the next will be the subscript of the shared buffer. As each process only has access to one of the shared buffers at any instant it is not possible for two of the processes to access the same buffer at the same time.

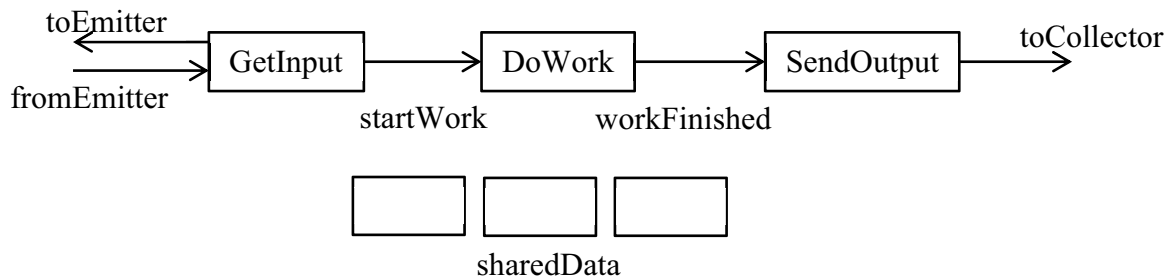


Figure 22-8 The Internal Process Structure of a Worker Node

The script that creates a Worker node is shown in Listing 22-6, which assumes that the Emitter and Collector nodes have been previously created.

```

10 def w = Ask.Int ("Worker Number ( 3 upwards) ? ", 3, 20)
11 def emitterIP = Ask.string("Emitter Process IP? ")
12 def collectorIP = Ask.string("Collector Process IP? ")
13
14 def addr = new TCIPNodeAddress ("127.0.0." + w, 3000)
15 Node.getInstance().init (addr)
16 println "Worker IP address = ${addr.getIpAddress()}"
17
18 def fromEmitter = NetChannel.net2one(new CodeLoadingChannelFilter.FilterRX())
19 def fromEmitterLoc = fromEmitter.getLocation()
20 println "Worker: from Emitter channel Location - ${fromEmitterLoc.toString()}"
21
22 def toEmitterAddr = new TCIPNodeAddress ( emitterIP, 3000)
23 def toEmitter = NetChannel.any2net(toEmitterAddr, 50 )
24
25 def toCollectorAddr = new TCIPNodeAddress ( collectorIP, 3000)
  
```



```
26 def toCollector = NetChannel.any2net(toCollectorAddr, 50,  
27   new CodeLoadingChannelFilter.FilterTX())  
28  
29 def base = new Worker ( toEmitter: toEmitter,  
30                        fromEmitterLoc: fromEmitterLoc,  
31                        fromEmitter: fromEmitter,  
32                        toCollector: toCollector,  
33                        baseId: w - 3 )  
34 new PAR([base]).run()
```

Listing 22-6 The Script that Creates a Worker Node

The initial user interaction establishes the identity w of this worker and the IP-addresses of the emitter and collector nodes {10–12}. The worker node instance is then created and details printed {14–16}. The script then creates a `net2one` channel, `fromEmitter`, which has to be a `CodeLoadingChannelFilter` because it is used to send class definitions from the Emitter to the Worker {18}. The location of the channel is obtained as this will need to be sent to the Emitter node {19} and this location is printed for information {20}. The Worker can now create the output channels to the Emitter and Collector nodes {22–27}. The `toEmitter` channel does not need to be a code loading channel but that to the Collector, `toCollector`, must be {27} because it will be used to transfer the data object class definition to the Collector node. The `Worker` process can now be constructed and run {29–34}.

The definition of the Worker process is shown in Listing 22-7. Initially, the `Worker` writes an `InitObject` to the Emitter process using the `toEmitter` channel {19}. It then creates the internal channels `startWork` and `workFinished` as `one2one` channels {20, 21} see also Figure 22-8. The `sharedData` buffers are then defined {22}. The process then reads a synchronisation signal from the Emitter process {23} and prints an information message {24}. Each of the internal processes are now constructed {25–37} and run. Finally, a message is printed to indicate the process has terminated.

```
10 class Worker implements CSProcess {  
11  
12   def toEmitter  
13   def fromEmitterLoc  
14   def fromEmitter  
15   def toCollector  
16   def baseId  
17  
18   void run() {  
19     toEmitter.write(new InitObject(id: baseId, channelAddress: fromEmitterLoc))  
20     def startWork = Channel.one2one()  
21     def workFinished = Channel.one2one()  
22     def sharedData = []  
23     def sync = (Signal)fromEmitter.read() //synchronisation signal  
24     println "Worker: $baseId initialised and about to run internal processes"
```

```
25     def getter = new GetInput ( toEmitter: toEmitter,  
26                               baseId: baseId,  
27                               fromEmitter: fromEmitter,  
28                               toWorker: startWork.out(),  
29                               sharedData: sharedData )  
30     def worker = new DoWork ( workOn: startWork.in(),  
31                               workCompleted: workFinished.out(),  
32                               workerId: baseId,  
33                               sharedData: sharedData )  
34     def putter = new SendOutput ( workerFinished: workFinished.in(),  
35                                   toCollector: toCollector,  
36                                   sharedData: sharedData)  
37     new PAR([getter, worker, putter]).run()  
38     println "Worker: $baseId terminated"  
39 }  
40  
41 }
```

Listing 22-7 The Worker Process definition

The `GetInput` process definition is shown in Listing 22-8. The property `sharedData {16}` is shared between each of the internal processes and if this was not carefully managed then disaster could result. The variable `index {19}` is used to identify which `sharedData` buffer the process is currently allowed to access.

TURN TO THE EXPERTS FOR SUBSCRIPTION CONSULTANCY

Subscribe is one of the leading companies in Europe when it comes to innovation and business development within subscription businesses.

We innovate new subscription business models or improve existing ones. We do business reviews of existing subscription businesses and we develop acquisition and retention strategies.

Learn more at [linkedin.com/company/subscribe](https://www.linkedin.com/company/subscribe) or contact
Managing Director Morten Suhr Hansen at mha@subscribe.dk

SUBSCRIB✓**BE** - to the future

```
10 class GetInput implements CSProcess {
11
12     def toEmitter
13     def baseId
14     def fromEmitter
15     def toWorker
16     def sharedData
17
18     void run(){
19         def index = 0
20         def running = true
21         while (running) {
22             toEmitter.write(new Signal(signal: baseId))
23             def o = fromEmitter.read()
24             if ( o instanceof Sentinel){
25                 running = false
26                 toWorker.write(o)
27             }
28             else {
29                 sharedData[index] = o
30                 toWorker.write(index)
31                 index = (index + 1)%3
32             }
33         }
34     }
35 }
```

Listing 22-8 The GetInput Process Definition

In the loop of the process {21–33} the first action is to send a signal {22} to the Emitter process asking for a data object to process, which is then read {23}. Notice that this interaction is typical client behaviour as required by the design of the complete process network. If the object is a `Sentinel` {24} then the Emitter has no more data to be processed and thus the Worker can terminate {25} after writing the `Sentinel` to the `toWorker` channel {26}. If the input object holds data, it can be placed in the `sharedData` buffer subscripted by `index` {29}. The process then writes the `index` to the `toWorker` channel {30}, which may be delayed if the `DoWork` process has not yet completed processing the previous data object. Once the channel output has completed the `index` can be incremented modulo 3 to construct a circular buffer. This process is the only one that can modify the value of `index`; the others just input `index` and use it.

If the Worker node is executing on a single core node then the input, work and output stages of the process cannot be overlapped and thus the processes will be interleaved. If however the Worker node is executing on a multicore node then each of the internal processes could be executed at the same time. Hence the input, work and output processes will be executed in parallel and thus the input and output operations will be overlapped with the work process. It is thus crucial that access to the `sharedData` buffers is managed correctly.

The Listing 22-9 shows the structure of the `DoWork` process. The process inputs either an `Integer` or `Sentinel` object {21}. If it is a `Sentinel` object the process terminates having written the `Sentinel` object to the `SendOutput` process using the `workCompleted` channel {24}. If the input is an `Integer` object {27} then it is the index of a buffer in `sharedData` which the process can manipulate {28}. The index of the `sharedData` buffer is then written to the `SendOutput` process using the `workCompleted` channel {29}.

```
10 class DoWork implements CSProcess {
11
12     def workOn
13     def workCompleted
14     def workerId
15     def sharedData
16
17     void run(){
18         def index = -1
19         def running = true
20         while (running) {
21             def o = workOn.read()
22             if ( o instanceof Sentinel) {
23                 running = false
24                 workCompleted.write(o)
25             }
26             else {
27                 index = (Integer)o.intValue()
28                 sharedData[index].manipulate(workerId)
29                 workCompleted.write(index)
30             }
31         }
32     }
33 }
```

Listing 22-9 The DoWork Process Definition

The `SendOutput` process structure is shown in Listing 22-10. The structure is similar to the `DoWork` process structure in that it processes either `Sentinel` or `Integer` objects. In the case of a `Sentinel` it ensures that it is written to the `Collector` process using the `toCollector` channel {23} before the process terminates.

In the case of an `Integer`, we know that a buffer in the `sharedData` has to be written to the `Collector` process as shown in {27} using the `index` value that has been input {26}.

```
10 class SendOutput implements CSPProcess {
11
12     def workerFinished
13     def toCollector
14     def sharedData
15
16     void run(){
17         def index = -1
18         def running = true
19         while (running) {
20             def o = workerFinished.read()
21             if ( o instanceof Sentinel){
22                 running = false
23                 toCollector.write(o)
24             }
25             else {
26                 index = (Integer) o.intValue()
27                 toCollector.write(sharedData[index])
28             }
29         }
30     }
31 }
```

Listing 22-10 The SendOutput Process Definition

22.5 Summary

In this chapter we have constructed a generic architecture that can create a process farm for any data object that needs to be processed in a data parallel mode. The application uses the polymorphic capabilities of Java. It also uses the JCSP net2 architecture to enable the dynamic loading of class files over the network between nodes in a process network.

In addition we have alluded to the fact that if a process node contains multiple processes and the node it runs on has multi-core capabilities then, provided the underlying operating system can dynamically allocate processes to cores, the application will automatically utilise the many cores.

The next chapter will develop these ideas further by creating a mechanism that loads a network of workstations with processes and creates the required application specific communication structure dynamically.

23 Programming High Performance Clusters

An architecture is developed that permits the loading of processes onto a High Performance Cluster regardless of the application and the specific communication channels required by

- defining the required input and output channels in specific data structures
- defining a Worker Object interface that contains the required process definition and channel data needed to create a process in the cluster
- using a single process on each worker node that is independent of the application
- requiring only the host node to have knowledge of the application
- exploiting the ability to transfer class definitions over the network



Losing track of your leads?

Bookboon leads the way
Get help to increase the lead generation on your own website. Ask the experts.

Interested in how we can help you?
email ban@bookboon.com 



In this chapter we bring together a number of ideas previously described to show how we can build systems using High Performance Clusters (HPC) systems. For this explanation we assume that a HPC comprises a collection of processing nodes connected by a high performance interconnect. Typically, the interconnect is provided by a high bandwidth Ethernet infrastructure with a dedicated gigabyte switch. The network is normally not connected to the wider internet using a private network. There is usually one node, called the host, which also has a connection to a wider network enabling users to access the cluster over the internet. Each of the nodes is then usually implemented using a high performance multi-core processing node.

Such clusters are often provided with an operating environment, such as Condor, that allows users to run batch jobs that farm work out to as many nodes as the user requests using a data parallel architecture. Such operating environments tend to provide very limited capability for designing the type of parallel systems the earlier part of this book has been concerned with.

To make more use of such clusters, with more flexible parallel architectures, we require a mechanism to distribute work from the host to all the other, worker, nodes. Ideally, the general cluster nodes should run a single process that is able to load the specific work process from the host node. The major challenge is to dynamically construct the channel connections between each of the processes.

The effective use of multi-core nodes is achieved very simply by having the process that is run on a node itself containing a parallel structure that has as many processes as there are cores.

23.1 Architectural Overview

We shall assume that the host process is running its process before the other nodes are started. It is also assumed the host node has access to all the process and class definitions required by the application. The host node needs to know the number of worker nodes that are to be used.

A worker node then can run a process that has the following functionality.

The worker node registers its IP-address with the host node.

The host then constructs a worker object for each worker node that contains the process the node is to execute together with data structures that characterise the net channel connections the process requires. These connections are separated into input and output connections because we have to build all the input channel ends before any of the output ends can be created.

The host then sends each worker its corresponding worker object.

The workers then read the worker objects and start to process the data structure. The first action is to create any net input and output channel ends. Once this is complete the worker node sends a signal back to the host.

The host node waits for all the workers to signal that they have created their input channel ends. It then creates the processes that run on the host node and creates their input channel ends. Once this is complete the host sends a signal to each of the worker nodes enabling them to construct the net output channel ends. The host can now start its processes, which typically, emit work to the worker nodes and then collect the results once the worker nodes have terminated.

Once the worker nodes have created the net output channel ends they can start the worker process they have been allocated. If the process contains a parallel then these processes will be allocated to the available cores by the Java Virtual machine and underlying operating system.



"I studied English for 16 years but...
...I finally learned to speak it in just six lessons"
Jane, Chinese architect

ENGLISH OUT THERE

Click to hear me talking before and after my unique course download



The allocation of channels is separated so that the channels used to register nodes and to load process are separated from and have a specific structure that is not related in any way to the application channels. This also means that the process loading channels can be used to transfer class definitions using the code loading channel capability described in the previous chapter. The advantage of this separation is that we can build the host and worker node scripts in a uniform manner that is independent of the application architecture. This means that users only have to become familiar with one architecture, to be able to use a HPC system using any style of parallel application.

23.2 The Host and Node Scripts

These scripts have to be run on the HPC host and worker nodes respectively. In a later section we shall show how we can create a means of launching these scripts from a runnable jar using operating system batch files. Each worker node has to run the Node script. Figure 23-1 shows the net channel architecture to be created amongst a set of host and worker nodes. Each node has an `any2net` output connection to the host. Each node has a `one2net` input channel from the host. Each input location uses the channel number 1 and we assume that a specific port is used for all connections. In this description each node and the host will use port 1000. Each node and the host have their own IP-address. The application channels will use any channel number other than 1.

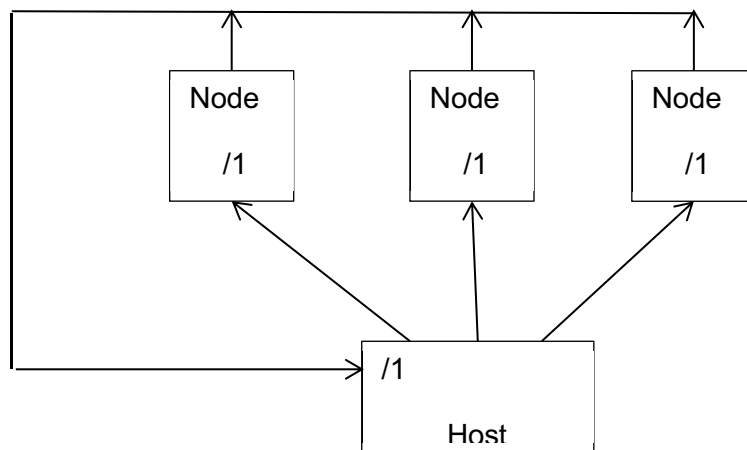


Figure 23-1 Net Channel Infrastructure to Load Worker Processes in a HPC

The application channel structure is shown in Figure 23-2 and as can be observed is completely different. The application uses a simple data parallel structure in that the work can be shared among the nodes in an equal manner. The host node runs two internal processes called Emitter and Collector. The Emitter process sends the work to be undertaken to each of the nodes. Once a worker Node has completed the task the results are returned to the Collector process. The input channel to each node is given the channel number 100 and the input channels to the Collector process are given the channel numbers 100, 101 and 102 respectively.

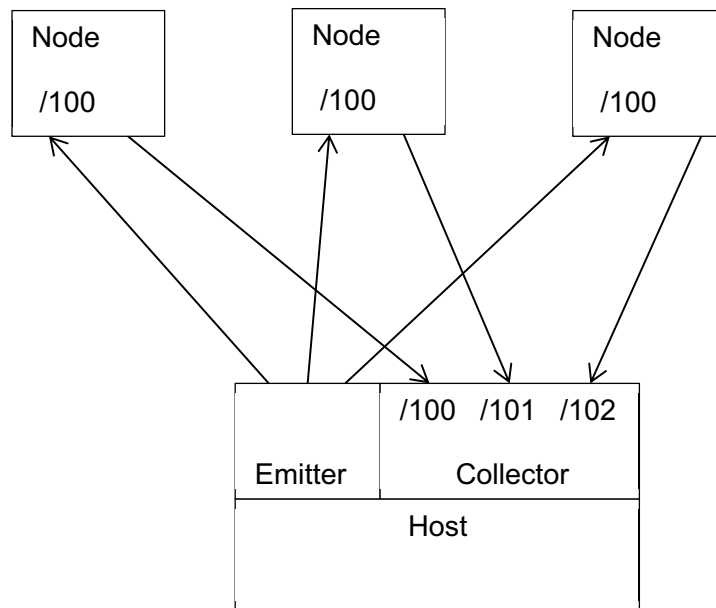


Figure 23-2 The Application Channel Structure

Listing 23-1 shows the first part of the host script. The script will be run using command line arguments. Thus `workers {10}` and `cores {11}` will be passed the number of worker nodes and the number of cores to be used at each node. These arguments are independent of the application. The argument `iterations {12}` is application specific and will be discussed later.

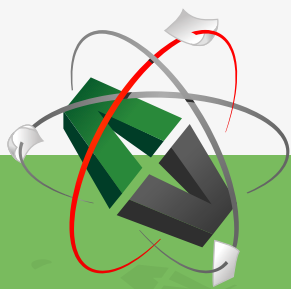
A `timer` is used {15–16} so that analysis of the time taken to complete the application can be reviewed in terms of the number of workers and cores used. This means that algorithm speed-up can be determined in terms of these values.

The host IP-address is determined as `hostAddr`, which is listening on port 1000 {17} and which can then be used to create a node instance {18}. The IP-address of the host is then obtained and printed {19} together with the number of `workers` and `cores` {20}. The `hostRequest` channel is then created {21} as channel number 1. This is the channel used by each of the nodes to communicate information about the worker node.

The `ChannelOutputList loadChannels` is used to hold the output channels used to send worker processes to the nodes {23}. The list `nodes` {24} is used to hold the IP-address of each of the worker nodes. The loop {25–31} is used to iterate over each of the worker nodes to obtain a `RequestWorker` object from each node {26}. One field of the `RequestWorker` object is the `loadLocation`, which holds the net channel location of the input channel to the Node. This is then used to create the output channel end `nodeLoadChannel` {27–18} as a code loading channel and will be used to send the worker process object to the node. The channel is then appended to the `loadChannels` list {30}. The `timer` is then read again to provide a measure of the time taken to read the initial communication from all of the nodes {33}.

```
10 def workers = new Integer(args[0]).intValue()
11 def cores = new Integer(args[1]).intValue()
12
13 def iterations = new Integer(args[2]).intValue()
14
15 def timer = new CTimer()
16 def startTime = timer.read()
17 def hostAddr = new TCPIPNodeAddress(1000)
18 Node.getInstance().init(hostAddr)
19 def hostIP = hostAddr. getIpAddress()
20 println "Host running on $hostIP for $workers worker nodes with $cores cores"
21 def hostRequest = NetChannel.numberedNet2One(1)
22
23 def loadChannels = new ChannelOutputList()
24 def nodes = []
25 for ( w in 1 .. workers ) {
26   def workerRequest = (RequestWorker)hostRequest.read()
27   def nodeLoadChannel = NetChannel.one2net( workerRequest.loadLocation,
28                                             new CodeLoadingChannelFilter.FilterTX())
29   loadChannels.append(nodeLoadChannel)
30   nodes << workerRequest.nodeIP
31 }
32 println "Processed $workers worker requests"
33 def requestReadTime = timer.read()
```

This e-book
is made with
SetaPDF



PDF components for PHP developers

www.setasign.com



```
34 def workerObjects = []
35 for ( w in 0..< workers) {
36   workerObjects << new WorkerObject ( workerProcess: new
      McPiWorker(cores: cores),
37                                     inConnections : [100],
38                                     outConnections: [[hostIP, 100 + w]])
39 }
40 for ( w in 0 ..< workers ) {
41   loadChannels[w].write(workerObjects[w])
42 }
43 def workersSentTime = timer.read()
44 println "Sent worker objects to workers"
45 def emitterInConnections = []
46 def emitterOutConnections = []
47 for ( w in 0..< workers) {
48   emitterOutConnections << [nodes[w], 100]
49 }
50 def collectorInConnections = []
51 for ( w in 0..< workers) {
52   collectorInConnections << (100 + w)
53 }
54 def collectorOutConnections = []
55 def emmitterInChannelList = new ChannelInputList()
56 def emitterOutChannelList = new ChannelOutputList()
57 def collectorInChannelList = new ChannelInputList()
58 def collectorOutChannelList = new ChannelOutputList()
59
60 for ( w in 0 ..< workers ) {
61   hostRequest.read()
62 }
```

Listing 23-1 The First part of the Host Script

The list `workerObjects` {34} is used to hold the objects that will be sent to each of the worker nodes. The loop {35–39} is used to iterate over the number of `workers` to create the objects that will be sent to each of the worker nodes.

A `WorkerObject` {36–38} comprises three fields. The first is the worker process itself {36}. The next is a list of input channel connections, `inConnections`, {37}, which comprises the input channel numbers used by this worker node. Finally, `outConnections` {38} is a list of two element lists that comprise the [IP-address, Channel number] of the node to which the worker node outputs. Both these channel connections are relative to the worker node itself. The loop {40–42} then outputs each of the created worker objects to each of the worker nodes. After which the `timer` is read again to provide an indication of the time taken to send all the worker objects from the host to the worker nodes.

The next phase of the script {45–58} is to create the channels that will be used by the `Emitter` and `Collector` processes that will run in the host node. These are specific to the application. By inspection it can be seen that the application structure shown in Figure 23-2 is created.

The loop {60–62} reads a `Signal` from each of the worker nodes. This `Signal` is used to indicate that each worker node has created its input channels. The worker nodes have to create all the input channels before any of the output channels can be created. The `Emitter` and `Collector` process input channels can now be created as shown in Listing 23-2 {63–68}. Once this is complete, a `Signal` can be sent to each of the worker nodes indicating that the output channels can now be created {69–71} because it is known that all input channel ends now exist. The output connections for the `Emitter` and `Collector` process can be created {73–80}. Each connection comprises a list of two elements, where the first element is the IP-address to which the output connection is to be made using port 1000. The second element is the channel number to be used. Each of the output channel ends are appended to a channel output list.

```
63 emitterInConnections.each{ cn ->
64   emitterInChannelList.append(NetChannel.numberedNet2One(cn))
65 }
66 collectorInConnections.each{ cn ->
67   collectorInChannelList.append(NetChannel.numberedNet2One(cn))
68 }
69 for ( w in 0 ..< workers ) {
70   loadChannels[w].write(new Signal())
71 }
72
73 emitterOutConnections.each{ connection ->
74   def outNodeAddr = new TCPIPNodeAddress(connection[0], 1000)
75   emitterOutChannelList.append(NetChannel.any2net(outNodeAddr,
76     connection[1]))
77 }
78 collectorOutConnections.each{ connection ->
79   def outNodeAddr = new TCPIPNodeAddress(connection[0], 1000)
80   collectorOutChannelList.append(NetChannel.any2net(outNodeAddr,
81     connection[1]))
82 }
83
84 def emitter = new McPiEmitter(workers: workers, iterations: iterations)
85 emitter.connect(emitterInChannelList, emitterOutChannelList)
86
87 def collector = new McPiCollector(workers: workers, iterations:
88   iterations, cores: cores)
89 collector.connect(collectorInChannelList, collectorOutChannelList)
90
91 def emitterPM = new ProcessManager( emitter )
92 def collectorPM = new ProcessManager( collector )
93 def hostStartTime = timer.read()
94 emitterPM.start()
95 collectorPM.start()
```

```
92
93 emitterPM.join()
94 collectorPM.join()
95 def hostEndTime = timer.read()
96 println "Host terminated"
97 def workerTimes = []
98 def hostStartup = requestReadTime - startTime
99 def hostLoad = workersSentTime - requestReadTime
100 def hostInitiate = hostStartTime - workersSentTime
101 def hostElapsed = hostEndTime - hostStartTime
102 workerTimes << ["Host", hostStartup, hostLoad, hostInitiate, hostElapsed]
103 for ( w in 0 ..< workers){
104   def workerRawTimes = hostRequest.read()
105   def startup = workerRawTimes[1] - workerRawTimes[0]
106   def load = workerRawTimes[2] - workerRawTimes[1]
107   def initiate = workerRawTimes[3] - workerRawTimes[2]
108   def elapsed = workerRawTimes[4] - workerRawTimes[3]
109   workerTimes << ["Wk: " + w, startup, load, initiate, elapsed]
110 }
111 println "Node\tstart\tload\tinit\telapsed"
112 workerTimes.each { timings ->
113   timings.each{ print "$it\t"}
114   println""
115 }
```

Listing 23-2 The Second part of the Host Script

The next part of the script initiates the Emitter and Collector processes that will be executed in the host node. The processes are called `McPiEmitter` and `McPiCollector` respectively {82, 84}. The actual processes used in these processes are described in 23.3 and implement a parallel solution to the calculation of pi using Monte Carlo methods. A worker process implements the `WorkerInterface` which requires the creation of a method called `connect`. The `connect` method has two channel list parameters; the first for input channel connections and the second for output channel connections as can be observed {83, 85}. The emitter and collector instances can now be passed to a `ProcessManager` instance and started {87–91}.

The script then waits for these processes to terminate {93–94}, after which the `timer` can be read to provide an indication of the time taken for the host to end processing. The remaining part of the script {97–115} concerns the input of times from each of the worker nodes so that analysis of the time taken in each phase of the script can be undertaken.

The Node script is shown in Listing 23-3. This script is universal in that it is able to load any worker process, regardless of the number of cores that will be used and also of the application communication structure. The number of cores is encoded in the worker process that will be loaded. The network channel connections are encoded in the `WorkerObject`'s `inConnections` and `outConnections` properties.

```
10 def timer = new CTimer()
11 def startTime = timer.read()
12 def nodeAddr = new TCPIPNodeAddress(1000)
13 Node.getInstance().init(nodeAddr)
14 def workerIP = nodeAddr.getIpAddress()
15 println "Worker is located at $workerIP"
16
17 def loadChannel = NetChannel.numberedNet2One(1, new
    CodeLoadingChannelFilter.FilterRX())
18 def loadChannelLocation = loadChannel.getLocation()
19 def hostIP = args[0]
20 def hostAddr = new TCPIPNodeAddress(hostIP, 1000)
21 def hostRequest = NetChannel.any2net(hostAddr, 1)
22 def requestWorker = new RequestWorker ( loadLocation: loadChannelLocation,
23                                         nodeIP: workerIP)
24 hostRequest.write(requestWorker)
25 def requestSentTime = timer.read()
26
27 def workerObject = (WorkerObject)loadChannel.read()
28 def workerReadTime = timer.read()
29
30 def wProcess = (WorkerInterface) workerObject.workerProcess
31 def inConnections = workerObject.inConnections
32 def outConnections = workerObject.outConnections
```

gaiteye
Challenge the way we run

**EXPERIENCE THE POWER OF
FULL ENGAGEMENT...**

**RUN FASTER.
RUN LONGER..
RUN EASIER...**

**READ MORE & PRE-ORDER TODAY
WWW.GAITEYE.COM**

The advertisement features a background image of a person running on a path during a sunrise or sunset. The Gaiteye logo is in the top left. The main text is in the center-left. A yellow button with a hand cursor icon is in the bottom right of the ad area.

```
33
34 def inChannels = new ChannelInputList()
35 inConnections.each{ cn ->
36   inChannels.append(NetChannel.numberedNet2One(cn))
37 }
38
39 hostRequest.write(new Signal())
40 loadChannel.read()
41
42 def outChannels = new ChannelOutputList()
43 outConnections.each{ connection ->
44   def outNodeAddr = new TCPIPNodeAddress(connection[0], 1000)
45   outChannels.append(NetChannel.any2net(outNodeAddr, connection[1]))
46 }
47 wProcess.connect(inChannels, outChannels)
48 def wPM = new ProcessManager(wProcess)
49 def workerStartTime = timer.read()
50 wPM.start()
51 wPM.join()
52 def workerEndTime = timer.read()
53 println "worker has terminated"
54 hostRequest.write([ startTime, requestSentTime, workerReadTime,
55                   workerStartTime, workerEndTime])
```

Listing 23-3 The Node Script

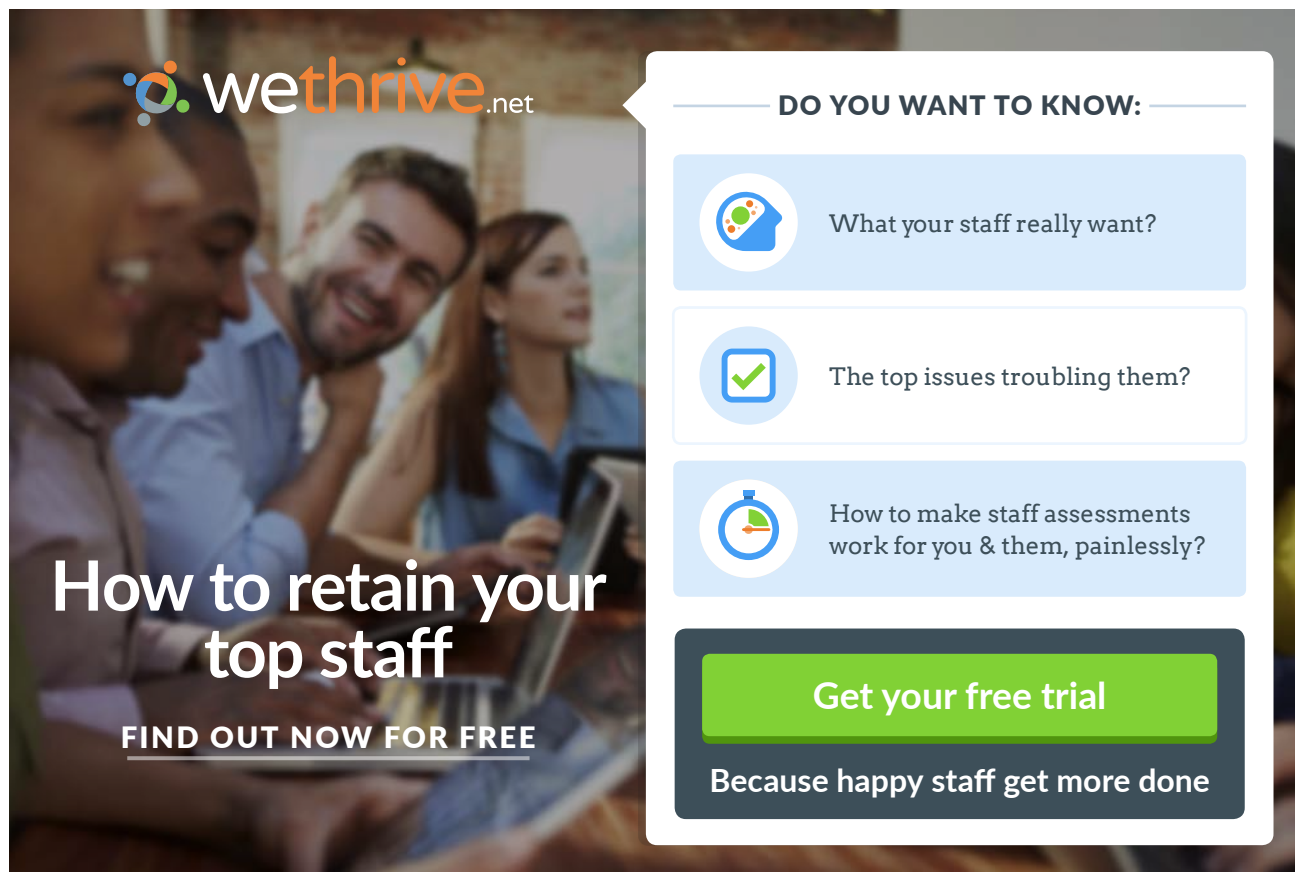
A `timer` is started {10} and the `startTime` read {11}. A node address is obtained at the node's IP-address listening on port 1000 {12} after which a node instance can be created {13}. The IP-address of the node is then obtained as `workerIP` {14} as this will be sent to the host node so that the host can create the required application net channel connections to the node. A `net2one` channel is created at the node using channel number 1 {17}. This channel is also a code loading receiving channel. The location of this channel is then created as `loadChannelLocation` {18}. The IP-address of the host node is passed to the script as `args[0]` {19}. The address of the host node can then be created {20} and the `any2net` channel connection can be created as shown in Figure 23-1 as the net channel `hostRequest` {21}.

An instance of the class `RequestWorker` is created that has two properties. The property `loadLocation` {22} holds the location of the net input channel the host uses to send the `WorkerObject` to the node created as `loadChannelLocation`. The second property is the `nodeIP` {23} holding the IP-address of the worker node created as `workerIP`. The new `RequestWorker` object is then written to the host {24}, thereby registering this node with the host node. This means that the host node does not have to know in advance the IP addresses of the worker nodes it is to use. Obviously a worker node may have to be allocated to a specific node if that node has access to specific hardware resources required by the application. The time the request was sent is recorded {25}.

The node then reads a `WorkerObject` from the host {27} and the time it was read is also recorded {26}. The properties of the `WorkerObject` are then extracted as `wProcess`, which holds the `workerProcess` itself {30}, `inConnections`, which holds the net input channel connections this node is to provide {31} and `outConnections`, which holds the output net channel addresses this node is to communicate with {32}.

Each node has to create its net input channel connections before it can create any of its output connections. The channel input list `inChannels` is used to hold each of the net channel inputs {34}. The variable `inConnections` is a list of channel numbers that are to be used as inputs to this node {31}. The script iterates through each of the elements, if any, appending a `net2one` channel to `inChannels` using the element value as the channel number {35–37}.

The node now writes a `Signal` object to the host {39}, using the `hostRequest` net channel, to indicate that all the input channels have been created. The node then reads another `Signal` object from the host node, using the `loadChannel` {40}. This `Signal` is only sent by the host when it has received `Signals` from all the nodes indicating they have all created their net input channels (Listing 23-2 {69–70}). In a similar manner the node can now create its net output channels. It iterates through each element of the `outConnections` list {43–46} appending `any2net` channels to the output channel list `outChannels` {42}. The use of `any2net` channels means that the node script can be used for `one2one` and `any2one` connections.



wethrive.net

How to retain your top staff

FIND OUT NOW FOR FREE

DO YOU WANT TO KNOW:

- What your staff really want?
- The top issues troubling them?
- How to make staff assessments work for you & them, painlessly?

Get your free trial

Because happy staff get more done

The object `wProcess` implements the interface `WorkerInterface`, which has a single method called `connect`. The parameters of the `connect` method are a `ChannelInputList` and a `ChannelOutputList`. These channel lists have been previously created as `inChannels` and `outChannels` {47}. An instance of `ProcessManager`, `wPM`, can now be created with `wProcess` as its parameter {48}. The `timer` is again read to determine the time it took to initiate the process and its channels {49}. The instance `wPM` can now be started {50} after which the script waits for it to terminate {51}. The `timer` is read once again to determine the processing time {52}. The script finishes by writing all the recorded times to the host node so the time taken for each stage in each node can be determined {54–55}.

23.3 An Application – Montecarlo Pi

The calculation of π using statistical methods is a simple and easily understood way of calculating an approximate value of the constant. Given a unit square we can determine for each random value of $[x, y]$ s.t. $0 \leq x < 1.0, 0 \leq y < 1.0$ whether the point lies inside the quadrant that has a unit radius using Pythagoras’ theorem. The ratio of the points within the quadrant to all the generated points is $\pi/4$. Thus the algorithm is one of simply generating many random points within the unit square and determining how many points lie within the unit quadrant; the more random points the greater the accuracy of the approximation.

This is a very simple algorithm to parallelise. Each worker node is allocated an equal share of the total number of generated points. Each worker then subdivides its allocation equally among the available cores on that node.

The structure of the worker processes at each node is shown in Figure 23-3. The number of `McPiCore` processes can be varied dynamically depending on the script argument.

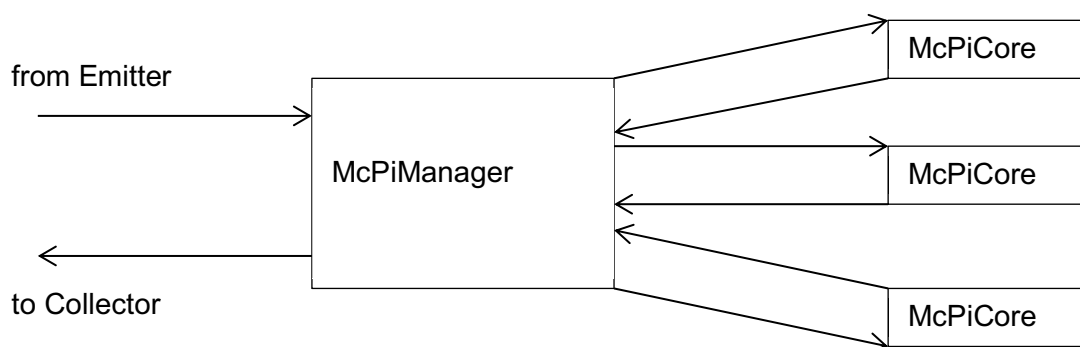


Figure 23-3 The Internal Structure of A Worker Node, called `McPiWorker`

23.3.1 The McPiCore Process

Listing 23-4 shows the `McPiCore` process, which is replicated for as many cores as it is intended to use (see `McPiWorker` 23.3.3). As shown in Figure 23-3 the process has one input channel and one output channel {12, 13}

```
10 class McPiCore implements CProcess {
11
12     def ChannelInput inChannel
13     def ChannelOutput outChannel
14
15     void run() {
16         def iterations = inChannel.read()
17         def rng = new Random()
18         def int inQuadrant = 0
19         1.upto(iterations) {
20             def randomX = rng.nextFloat()
21             def randomY = rng.nextFloat()
22             if ( ((randomX * randomX)+(randomY * randomY)) < 1.0) inQuadrant
                = inQuadrant + 1
23         }
24         outChannel.write(inQuadrant)
25     }
26 }
```

Listing 23-4 The `McPiCore` Process

The number of `iterations` to be determined is read from `inChannel` {16}. A `Random` number generator is defined as `rng` {17}. The variable `inQuadrant` is initialised and will be used to count the number of randomly generated points that are within the quadrant {18}. A loop is then formed that iterates over the number of `iterations` {19–23}. Two random values are generated {20, 21} and used to determine whether the point thereby represented is within the unit quadrant; if so `inQuadrant` is incremented. Once all the points have been evaluated the value of `inQuadrant` is written to the `outChannel` {24} and the process terminates. This process is compute bound as it only undertakes two communications.

23.3.2 The McPiManager Process

Listing 23-5 shows the McPiManager process, which inputs the number of iterations to be undertaken by the worker as a whole and then subdivides the work amongst the cores.

```
10 class McPiManager implements CProcess {
11
12     def ChannelInput inChannel
13     def ChannelOutput outChannel
14     def ChannelOutputList toCores
15     def ChannelInputList fromCores
16
17     void run() {
18         def cores = fromCores.size()
19         def iterations = inChannel.read()
20         for ( c in 0..< cores) toCores[c].write(iterations / cores)
21         def quadSum = 0
22         for ( c in 0..< cores) quadSum = quadSum + fromCores[c].read()
23         outChannel.write(quadSum)
24     }
25 }
```

Listing 23-5 The McPiManager Process



The advertisement features a black header with the CMO Inspired Conference logo on the left, which consists of a green speech bubble containing the letters 'CMO'. To the right of the logo, the text 'INSPIRED CONFERENCE' is written in large, white, bold, sans-serif capital letters. Below this, in smaller white capital letters, is the date and location: '25 OCTOBER | DE VERE BEAUMONT ESTATE | OLD WINDSOR UK'. The main body of the ad is a collage of images: the top half shows a large, white, classical-style building with many windows, surrounded by green trees and a fountain in the foreground; the bottom half shows a collage of people at a conference, including a woman speaking into a microphone, a man presenting at a screen, and a large audience seated in a hall. At the bottom of the ad, a black banner contains the text 'Join Over 100 Chief Marketing Officers & Digital Innovators' in green.



The channels `inChannel` and `outChannel` will eventually be connected to net channels of the `McPiEmitter` and `McPiCollector` processes respectively {12, 13}. The channel lists `toCores` and `fromCores` {14, 15} will be created in the `McPiWorker` process described in the next section. The number of cores can be determined from the size of either of the channel lists {18}. The number of iterations to be undertaken is read from `inChannel` {19}. The number of iterations per core is written to each core using the `toCores ChannelOutputList` {20}. The variable `quadSum` {21} is used to tally the total number of points in each quadrant and is incremented by reading a value from each of the cores using the `ChannelInputList fromCores` {22}. Finally, the process writes the value held in `quadSum` to `outChannel` and thus to the `McPiCollector` process and then terminates {23}. This process is essentially communication bound as the effective work undertaken comprises communication only. Hence once the `McPiCore` processes have started it will not incur any processing overhead.

23.3.3 The `McPiWorker` Process

Listing 23-6 shows the `McPiWorker` process, which is the process that is loaded into each node as shown in Figure 23-2. The role of this process is to create the processes required at each node, to read the number of iterations the node is to compute from the `McPiEmitter` process and then to return the number of points lying within the unit quadrant to the `McPiCollector` process. The process thus only undertakes communication and does not therefore incur any processing overhead. This means that the available processing resource of the node as indicated by the number of cores can be fully utilised by the `McPiCore` processes without any interruption by the `McPiManager` and `McPiWorker` processes until they terminate.

The `McPiWorker` process implements the `WorkerInterface` interface that has one method, `connect`. This method {16-19} expects two channel lists which are then associated with the properties `inChannels` and `outChannels` {12, 13}. The `connect` method is called in the script that runs on each node (see Listing 23-3 {43}). The property `cores` {14} can be modified when an instance of the object is created in the script that runs on the host node (see Listing 23-1 {35–39}).

```
10 class McPiWorker implements WorkerInterface {
11
12     def ChannelInputList inChannels
13     def ChannelOutputList outChannels
14     def cores = 1
15
16     def connect(inChannels, outChannels){
17         this.inChannels = inChannels
18         this.outChannels = outChannels
19     }
20
21     void run(){
22         println "running McPiWorker"
```

```
23     def M2C = Channel.one2oneArray(cores)
24     def C2M = Channel.one2oneArray(cores)
25     def toCores = new ChannelOutputList(M2C)
26     def fromCores = new ChannelInputList(C2M)
27     def coreNetwork = (0 ..< cores).collect { c ->
28         return new McPiCore( inChannel: M2C[c].in(),
29                             outChannel: C2M[c].out() )
30     }
31     def manager = new McPiManager ( inChannel: inChannels[0],
32                                   outChannel: outChannels[0],
33                                   toCores: toCores,
34                                   fromCores: fromCores)
35     new PAR(coreNetwork + [manager]).run()
36 }
37 }
```

Listing 23-6 The McPiWorker Process

The process creates the channel arrays used to connect the `McPiManager` process to the `McPiCore` process {23, 24}, from which the required channel lists can be created {25, 26} see Figure 23-3. The network of `McPiCore` processes uses the Groovy method `collect` that iterates over a range and returns a list of objects. In this case the objects are instances of the `McPiCore` process with connections to the correct channels of the previously created channel arrays, `M2C` and `C2M` {27–30}. An instance of the `McPiManager` process is then created as `manager`. In this case only one channel is used in each of the channel lists {31–34}; in more complex node communication structures then more will be used and it is the programmer's responsibility to check that the correct channels are connected. In the accompanying Chapter Examples there is a network that uses a more complex communication structure amongst the worker nodes.

Finally, a parallel is constructed from the `coreNetwork` and the `manager` process. This terminates when all the processes created internally terminate. Thus, provided each internal process terminates, then the whole process network will terminate and thus the node will return to an idle state.

23.3.4 The McPiEmitter Process

Listing 23-7 shows the `McPiEmitter` process that runs in the host node, see Figure 23-2. This process also implements `WorkerInterface` and thus has a `connect` method. The `connect` method is called from the host script (see Listing 23-2 {83}).

The properties `workers` and `iterations` are initialised when the process is constructed (see Listing 23-2 {82}) in the host node script.

The effect of the process is to write {24} to each of the worker nodes, running an instance of `McPiWorker`, the number of iterations to be undertaken by each worker. The process then terminates.

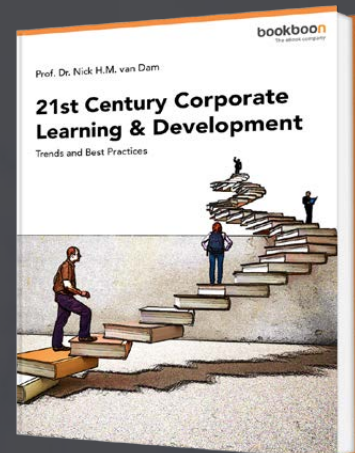
```
10 class McPiEmitter implements WorkerInterface {
11
12     def ChannelInputList inChannels
13     def ChannelOutputList outChannels
14     def workers = 1
15     def iterations = 192
16
17     def connect(inChannels, outChannels){
18         this.inChannels = inChannels
19         this.outChannels = outChannels
20     }
21
22     void run(){
23         println "running McPiEmitter"
24         for ( w in 0 ..< workers) outChannels[w].write(iterations / workers)
25     }
26 }
```

Listing 23-7 The McPiEmitter Process

Free eBook on Learning & Development

By the Chief Learning Officer of McKinsey

Download Now



Click on the ad to read more

23.3.5 The McPiCollector Process

Listing 23-8 shows the McPiCollector process that runs in the host node. This also implements the WorkerInterface. The process is constructed in the host node script (see Listing 23-2 {84–85}). This initialises the values of the properties workers, iterations and cores {14–16} and then calls the connect method {18–21}.

```
10 class McPiCollector implements WorkerInterface {
11
12     def ChannelInputList inChannels
13     def ChannelOutputList outChannels
14     def workers = 1
15     def iterations = 192
16     def cores = 1
17
18     def connect(inChannels, outChannels){
19         this.inChannels = inChannels
20         this.outChannels = outChannels
21     }
22
23     void run(){
24         println "running McPiCollector"
25         def quadSum = 0
26         for (w in 0 ..< workers) quadSum = quadSum + inChannels[w].read()
27         def pi = quadSum / iterations * 4
28         println "The value of pi is $pi"
29         println "Workers: $workers, Iterations: $iterations, Cores :
30             $cores"
31     }
```

Listing 23-8 The McPiCollector Process

The process initialises a variable quadSum {25} which is then incremented by reading a value from each of the worker nodes {26}. The value of π (pi) is then evaluated and printed {27–29}.

23.3.6 Analysis

The system was run on a network of workstations connected by a gigabit Ethernet. The Ethernet is part of a large network and thus there would be other traffic on the network and not just communications relating to this application. The Montecarlo π system does not impose a high communication overhead as it is mostly compute bound once it is running, however, initially when the processes are being distributed from the host to the worker nodes there is a larger communication, but this only occurs once. The class file for a McPiWorker process is about 9KB and those of the McPiCore and McPiManager are 7Kb and 8KB respectively. Thus the total communication overhead is still very small.

Each workstation was an Intel Core™ i5-2500 CPU running at 3.30GHz with 4GB RAM. One node was designated as the host and up to four other nodes were used. Each processor had 4 cores. The system was executed with 1 to 4 worker nodes utilising 1 to 4 cores. Thus 16 timings were collected and the process was repeated 5 times and the average values were obtained over all the runs for the subsequent analysis.

Typical output from the operation of the system is shown below, all times are in milliseconds. The output is from an execution of the system using 4 workers each using 4 cores. The number of iterations was 192 million. Each core therefore calculated one-sixteenth or 12 million iterations.

```
The value of pi is 3.1417472500
Workers: 4, Iterations: 192000000, Cores : 4
Host terminated
Node   start   load   init  elapsed
Host   9235    187    671   4306
Wk: 0  358     6053   406   4446
Wk: 1  359     4430   390   4509
Wk: 2  327     1593   358   4579
Wk: 3  432     190    335   4613
```

The times are obtained from the scripts that run on the host and other nodes. No timing information is obtained from the application processes. The **start** and **load** times are influenced by the time it takes to get round each of the workstations to hit the enter key to start the batch file that runs the host and worker node scripts. The **init** time is the time it takes to set up each of the worker nodes once they have received the worker process. The **elapsed** time is that taken to undertake the computation and return the results. In the following analysis the minimum values for start and load was taken to remove operator influence. The average value for init and elapsed was taken.

Figure 23-4 shows a graph of the Elapsed times for each of the combinations of workers and cores.

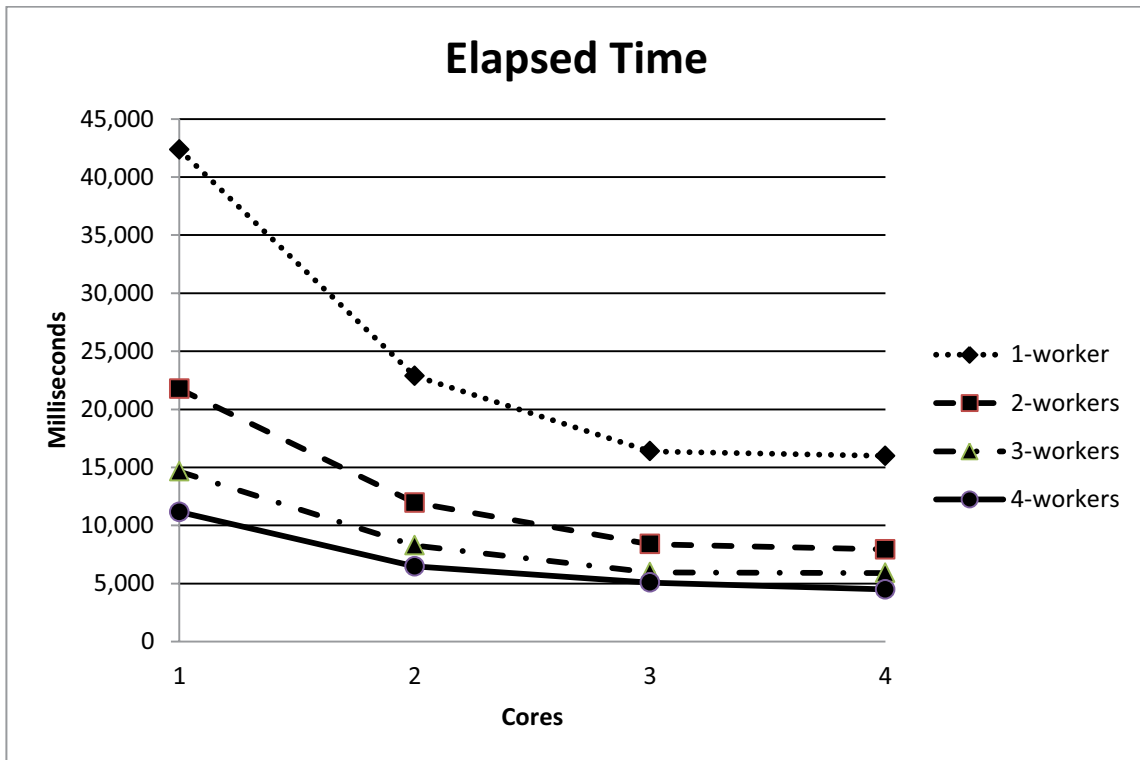


Figure 23-4 Graph Showing the Performance of the Montecarlo Pi System



Discover the truth at www.deloitte.ca/careers

Deloitte.

© Deloitte & Touche LLP and affiliated entities.



The shape of the curves is similar and the initial speedup when the number of workers is increased but the number of cores is kept at one is reasonable. It is interesting to note that, for example, using two workers with one core is faster than using one worker with two cores. This behaviour is consistent over all combinations. The obvious area of interest is the non-linearity of the curves with respect to the increase in worker and cores. This is shown in the following table.

workers	cores	Total Cores	Speedup	Ideal / Actual	Startup / Run
1	1	1	1.00	100%	1%
1	2	2	1.85	93%	2%
1	3	3	2.58	86%	2%
1	4	4	2.65	66%	5%
2	1	2	1.95	97%	3%
2	2	4	3.54	89%	3%
2	3	6	5.05	84%	6%
2	4	8	5.34	67%	5%
3	1	3	2.89	96%	4%
3	2	6	5.12	85%	5%
3	3	9	7.10	79%	7%
3	4	12	7.17	60%	10%
4	1	4	3.80	95%	5%
4	2	8	6.54	82%	7%
4	3	12	8.32	69%	9%
4	4	16	9.43	59%	11%

The Speedup column is based on the elapsed time only. Thus it is immediately obvious that it is better to increase the number of workers *before* increasing the use of cores. The column Ideal / Actual shows the relationship between the speedup versus the ideal speedup if the relationship had been linear. Thus using 4 workers each with 4 cores only achieves 59% of the ideal 16 it would be hoped for. The Startup / Run column shows the relationship between the sum of the start and load times against the init and elapsed times. This column indicates the percentage of time spent loading the system as opposed to running the system once the processes have been distributed around the network. As expected, as the number of workers and cores increase the amount of time spent in the Startup phase increases. Even in the case of 4 workers this is only 11% of the total time. The possibility of course with using 4 workers is to increase the number of iterations to obtain a more accurate result. This is typical of many parallelising activities that as the available resource increases, the opportunity to improve the size of the data set or accuracy is increased, nullifying the improvement in absolute performance.

23.3.7 Running the System

Rather than installing Eclipse at each node, a far more satisfactory mechanism would be to create a runnable `jar` file. A method for achieving this is described elsewhere (Anon., 2009). A method specific for this system is contained within the Eclipse package for Chapter Examples. A package called `c23.launcher` contains a simple Java class that contains a `main` method as shown in Listing 23-9. The `main` method is passed command line arguments as the array `args` {11}. The first argument contains the name of the Groovy script that is to be executed by the call to `GroovyShell.main()` {13}. The first argument is removed from the array of `args` and the remainder are passed to the Groovy script.

```
10 public class GroovyLauncher {
11     public static void main(String[] args) {
12         System.out.println("running script: " + args[0]);
13         GroovyShell.main(args);
14     }
15 }
```

Listing 23-9 The `GroovyLauncher` `main` class

The `GroovyLauncher` class is executed as a Java Application, within Eclipse to create a Launch Configuration. The application will fail with errors but that does not matter. Within Eclipse a runnable JAR file is exported for the project `ChapterExamples`. The `jar` file should be saved in the same folder as that containing the folder `ChapterExamples`, which contains the Groovy source of all the examples. Two batch files should now be created similar to those given in the package `c23.MontecarloPi`.

The content of the batch file to run the host node is:

```
java -jar ChapterExamples.jar .\ChapterExamples\src\c23\MontecarloPi\RunArgMcPiHost.groovy %1
%2 %3
```

where `%1` is the number of workers, `%2` is the number of cores and `%3` is the number of iterations.

The content of the batch file to run a worker node is:

```
java -jar ChapterExamples.jar .\ChapterExamples\src\c23\MontecarloPi\
RunArgMcPiNode.groovy %1
```

where `%1` is the IP-address of the host node.

The simplest way to use the system is to create a folder containing the `ChapterExamples` source folder, the `ChapterExamples.jar` file and the two batch files.

Running the appropriate batch file on the workstation will result in the required script being executed. The host node **MUST** be executed first and it prints out the IP-address of the host node that can then be passed as an argument to the nodes.

23.4 Summary

This chapter has explored ways in which applications can be loaded onto networks of workstations that use mutli-core processors. The infrastructure is capable of loading any application regardless of the communication links required. It has also been shown how easy it is to exploit multi-core processors, simply by creating a parallel structure on each of the nodes. The underlying Java Virtual Machine (JVM) and operating system support makes it very easy to exploit such multi-core processors. The ability to create the required parallel structure to exploit this ability is much more challenging and this chapter has shown that communicating process architectures make it especially easy. The fact that this parallel activity is achieved in a manner that is parameterised simply by the number of cores available demonstrates how easy it is to exploit parallelism, provided the correct environment and tools are used.

© 2013 Accenture. All rights reserved.

be > your degree

Bring your talent and passion to a global organization at the forefront of business, technology and innovation. Discover how great you can be.

Visit accenture.com/bookboon

Be greater than.
consulting | technology | outsourcing

accenture
High performance. Delivered.



24 Big Data – Solution Scaling

Using concordance as the basis we explore Big Data problems by:

- defining the concordance problem
- discussing how it can be subdivided into parallel tasks
- instrumenting the algorithms to determine which parts may be amenable to parallelisation
- showing how the algorithm can be implemented, sequentially, in a single core and also on a multi-core processor
- describing one way in which the algorithm could be distributed over a network of workstations, using the results of the instrumentation
- analysing the resulting implementations in terms of their processing times

A key capability of parallel systems is that they can be scaled to deal with data sets of different sizes. Typically, as the size of a data set increases more processing resource can be utilised to ensure processing time remains within reasonable bounds. The design of the process network has to be scalable in terms of at least one of the data set parameters. The key design constraint is to choose such parameters with care. The other key aspect of such systems is that it is always possible to read a data structure in parallel but writing to a data structure in parallel is not usually feasible unless the programmer takes care to partition the data structure so this becomes possible. A general solution to this problem using concurrent reads and exclusive writes was discussed in Chapter 13. A more specialised solution, using shared buffers, was described in Chapter 22.4.3.

24.1 Concordance – A Typical Problem

A concordance is a means of determining the places where the same string of words is repeated in a text. Usually the concordance is constructed for sequences of words for length 1 up to some defined value N . A concordance is constructed for large texts in which such repetitions are indicative of consistency or as a means of determining the likelihood of common authorship of a number of different texts. In this discussion we are not concerned with such matters but simply in the problem of constructing the concordance.

The production of the concordance is mainly limited by file reading and writing. It is difficult to parallelise the reading of the file containing the text unless it is provided in separate files. We shall assume the initial text is supplied as a single file. Writing the files that make up the concordance can be parallelised if the concordance corresponding to each value of N is written to a separate file. For example, the text of the Bible is 4.6 Mbytes but generates 26 Mbytes of output for $N = 6$. The files generated by each of the values of N vary from 13 Mbytes ($N=1$) to 2 Mbytes ($N=6$). There are many more repetitions of single words than there are repetitions of sequences of six words. It seems reasonable to parallelise the solution in terms of the value of N .

The next aspect of this particular problem is that of processing character strings. String processing is typically much less efficient than processing numbers and in particular integer values. The approach adopted is to extract each word from the text removing extraneous punctuation and convert the word to an integer value based on the sum of the character values for each character in the word. Each word is thus represented by an integer value which will be the same for every instance of the word. Obviously different words may generate the same integer value. The processing of word sequences of length greater than 1 can be achieved by adding up the sums of the word values for each word in the sequence. Thus word sequences can only possibly be equal if they generate the same value, which will speed up processing of multi-word sequences. Yet again some sequences will generate the same value even though the words making up the sequence are not the same.

24.2 Concordance Data Structures

Initially, the file is read, line-by-line, and the words extracted and extraneous punctuation removed and stored in a list, called word-List. At the same time the integer value of the word is stored in another list. This list will also contain the values for word sequences of length 1.

Each value of N will have its own data structure, stored in a list of such data structures. The next stage is to determine the sequence values of length 2 to N , simply by summing the values from the initial length 1 sequence in sets of the desired length. These values are saved in a list, one for each value of N , called a sequence-List.

The next stage is to process each of these lists to find the index or location in the text of each sequence value that has the same value. These are held in a map with the key entry given by the sequence value and the entry value comprising a list of the index values where that sequence value is found in the sequence-List. This is referred to as an equal-Key-Map.

The final stage is to process each of the N equal-Key-Maps to extract the concordance and write it to a file. The problem is that the sequence values held in the map may refer to different word sequences. Thus for each sequence value we can extract the index values and for each determine from the initial word list the words that make up the sequence. We can thus build up a map comprising the word sequence as key and a list containing the index values where that specific word sequence is found. Once the map has been constructed it can be written to a file corresponding to that part of the concordance related to word sequences of length N .

The data structures have been designed so that only the ones that are empty are written to in parallel, with each element indexed by the value of N. Such structures are subsequently read in parallel but that does not cause a problem. Thus initially, the word-List and the sequence-List for N=1 are the only ones that are written. In the next stage, only the sequence-List for N= 1 is read and the sequence-List for the remaining values up to N are written in parallel. In the next stage each of the sequence-Lists can be read in parallel as each of the equal-Key-Maps is written in parallel. Finally, each of the equal-Key-Maps can be read in parallel as the concordance is extracted and written to different files in parallel.

24.3 The Algorithm

The coding of the parallel version of the system is shown in Listing 24-1. A `timer` is defined {10}, which will be used to time the various phases of the processing. The file path for input text files and the location of output files are defined {11–13}. `N`, the sequence length is defined {14} as 6. The variable `minSeqLen` {15} specifies the minimum number of occurrences in a sequence for it to be printed; the value 2 means that a sequence will be printed provided there are two or more occurrences of a sequence in the file. The `timesFile` is a file to which times for each phase of the algorithm are written {17–22}.

What if you could build your future and create the future?

The innovation accelerator

One generation's transformation is the next's status quo. In the near future, people may soon think it's strange that devices ever had to be "plugged in." To obtain that status, there needs to be "The Shift".

.....Alcatel-Lucent 

www.alcatel-lucent.com/careers



In this example, six source texts are used, which will be processed one after the other {24} and `fileName` {25} gives the full path name of the required text file. The texts are ACM – A Changed Man by Thomas Hardy, TMM – The Manchester Marriage by Elizabeth Gaskell, WAD – Wives and Daughters by Elizabeth Gaskell, bible – the complete bible, 2bibles – two copies of the bible and 4bibles – four copies of the bible (Project Gutenberg, 2014). Lines {26-36} create a collection of print writers, `parPrintWriter` {28} that can be accessed in parallel during the last phase of the algorithm. This coding makes extensive use of the Groovy IO file methods.

The algorithm is run 8 times so that processing times can be averaged over a number of runs {37}. The first phase of the process is to create the `wordBuffer` {40}, which holds the words from the input file. The variable `NSequenceLists` {41} holds each of the sequence lists for each of the values of `N`. A `fileReader` is created for the input file {43, 44}. The reader will have to copy the source files from the `ChapterExamples` folders into an appropriate location on their computer system as indicated by and modified as necessary in lines {11-13}.

```
10 def timer = new CTimer()
11 def drive = "D"
12 def inRoot = drive + ":\Concordance\SourceFiles\"
13 def outRoot = drive + ":\Concordance\OutputFiles\"
14 def N = 6
15 def minSeqLen = 2
16
17 def timesFileName = outRoot + "Times" + N + minSeqLen + "_Par.txt"
18 def timesFile = new File(timesFileName)
19 if (timesFile.exists()) timesFile.delete()
20 def timesWriter = timesFile.newPrintWriter()
21 timesWriter.print "Par\tSource\tN\tminSeqLen\t"
22 timesWriter.println "Read\tGenerate\tEqualKeys\tConcordance\tTotal\tWords"
23
24 for (source in ["ACM", "TMM", "WAD", "bible", "2bibles", "4bibles"]){
25     def fileName = inRoot + source + ".txt"
26     def parOutFileName = []
27     def parOutFile = []
28     def parPrintWriter = []
29     for ( n in 1..N){
30         def parFileName = outRoot + source + N + minSeqLen + "_N_" + n +
31             "_Par.txt"
32         parOutFileName << parFileName
33         def parFile = new File(parFileName)
34         parOutFile << parFile
35         if (parFile.exists()) parFile.delete()
36         parPrintWriter << parFile.newPrintWriter()
37     }
38 }
39 for (run in 1..8){
```

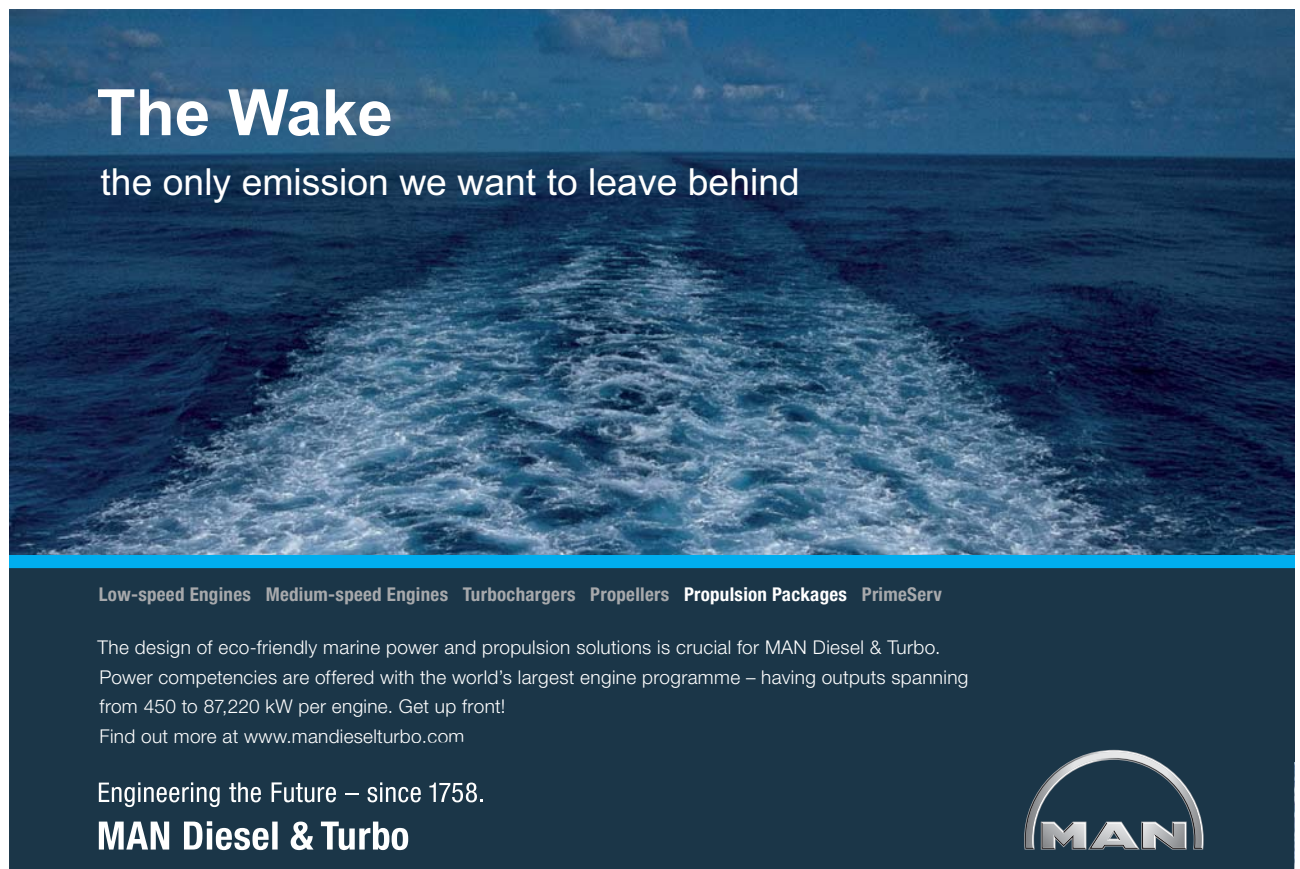
```
38     println "Processing: $fileName, N: $N, minSequenceLength: $minSeqLen"
39     def startTime = timer.read()
40     def wordBuffer = new ArrayList(10000)
41     def NSequenceLists = []
42     for ( n in 1..N) NSequenceLists[n] = new ArrayList(10000)
43     def fileHandle = new File (fileName)
44     def fileReader = new FileReader(fileHandle)
45     def wordCount = 0
46     fileReader.eachLine { line ->
47         def words = defs.processLine(line)
48         for ( w in words) {
49             wordBuffer << defs.removePunctuation(w)
50             NSequenceLists[1] << defs.charSum (wordBuffer[wordCount])
51             wordCount = wordCount + 1
52         }
53     }
54     def endRead = timer.read()
55     def procList1 = (2..N).collect {n ->
56         return new parSequencer( n:n, inList: NSequenceLists[1],
57                                 outList: NSequenceLists[n])}
58     new PAR(procList1).run()
59     def endGenSeq = timer.read()
60     def equalKeyMapList = []
61     for ( n in 1..N) equalKeyMapList[n] = [:]
62     def procList2 = (1..N).collect { n ->
63         return new parFindEqualKeys ( words: (wordCount - 1),
64                                     startIndex: 0,
65                                     inList: NSequenceLists[n],
66                                     outMap: equalKeyMapList[n])}
67     new PAR(procList2).run()
68     def endFindEqualKeys = timer.read()
69     def procList3 = (1..N).collect { n ->
70         return new parExtractConcordance ( equalMap: equalKeyMapList[n], n: n,
71                                         startIndex: 0, words: wordBuffer,
72                                         minSeqLen: minSeqLen,
73                                         printWriter: parPrintWriter[n-1])}
74     new PAR(procList3).run()
75     def endConcordance = timer.read()
76     def readTime = endRead - startTime
77     def genTime = endGenSeq - endRead
78     def equalKeysTime = endFindEqualKeys - endGenSeq
79     def concordanceTime = endConcordance - endFindEqualKeys
80     def totalTime = endConcordance - startTime
81     timesWriter.print "$run\t$source\t$N\t$minSeqLen\t\t"
82     timesWriter.println "$readTime\t$genTime\t\t$equalKeysTime" +
83         "\t\t$concordanceTime\t\t$totalTime\t$wordCount"
84     print "Par\tSource\tN\tminSeqLen\t"
```

```
84     println "Read\tGenerate\tEqualKeys\tConcordance\tTotal\tWords"
85     print  "$run\t$source\t$N\t$minSeqLen\t\t"
86     println"$readTime\t$genTime\t\t$equalKeysTime" +
87           "\t\t$concordanceTime\t\t$totalTime\t$wordCount"
88   }
89   println ""
90   timesWriter.println "\n\n\n"
91   timesWriter.flush()
92 }
93 timesWriter.close()
```

Listing 24-1 Parallel Version of the Concordance Algorithm for a Multi-core Processor

The class `defs` defines a number of static methods that are used during the algorithm. The file is read in a line at a time and the words extracted using the `processLine` method {47}.

Punctuation is removed from the word {49}, which is then appended to the `wordBuffer`. The method `charSum` {50} is used to calculate the integer value corresponding to the word and this value is appended to the `List NSequenceLists[1]`.




The Wake

the only emission we want to leave behind

Low-speed Engines Medium-speed Engines Turbochargers Propellers Propulsion Packages PrimeServ

The design of eco-friendly marine power and propulsion solutions is crucial for MAN Diesel & Turbo. Power competencies are offered with the world's largest engine programme – having outputs spanning from 450 to 87,220 kW per engine. Get up front!
Find out more at www.mandieselturbo.com

Engineering the Future – since 1758.
MAN Diesel & Turbo



The next phase of the algorithm is to create a set of parallels {55-58} that determines the sequence values for each of the values 2 to N. The process `parSequencer` accesses the `inList` parameter for reading. The `outList` parameter is written to and will contain the sequence list for the value of the parameter `n`.

The next phase of the algorithm populates `equalKeyMapList` {60} by means of a parallel of `parFindEqualKeys` processes {63-65}. The parameter `words` indicates the number of words in the input file. The parameter `startIndex` indicates the subscript of the first word in the word buffer, which in this case is 0. The parameter `inList` is a read parameter and refers to the `n`th sequence list created in the previous phase. The process creates the `n`th equal-Key-Map as described previously as the parameter `outMap`.

The final phase of the algorithm is to write the concordance to the output files. This is achieved by means of a collection of parallel instances of `parExtractConcordance` processes {68-72}. The process reads the `n`th `equalMapListMap` parameter which is used as the basis of the concordance. The `words` parameter is the `wordBuffer` created during the first phase of the algorithm. The `printWriter` parameter is passed the `n`th `parPrinterWriter` which is used to print the concordance to file.

Finally, the times for each of the phases are calculated {74-87} and printed to the `timesWriter` file after each run. Once all the runs for a particular input file are completed then the next input file is processed.

A sequential version of the algorithm is easily achieved by replacing each of the parallel constructs by a for loop that iterates over the values of `n`. For example, the process `parSequencer` shown in Listing 24-2 simply invokes the method `sequencer`, defined in `defs`, within a process' `run` method.

```
10 class parSequencer implements CSProcess {
11     def n
12     def inList
13     def outList
14
15     void run() {
16         defs.sequencer(n, inList, outList)
17     }
18 }
```

Listing 24-2 The parSequencer Process

The sequential version simply invokes the method `sequencer` directly in a for loop that iterates over the data structures one after the other. The same mechanism is used for each of the phases of the algorithm. In the sequential version only one core is used in effect, even though the processing is moved between cores to improve heat dissipation.

24.4 Analysis of Total Time Results

The data for the six input files is presented in Table 24-1. The Source File size in Kbytes gives the size of the input file. The output file size is the size of the single output file produced by the sequential algorithm. The sum of the output files produced by the parallelised version is very similar. These two values are then added together to show the total input / output generated by the specific text. The words row gives the number of words found in each source file. There are two times, in seconds, presented which are derived from the averaging of the algorithm's execution over 8 runs. The Seq Time is the time taken to execute the sequential version of the algorithm. The Par Time is the time taken to run the parallel version on a quad core processor. It can be noted that even though the parallel multi-core algorithm used four cores the best Speed-up was only 2.51, where the ideal would be close to 4. Speed-up is given by the Sequential time divided by the Parallel time. Efficiency is given by the Speed-up divided by the number of available processors, which in this case is 4. The closer the Efficiency is to 1.0 the better. For this example we see that the Efficiency reaches a peak of 0.63 for the bible text. We can surmise that the drop in performance on either side of the peak is due to the amount of input and output associated with this application.

In all executions of the algorithm the value of N was 6 and only sequences that had two or more occurrences were output, except for 2bibles and 4bibles where the number of repetitions had to be at least 3 and 5 respectively.

Values	ACM	TMM	WAD	bible	2bibles	4bibles
Input File KB	37	62	1,458	4,681	9,362	18,723
Output File KB	65	134	5,487	26,107	45,464	85,364
Total I/O KB	102	196	6,945	30,788	54,826	104,087
Words	6,418	11,354	268,429	802,317	1,604,634	3,209,268
Seq Time secs	0.14	0.12	3.15	9.94	20.42	46.46
Par Time secs	0.10	0.05	1.31	3.96	12.41	26.70
Speed-up	1.42	2.43	2.41	2.51	1.65	1.74
Efficiency	0.36	0.61	0.60	0.63	0.41	0.43

Table 24-1Base data

In order to evaluate the above data we can calculate the ratio of each value relative to the ACM data. This leads to the representation shown in Figure 24-1.

With small texts no effect can be observed but as we move to larger texts we see that the sequential time closely follows the increase in the number of words, even though the amount of file input/output is increasing faster. More importantly the increase in the parallel version shows a slower rate of increase than the sequential version and thus there is some benefit resulting from the parallelisation of the algorithm.

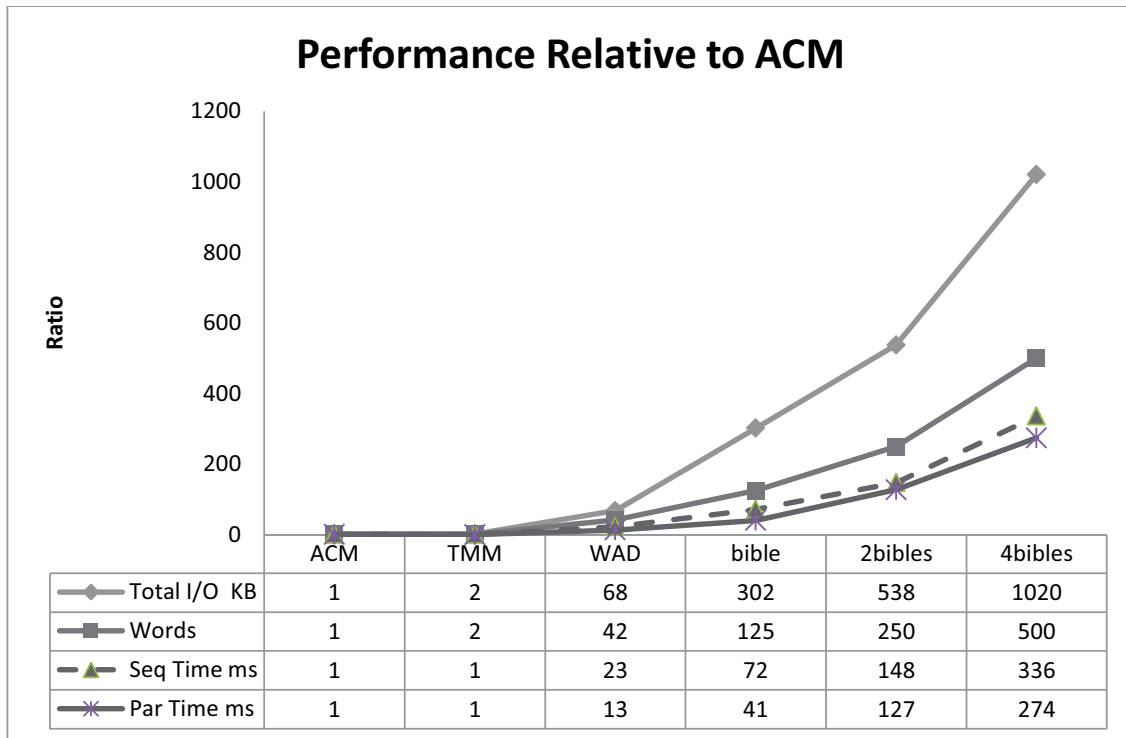


Figure 24-1 Performance Relative to ACM

24.5 Analysis of Algorithm Phases

The coding shown in Listing 24-1 determined the time taken for each of the phases of the algorithm. These times are summarised in Table 24-2, as before the times are averaged over 8 runs. The algorithms were run on a quad core machine so that the maximum attainable speedup is 4.0.

In general, we observe that the Read phase shows no speedup of the parallel solution over the sequential solution. This is not surprising as this phase is limited by the ability to read the source file. The amount of speedup for the Generate phase varies, with a maximum speed-up of 4.81. Of more interest is that for the short text (ACM) both the Read and Generate phases run slower in the parallel version than the sequential version. This may be accounted for by the time taken to set up the parallel.

The Equal Keys phase is the longest phase in terms of processing time and is the one which should result in the greatest parallel speed-up. This can in fact be observed with speedup of nearly 4.0 in two cases. The variation can be accounted for by the proportion of sequences of various lengths that occur in the different texts.

The final Concordance phase is the one that creates the output files. This phase is thus governed more by the ability of the underlying file system to manage multiple file handles rather than raw processing ability. Perhaps the most important conclusion to take from this exercise is that instrumenting the algorithm is crucial so as to be able to observe which parts are the slowest and which ones may benefit from parallelisation efforts. Thus there is little point in expending effort on further parallelisation of the Read phase but efforts to improve the Concordance phase may be beneficial, especially for larger texts. This will be explored in a more detail in the next section where the scaling of the solution is discussed. In particular, further improvement may be achieved if the file output for each value of N takes place on a different processor.

The advertisement features a central graphic on the left with three stylized human figures surrounded by gears, all enclosed within a circular arrow indicating a cycle. To the right, the text 'UNLEASHING CHANGE MANAGEMENT' is written in large, bold, blue capital letters. Below this, the dates 'OCTOBER 18 & 19, 2018' and the location 'DE RODE HOED AMSTERDAM' are listed. The bottom of the ad shows a silhouette of an Amsterdam skyline with a windmill and a bridge. In the bottom left corner, the text 'Global Executive Events' is visible.

Text	Words	Style		Read	Generate	Equal Keys	Concordance
ACM		Seq		31	10	23	74
		Par		35	10	6	47
			Speed Up	0.89	1.00	3.98	1.58
	6418						
TMM		Seq		8	10	27	74
		Par		12	2	12	23
			Speed Up	0.66	4.81	2.33	3.17
	11354						
WAD		Seq		171	322	534	2127
		Par		176	121	134	878
			Speed Up	0.97	2.66	3.98	2.42
	268429						
Bible		Seq		484	854	1601	7004
		Par		477	482	416	2586
			Speed Up	1.02	1.77	3.85	2.71
	802317						
2Bibles		Seq		1284	1808	3379	13946
		Par		1276	1031	1039	9059
			Speed Up	1.01	1.75	3.25	1.54
	1604634						
4Bibles		Seq		2296	5050	7667	31449
		Par		2228	3907	4495	16075
			Speed Up	1.03	1.29	1.71	1.96
	3209268						

Table 24-2 Times (milliseconds) for Each Phase of the Algorithm

24.6 Dealing with Larger Data Sets

Say we had a much larger text to process that could not fit into the memory of the system being used. The solution then might be to distribute the algorithm over a number of network connected workstations. A possible architecture is shown in Figure 24-2, where the value of N is 5. It is assumed that all processes are executed on their own workstation. All the connections are assumed to be network channels; not all of the connections are shown.

The Reader process will process the input file, a line at a time, extracting words. It will place these words into a fixed length array, word-buffer. The content will be adjusted so that the last N-1 words of one word-buffer are the same as the first in the next word-buffer. This means that we can deal with the start and end of each word-buffer without swapping words between processing Nodes. The Reader process will send each word-buffer in turn to a different processing Node. The processing Node will then remove punctuation and also calculate the integer sum of each word in the word-buffer. The processing Node will also need to be informed of the start index of the word-buffer relative to the start of the input file. The size of the word-buffer should be optimised so that a processing Node is ready to receive the next word-buffer, having just completed the above processing. This will be a function of the speed of the processors and of the network and will have to be found by experiment.

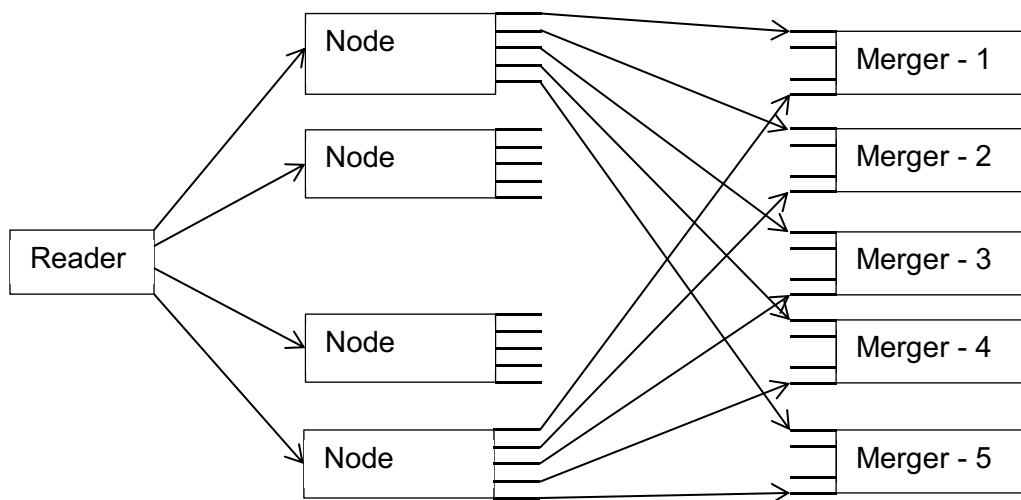


Figure 24-2 Architecture of a Scalable Networked Solution to the Concordance Problem

Once the Reader process has read the complete input file and distributed it to the processing Nodes it can inform the Nodes, which can now start to process each word buffer in a manner similar to that described in section 24.3. In this case, however, instead of outputting the concordance directly to a file, the processing Node will need to save a partial concordance for each word-buffer in its local memory for each value of 1 to N. Once this phase is completed the next sort-merge phase can begin. Each processing Node will comprise a Concordance Producer Process (Worker) together with a number of Output or Sorter processes; one for each value from 1 to N as shown in Figure 24-3, where N = 5 is assumed.

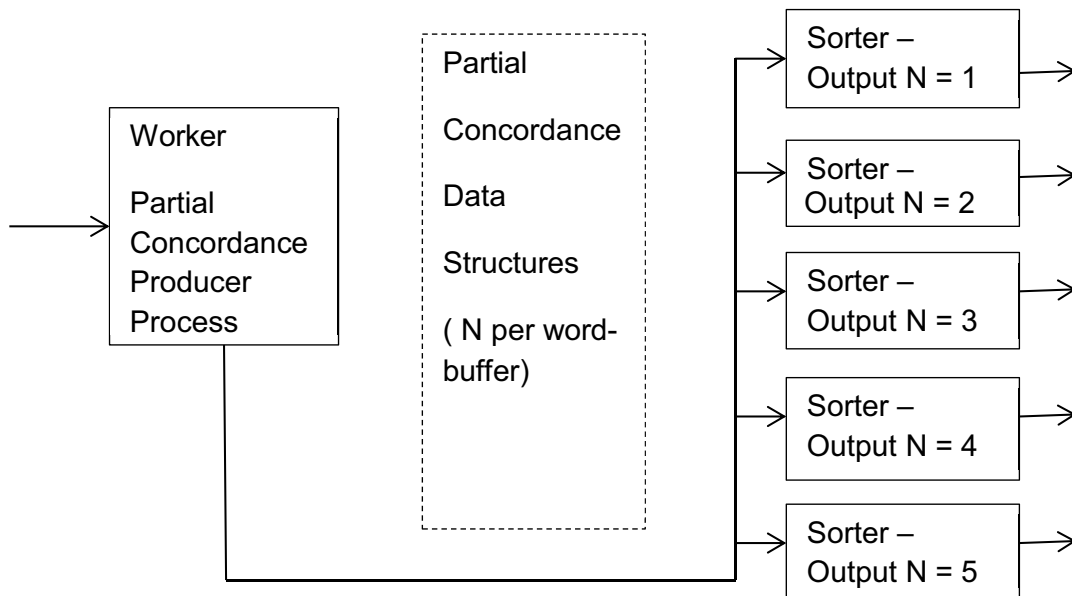


Figure 24-3 The Internal Architecture of a Processing Node

bookboon.com

Corporate eLibrary

See our Business Solutions for employee learning

[Click here](#)

Management Time Management

Problem solving Self-Confidence Effectiveness

Project Management Goal setting Motivation Coaching



The Worker – Partial Concordance Producer Process writes the partial concordances into the shared data area shown. The partial concordance for each value of N is held in a map data structure where the key is the sequence value, as previously described and the entry is the concordance entries for that sequence value. Each word-buffer has its own partial concordance for each value of N. The last part of the Worker – Partial Concordance Producer Process is to determine the ascending sorted order of sequence value for each partial concordance. Once it has completed, it sends a signal to each of the Sorter – Output processes indicating they can now start to read the Partial Concordance Data Structures.

Each Sorter – Output process accesses the partial concordances in sequence value order. It is thus able to create a complete concordance for all the word-buffers held by the Processing node. This can be undertaken in parallel for each value from 1 to N. The Sorter – Output process then sends a complete Concordance entry for its word-buffers to the Merger process shown in Figure 24-2. Each concordance entry will be preceded by its sequence value, which will be created in sorted order.

The Merger process can now merge the concordances from each Processing node. The Merger reads concordance entries from each processing Node using the sequence value to ensure that the concordance entries from the different processing Nodes are merged correctly. The Merger process can write the file associated with that value of N as it processes each partial concordance. The processing Node sends a special sentinel value to indicate that all the partial concordances for that node have been communicated. Once a Merger process receives all the sentinel values it can complete the writing of the output file and terminate. This architecture results in each of the N concordance files being written to a different workstation. However, if there is a network accessible storage facility these files can be written to that device, and each file will still be written in parallel provided the network storage capability has the ability to process multiple files at the same time.

An analysis of the architecture indicates that the reading of the input file is undertaken by one node. This could be seen as a bottle-neck, but because we output the word-buffers to other nodes for subsequent processing, this means that most of the file input is undertaken in parallel with some of the subsequent processing. Once the word-buffers have been created at each processing Node the available cores can be set to the task of creating the Partial Concordance Data Structures. Further, this activity is undertaken at each such node in parallel. The Merge phase also involves many parallel nodes; namely all the processing Nodes and Merger nodes. In addition the available cores on the Processing node can be utilised by the Sorter – Output processes.

24.7 Implementation of the Scalable Architecture

The architecture uses two capabilities previously described in Chapter 23. First, the use of a Java `main()` called `Launcher` to enable the required code to be executed at each node in the distributed system. Secondly, the use of the process loading architecture to distribute processes from a single host to all the nodes used in the distributed system. This makes use of the system much simpler. These aspects are not discussed further but the required code is available in the folders `ChapterExamples/c24.*`, including the batch files required to invoke the host and node processes. In this section we shall discuss the processes that are run on each of the nodes in the distributed system. It will not discuss the way in which the processes are communicated from the host to the other nodes, nor the specific detail of creating the net channel connections between the nodes.

24.7.1 The Reader Process

The Reader process, Listing 24-3, has a number of properties {12–19}, the majority of which deal with the operating environment. The only property associated with the parallel operation is the `ChannelOutputList outChannels` {12}, which contains the channels used to connect the Reader process to the Node processes. The property `inRoot` {13} provides the path to the source files. The value of `N` {14} is specified. The property `blockLength` {15} indicates the number of words in each block. This value can be varied depending on the number of processing Nodes and the speed of the distributed system's interconnect and has to be found by experiment. The value of `runs` {16} is the number of the times the algorithm will be run, over which an average time can be found. The property `sourceList` {17} is a list of files that are to be processed. The property `timeRoot` {18} gives the path to the folder into which the timing output files will be written. Each execution of the system will be identified by a different `runId` {19} so that the various versions of the system can be easily identified. This property is used when forming file names.

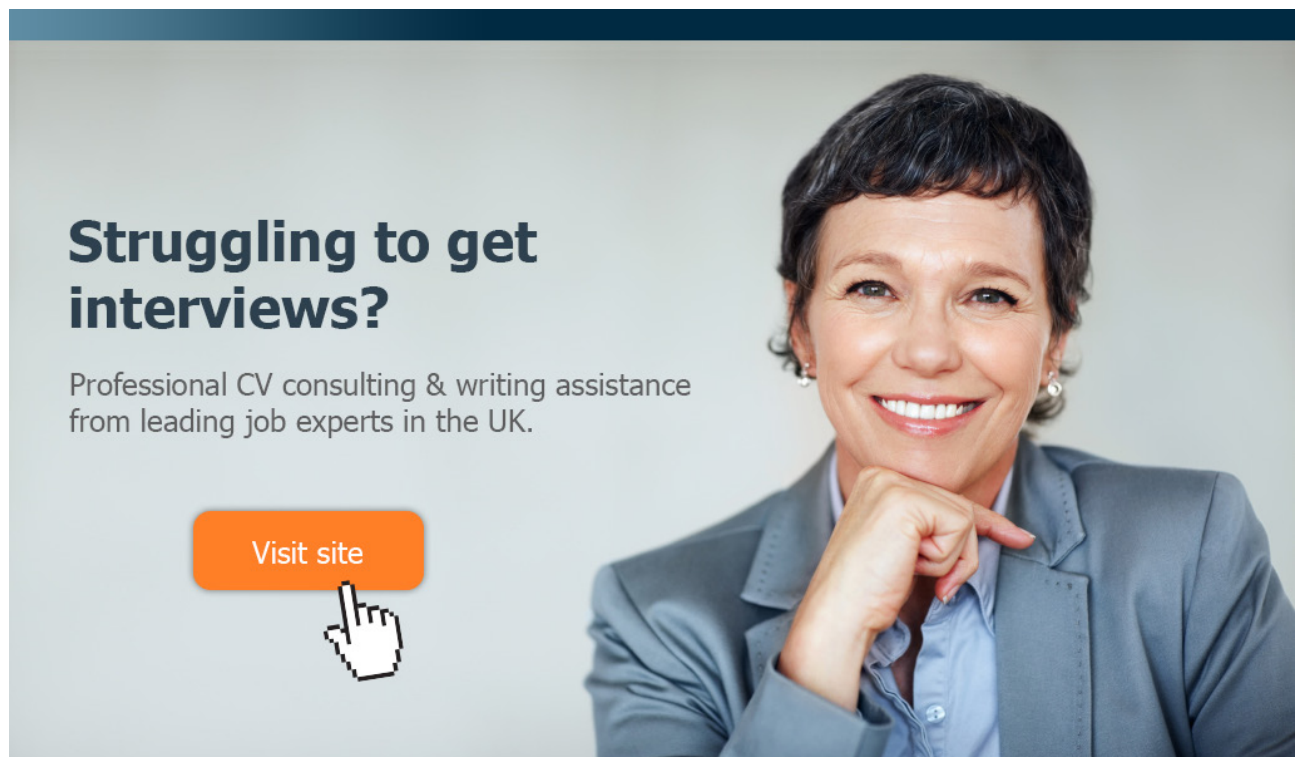
The first part {22–25} of the process deals with the creation of `timeWriter`, a `PrintWriter` to which the various times associated with the processing phases can be written. A `timer` is then created {26}. The number of `nodes` in the system is determined from the size of `outChannels` {27}. Finally the effective length of each block is determined as `blockStride` {28}. This is less than the length of a block because the `N-1` words at the end of one block are repeated at the start of the next block.

The process iterates through each `source` in the `sourceList` {29}. The `fileName` containing the source is created {30} and some timing information is written both to the console and the `timeWriter`. The process then repeats the processing for each `run` {35}.

At the start of each run a number of variables are defined together with a `FileReader` for the source file {39}. The array `wordBuffer` {37} is used to hold the words that are extracted from the source file. The variable `globalIndex` {40} is used to hold the start index of each `wordBuffer` within the whole file, while `localIndex` {41} maintains the subscript of each word within `wordBuffer` and is reset to zero once a `wordBuffer` has been written to a Node. The variable `currentNode` {42} keeps track of the next Node to which a `wordBuffer` should be written.

Each line of the file is read {45} and processed to extract the words {46} in the line. The words are then appended, in turn, to the `wordBuffer` until it contains `blockLength` entries {50}. At this point a `WordBlock` is created from the `wordBuffer` and `globalIndex` {51–52}. After which, `globalIndex` is incremented {53} and the `wordBlock` is written to the `currentNode` {54}. If this is the first block to be written we note the time this takes place {55–58}. The value of `currentNode` is then incremented {59}.


The next phase of processing deals with the initial filling of the next `wordBuffer` with the last few words of the previous one {60–69}. An `overlapBuffer` is created {60} into which the last few words of `wordBuffer` are copied {61–63}. The `wordBuffer` is then emptied {64} and the `overlapBuffer` copied into it {66–69} modifying the value of `localIndex` accordingly {65, 68}.



Struggling to get interviews?

Professional CV consulting & writing assistance from leading job experts in the UK.

Visit site

 Take a short-cut to your next job!
Improve your interview success rate by 70%.

 **TheCVagency**
Visit theagency.co.uk for more info.

  **Click on the ad to read more**

Once the end of the source file is reached there may be some words remaining in the `wordBuffer` and so the next phase is to send the remaining words to the next node {74–77}. Finally, a Sentinel is sent to each of the Nodes {78} informing them that the complete file has been processed and they can then start the next phase of processing.

Finally, salient times are written to the console and to the file of times. The difference between the `startTime` and the `beginTime` is that the former records when the run starts and the latter when the first block of the run has been written. The difference between these times will be the time it takes for the rest of the processing in the Nodes and Mergers to take place from the previous run.

```
10 class Reader implements CSProcess {
11
12     def ChannelOutputList outChannels
13     def inRoot
14     def N = 6
15     def blockLength = 5000
16     def runs = 8
17     def sourceList
18     def timeRoot
19     def runId
20
21     void run() {
22         def timeFileName = timeRoot + runId + "_R_" + "_times.txt"
23         def timeHandle = new File(timeFileName)
24         if (timeHandle.exists()) timeHandle.delete()
25         def timeWriter = timeHandle.newPrintWriter()
26         def timer = new CTimer()
27         def int nodes = outChannels.size()
28         def int blockStride = blockLength - N + 1
29         for (source in sourceList) {
30             def fileName = inRoot + source + ".txt"
31             println "READER - Processing: $fileName, N: $N," +
32                 "block length: $blockLength, nodes: $nodes, runs: $runs"
33             timeWriter.println "READER - Processing: $fileName, N: $N, " +
34                 "block length: $blockLength, nodes: $nodes, runs: $runs"
35             for (run in 1..runs) {
36                 def startTime = timer.read()
37                 def wordBuffer = new ArrayList(blockLength)
38                 def fileHandle = new File(fileName)
39                 def fileReader = new FileReader(fileHandle)
40                 def globalIndex = 0
41                 def localIndex = 0
42                 def int currentNode = 0
43                 def firstWrite = true
44                 def beginTime
```

```
45     fileReader.eachLine { line ->
46         def words = defs.processLine(line)
47         for ( w in words) {
48             wordBuffer << w
49             localIndex = localIndex + 1
50             if (localIndex == blockLength) {
51                 def wordBlock = new WordBlock( startSubscript: globalIndex,
52                                                 words: wordBuffer)
53                 globalIndex = globalIndex + blockStride
54                 outChannels[currentNode].write(wordBlock)
55                 if (firstWrite) {
56                     beginTime = timer.read()
57                     firstWrite = false
58                 }
59                 currentNode = (currentNode + 1) % nodes
60                 def overlapBuffer = []
61                 for (overlap in blockStride..(blockLength - 1)) {
62                     overlapBuffer << wordBuffer[overlap]
63                 }
64                 wordBuffer = []
65                 localIndex = 0
66                 for (ow in overlapBuffer) {
67                     wordBuffer << ow
68                     localIndex = localIndex + 1
69                 } // end for ow
70             } // end if
71         } // end for words
72
73     } // end eachLine
74     def wordBlock = new WordBlock( startSubscript: globalIndex,
75                                   last: true,
76                                   words: wordBuffer)
77     outChannels[currentNode].write(wordBlock)
78     for ( n in 0..< nodes) outChannels[n].write(new Sentinel())
79     def endTime = timer.read()
80     def words = localIndex + globalIndex
81     println "READER, $source, $run, ${endTime - startTime}, " +
82            "${endTime - beginTime}, $words"
83     timeWriter.println "READER, $source, $run, ${endTime - startTime}, " +
84            "${endTime - beginTime}, $words"
85     } // end for run
86 } // end for source
87 timeWriter.flush()
88 timeWriter.close()
89 } // end void run
90 }
```

Listing 24-3 The Reader Process

24.7.2 The Node Process

Listing 24-4 shows the `Node` process which is itself a parallel formed from one `Worker` process and `N` `Sorter` processes. A `Node` has an input channel from the `Reader` process referred to as `nodeInChannel` {12}. Each `Node` is connected to each of the `Merger` processes and this set of output channels are held in the `ChannelOutputList toMergers` {13}. A channel called `startSortPhase` is created {22}, which will be used to signal that the `Sorter` processes can begin once the `Worker` process has completed its processing.

The process iterates through the `sourceList` {15, 23} and the `runs` {16, 24}. The variable `sequenceBlockList` {26} is used to hold the shared data that is accessed initially by the `Worker` process to write to the required data structures into the shared data and subsequently by the `Sorter` processes to read from those data structures in parallel. For each run a new network of processes is created comprising the single `Worker` process and the `N` `Sorter` processes {27–43}. Once the network terminates the processing times for the `Node` are determined and written to the console and the time file {44–46}.



e-learning for kids

- The number 1 MOOC for Primary Education
- Free Digital Learning for Children 5-12
- 15 Million Children Reached

About e-Learning for Kids Established in 2004, e-Learning for Kids is a global nonprofit foundation dedicated to fun and free learning on the Internet for children ages 5 - 12 with courses in math, science, language arts, computers, health and environmental skills. Since 2005, more than 15 million children in over 190 countries have benefitted from eLessons provided by EFK! An all-volunteer staff consists of education and e-learning experts and business professionals from around the world committed to making difference. eLearning for Kids is actively seeking funding, volunteers, sponsors and courseware developers; get involved! For more information, please visit www.e-learningforkids.org.


```
10 class Node implements CSProcess {
11
12     def ChannelInput nodeInChannel
13     def ChannelOutputList toMergers // N of these
14     def N = 0
15     def sourceList
16     def runs
17     def node
18     def timeWriter
19
20     void run(){
21         def timer = new CTimer()
22         def startSortPhase = Channel.one2any()
23         for ( s in sourceList){
24             for ( r in 1 .. runs){
25                 def startTime = timer.read()
26                 def sequenceBlockList = [] // holds each of the sequence blocks
27                 def worker = new Worker( N: N, inChannel: nodeInChannel,
28                                         ssp: startSortPhase.out(),
29                                         sbl: sequenceBlockList, source: s,
30                                         run: r, node:node,
31                                         timeWriter: timeWriter )
32                 def sorters = (1..N).collect{sn ->
33                     new Sorter(Nvalue: sn,
34                               startChannel: startSortPhase.in(),
35                               toMerger: toMergers[sn-1],
36                               sbl: sequenceBlockList,
37                               source: s,
38                               run: r,
39                               node: node,
40                               timeWriter: timeWriter )
41                 }
42                 def network = sorters + worker
43                 new PAR(network).run()
44                 def endTime = timer.read()
45                 println "NODE, $node, $s, $r, ${endTime - startTime}"
46                 timeWriter.println "NODE, $node, $s, $r, ${endTime - startTime}"
47             } // end runs
48         } // end sources
49         timeWriter.flush()
50         timeWriter.close()
51     } // end run()
52 }
```

Listing 24-4 The Node Process

24.7.3 The Worker Process

The Worker process is shown in Listing 24-5 and has a very close relationship to the single machine algorithm discussed in section 24.3. It can be seen that similar or identical processes are called as in the single machine version. The differences arise due to the way the data is stored in the shared data area in the Node process. In the single machine version the words are stored in a single data structure. In the distributed version each block of words is going to be stored in its own data structure. In addition, each block of words will also have all the associated maps and indexes stored for that block with that block. The list of such data structures is referred to as the property `sbl` {13}.

The processes iterates over incoming messages from the reader process until it reads a `Sentinel` object {26, 28}. The words are processed to remove punctuation and to calculate the integer value corresponding to the word {37–41}. A new version of the Sequencer process has been created {45–48} which deals with the fact that there are individual word blocks with different starting subscripts relative to the whole file and that the last block of words will probably not be full. Its function however remains the same, which is to calculate the sequence value for the words in the block for each value of `N`. This operation is undertaken in parallel for each value of `N`.

The next phase uses the same process as in the single machine version to create the `equalKeyMaps` {55–61}. Yet again this can be undertaken in parallel for each value of `N`.

The final phase, {66–72} is similar to the single machine version in that it generates concordance but instead of writing them to file directly stores the part concordances in a data structure called `equalWordMapList`. This change is required because we are creating partial concordance for each block of words on each node. Once the data structures have been created they can be appended as a new `SequenceBlock` to the shared data structure `sbl` {73–77}. During the processing various times are calculated and added together to determine how long each phase of the algorithm takes in total for all the input word blocks. Finally, once all the words sent to this Node have been processed, the relevant times can be written to the console and the time file {82–87}. The last action of the process is to send a `Sentinel` to each of the Sorter processes so they can commence processing {88}.

```
10 class Worker implements CProcess {
11
12     def N = 0
13     def sbl // the reference to the sequenceBlockList
14     def source
15     def run
16     def ChannelInput inChannel
17     def ChannelOutput ssp // connection to startSortPhase
18     def node
19     def timeWriter
20
```

```
21 void run(){
22     def timer = new CTimer()
23     def endTime
24     def timesList = []
25     for (t in 0..< 4) timesList[t] = 0
26     def o = inChannel.read()
27     def beginTime = timer.read()
28     while ( ! (o instanceof Sentinel)) {
29         def startTime = timer.read()
30         def startSub = o.startSubscript
31         def punctuatedWords = o.words
32         def lastBlock = o.last
33         def NSequenceLists = []
34         def wordCount = 0
35         for ( n in 0..N) NSequenceLists[n] = new ArrayList(1000)
36         def wordBuffer = new ArrayList(punctuatedWords.size())
37         for ( w in punctuatedWords){
38             wordBuffer << defs.removePunctuation(w)
39             NSequenceLists[0] << defs.charSum (wordBuffer[wordCount])
40             wordCount = wordCount + 1
41         } // end for punctuatedWords
42         def endRemove = timer.read()
43         timesList[0] = timesList[0] + endRemove - startTime
44         def procList1 = (1..N).collect {n ->
```



FACTCARDS

Are you working in academia, research or science? And have you ever thought about working and moving to the Netherlands?

Arriving 33

Living 50

Studying 51

Working 101

Research 50

Factcards.nl offers all the **information** that you need if you wish to proceed your **career** in the **Netherlands**.

The information is ordered in the categories arriving, living, studying, working and research in the Netherlands and it is freely and easily accessible from your smartphone or desktop.

VISIT FACTCARDS.NL

```
45         return new parMultiSequencer( Nmax: N, n:n,  
46                                     baseList: NSequenceLists[0],  
47                                     outList: NSequenceLists[n],  
48                                     lastBlock: lastBlock)  
49     }  
50     new PAR(procList1).run()  
51     def endSequencer = timer.read()  
52     timesList[1] = timesList[1] + endSequencer - endRemove  
53     def equalKeyMapList = []  
54     for ( n in 1..N) equalKeyMapList[n] = [:]  
55     def procList2 = (1..N).collect { n ->  
56         return new parFindEqualKeys ( words: (wordCount - 1),  
57                                     startIndex: startSub,  
58                                     inList: NSequenceLists[n],  
59                                     outMap: equalKeyMapList[n])  
60     }  
61     new PAR(procList2).run()  
62     def endEqualKeys = timer.read()  
63     timesList[2] = timesList[2] + endEqualKeys - endSequencer  
64     def equalWordMapList = []  
65     for ( n in 1..N) equalWordMapList[n] = [:]  
66     def procList3 = (1..N).collect { n ->  
67         return new parExtractUniqueSequences (   
68             equalMap: equalKeyMapList[n], n: n,  
69             startIndex: startSub, words: wordBuffer,  
70             equalWordMap: equalWordMapList[n] )  
71     }  
72     new PAR(procList3).run()  
73     sbl << new SequenceBlock (startSubscript: startSub,  
74                             words: wordBuffer,  
75                             NSequenceLists: NSequenceLists,  
76                             equalKeyMapList: equalKeyMapList,  
77                             equalWordMapList: equalWordMapList)  
78     o = inChannel.read()  
79     endTime = timer.read()  
80     timesList[3] = timesList[3] + endTime - endEqualKeys  
81 } // end while not Sentinel  
82 println "WORKER, $source, $run, ${timesList[0]}, " +  
83         "${timesList[1]}, ${timesList[2]}, ${timesList[3]}, "+  
84         "${endTime - beginTime}"  
85 timeWriter.println "WORKER, $source, $run, ${timesList[0]}, "+  
86         "${timesList[1]}, ${timesList[2]}, ${timesList[3]}, "+  
87         "${endTime - beginTime}"  
88     for ( n in 1..N) ssp.write(new Sentinel())  
89 } // end run()  
90 }
```

Listing 24-5 The Worker Process

24.7.4 The Sorter Process

The Sorter process is shown in Listing 24-6.

```
10 class Sorter implements CProcess {
11
12     def Nvalue = 0
13     def ChannelInput startChannel
14     def ChannelOutput toMerger
15     def sb1 // the reference to the shared sequenceBlockList
16     def source
17     def run
18     def node
19     def timeWriter
20
21     void run(){
22
23         def union = {s1, s2 ->
24             s2.each{v ->
25                 if ( !(s1.contains(v))) s1 << v
26             }
27         } // end union
28
29         def timer = new CTimer()
30         def sbKeys = []
31         startChannel.read()
32         def startTime = timer.read()
33         sb1.each{ sb ->
34             def ewmN = sb.equalWordMapList[Nvalue]
35             def mapKeys = ewmN.keySet()
36             mapKeys.each { mk ->
37                 union (sbKeys, mk)
38             }
39         }
40         def sortedKeys = sbKeys.sort()
41         def sortedTime = timer.read()
42         sortedKeys.each { keySV ->
43             def compositeWordSSMap = [:]
44             sb1.each { sb ->
45                 def wmEntry = sb.equalWordMapList[Nvalue].get(keySV, [] )
46                 wmEntry.each {
47                     def wordKey = it.key
48                     def subScripts = it.value
49                     def existingSS = compositeWordSSMap.get(wordKey, [])
50                     existingSS << subScripts
51                     compositeWordSSMap.put(wordKey, existingSS)
52                 } // end of each wmEntry
```

```
53     } // end of each sbl
54     def partConcordance = new PartConcordance( seqVal: keySV,
55                                               entryMap: compositeWordSSMap)
56     toMerger.write(partConcordance)
57 } // end of each sbKeys
58 def endTime = timer.read()
59 toMerger.write(new Sentinel())
60 println "SORTER, $source, $run, $node, $Nvalue, ${sortedTime - startTime}, " +
61        "${endTime - sortedTime}, ${endTime - startTime}"
62 timeWriter.println "SORTER, $source, $run, $node, $Nvalue, " +
63                  "${sortedTime - startTime}, ${endTime - sortedTime}, " +
64                  "${endTime - startTime}"
65 } // end of run()
66 }
```

Listing 24-6 The Sorter Process

The input channel `startChannel {13}` is the channel upon which the signal to start processing is received from the `Worker` process. The output channel `toMerger {14}` is the channel used by `Sorter` to write information to the `Merger` that is collecting data for `NValue {12}`. A method, called `union {23–27}`, is defined that undertakes the set union operation.



Brain power

By 2020, wind could provide one-tenth of our planet's electricity needs. Already today, SKF's innovative know-how is crucial to running a large proportion of the world's wind turbines.

Up to 25 % of the generating costs relate to maintenance. These can be reduced dramatically thanks to our systems for on-line condition monitoring and automatic lubrication. We help make it more economical to create cleaner, cheaper energy out of thin air.

By sharing our experience, expertise, and creativity, industries can boost performance beyond expectations. Therefore we need the best employees who can meet this challenge!

The Power of Knowledge Engineering

Plug into The Power of Knowledge Engineering.
Visit us at www.skf.com/knowledge

SKF

The process waits until it receives an input on its `startChannel` {31}. The process then iterates through all the shared sequence blocks obtaining the `equalWordMapList` elements for `NValue` {34} to form the union of all the key sets present on this Node {33–39}. The key set is then sorted into ascending order {40}. The keys are the sequence values that were previously created. The sorting of the keys is crucial because this will be used by the subsequent Merger process to determine whether or not it is going to receive specific word strings from a Node.

The next phase {42–57} of the algorithm is to take each key value in turn and then to iterate through the shared sequence block lists to create a composite entry for each sequence value. A sequence value probably relates to more than one string of words so each word map entry corresponding to a sequence value is itself a map of a word string as key together with a list of subscripts where the word string is found. Thus this phase combines data from all the shared sequence blocks to create a composite word map entry that indicates the subscript where each word string corresponding to the sequence value is found in all the blocks stored at this node. This data structure is formed into a new `PartConcordance` {54–55} and then written to the Merger process associated with this node's `NValue`.

24.7.5 The Merger Process

The coding for the Merger Process is shown in Listing 24-7.

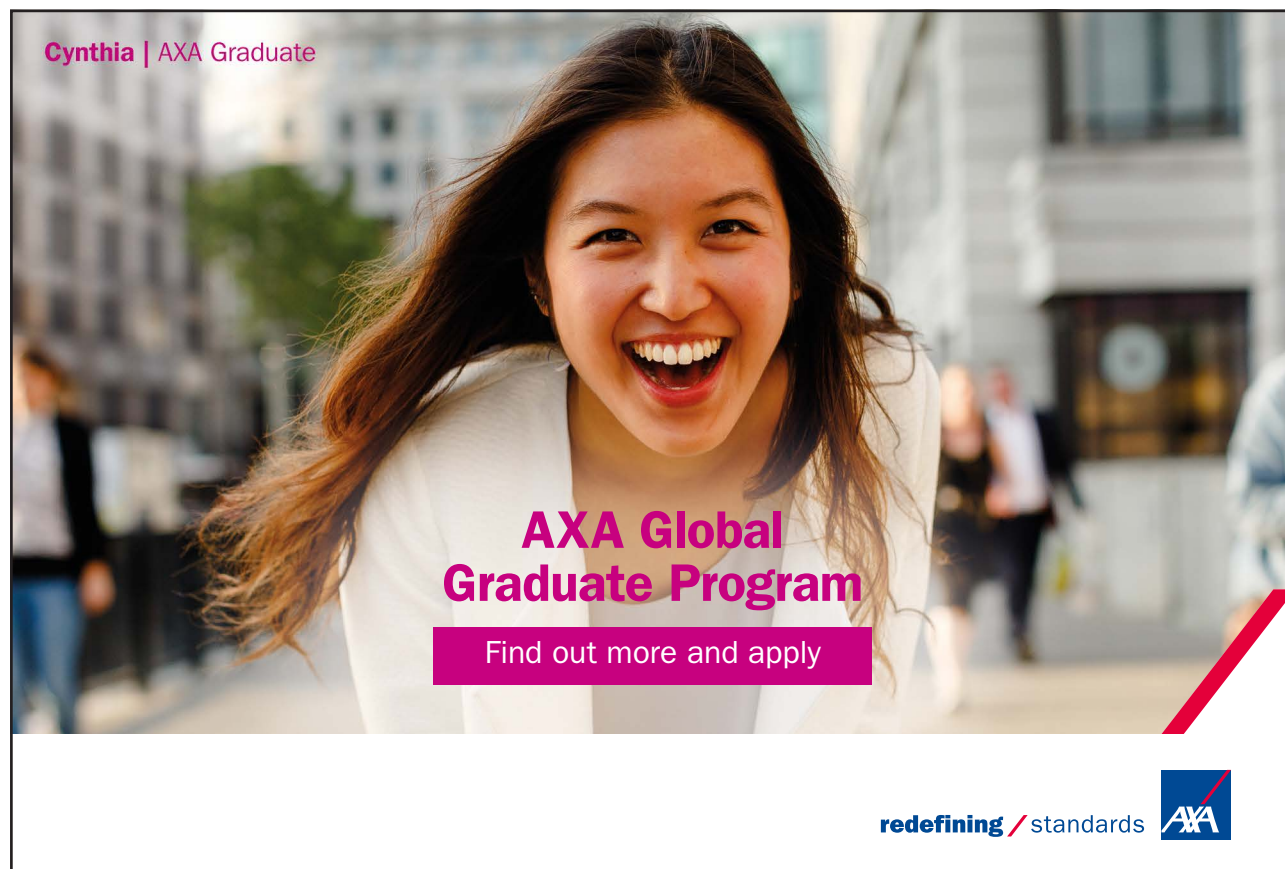
```
10 class Merger implements CSProcess {
11
12     def ChannelInputList fromWorkers
13     def sourceList
14     def runs
15     def N
16     def minSeqLen
17     def outRoot
18     def timeRoot
19     def runId
20
21
22     void run(){
23
24         def allSentinels = { b ->
25             def finished = false
26             def ss = 0
27             def len = b.size()
28             while ( !finished && (b[ss] instanceof Sentinel ) ) {
29                 ss = ss + 1
30                 if ( ss == len) finished = true
31             }
32             return finished
33         } // end allSentinels
34
35     def minKey = { buffers ->
```

```
36     def buffIds = []
37     def minKey = Integer.MAX_VALUE
38     buffers.each { b ->
39         if ((b instanceof PartConcordance) && (b.seqVal < minKey))
40             minKey = b.seqVal
41     }
42     def buffId = 0
43     buffers.each { b ->
44         if ( (b instanceof PartConcordance) && (b.seqVal == minKey) )
45             buffIds << buffId
46             buffId = buffId + 1
47     }
48     return buffIds
49 } // end minKey
50
51 def timer = new CTimer()
52 def concordanceEntry = " "
53 def nodes = fromWorkers.size()
54 def timeFileName = timeRoot + runId + "_M_" + N + "_times.txt"
55 def timeHandle = new File(timeFileName)
56 if (timeHandle.exists()) timeHandle.delete()
57 def timeWriter = timeHandle.newPrintWriter()
58 for ( s in sourceList){
59     for ( r in 1..runs){
60         def startTime = timer.read()
61         def fileName = outRoot + s + "_L_" + minSeqLen + "_N_" + n + "_Dist.txt"
62         def fileHandle = new File (fileName)
63         if (fileHandle.exists()) fileHandle.delete()
64         def fileWriter = fileHandle.newPrintWriter()
65         def inputBuffers = []
66         for ( n in 0..< nodes){
67             inputBuffers << fromWorkers[n].read()
68         }
69         while ( ! allSentinels(inputBuffers)){
70             def minBuffers = minKey(inputBuffers)
71             def wordMap = [:]
72             minBuffers.each{ minId ->
73                 def em = inputBuffers[minId].entryMap
74                 em.each {
75                     def key = it.key
76                     def value = it.value
77                     def currentEntry = wordMap.get(key, [])
78                     wordMap.put(key, (currentEntry + value).sort())
79                 }
80             } // end of each minBuffers
81             wordMap.each { concordanceEntry = " "
82                 def keyWords = it.key
83                 def indexes = it.value
```



```
83     def flatIndex = indexes.flatten()
84     if (flatIndex.size() >= minSeqLen) {
85         concordanceEntry = concordanceEntry + keyWords + ", "
86         concordanceEntry = concordanceEntry + flatIndex.size() + ", "
87         concordanceEntry = concordanceEntry + flatIndex.sort()
88         fileWriter.println "$concordanceEntry"
89     }
90 } // end each wordMap entry
91 minBuffers.each { minId ->
92     inputBuffers[minId] = fromWorkers[minId].read()
93 }
94 } // end while not all sentinels
95 fileWriter.flush()
96 fileWriter.close()
97 def endTime = timer.read()
98 println "MERGER, $s, $N, ${endTime - startTime} "
99 timeWriter.println "MERGER, $s, $N, ${endTime - startTime} "
100 } // end of for r
101 } // end of for s
102 timeWriter.flush()
103 timeWriter.close()
104 } // end run()
105 }
```

Listing 24-7 The Merger Process



Cynthia | AXA Graduate

**AXA Global
Graduate Program**

Find out more and apply

redefining / standards AXA

Each Merger corresponds to a single value for `N` {15} and expects to receive inputs from each of the Worker processes using the `ChannelInputList` `fromWorkers` {12}. Two methods are defined that will be used subsequently. The method `allSentinels` {24–33} returns `true` if the inputs from all of the workers are the terminating `Sentinel` and `false` otherwise. The method `minKey` {35–47} determines the minimum key value that is currently available in the input buffers of the process. It returns the subscript of the one or more buffers that have the current minimum key value.

After some preliminaries to define a `PrintWriter` for the file that will hold the timing data and the number of nodes {50–56}, the process iterates through each of the `sources` and `runs` {57, 58} receiving `PartConcordance` objects from the Sorter processes in each of the Nodes. At the start of each loop the process creates a `PrintWriter` for the file that will hold the concordance for the value `N` {60–63}. Initially an object is read from each of the Nodes and placed in `inputBuffers` {64–67}.

The main loop of the process is a while loop {68–94} that terminates when `allSentinels()` returns `true` {68}. The process then calls `minKey()` to determine which buffers are to be processed {69}. The aim of the next phase is to create a single `wordMap` {70} that contains entries from each of the nodes for each word string corresponding to the sequence value that is the key of each of the active buffers. Each of the active `inputBuffers` is processed in turn to obtain its `entryMap` {72}. The `entryMap` is then processed to obtain each word string key and to add its value to `wordMap` {73–78}. The value of an entry is a list of subscripts where that string is found in the input file. These are sorted to ensure that the subscripts appear in numerical order. Once all the `inputBuffers` have been processed the `wordMap` contains the concordance details that can be transformed into a string and written to file {80–90}. The last part of the main loop is to read in new values to each of the buffers that have just been processed {91–93}. Once all the inputs have been processed times can be written to the time output file {97–99}.

24.8 Performance Analysis of the Distributed System

The system was evaluated using three different source files comprising the text of bible, 2bibles made by concatenating two copies of bible and 4bibles made by concatenating four copies of the bible. The corresponding output file sizes are shown Table 24-3, which are all in Kbytes.

MinSeqLen	2	3	5
N	bible	2bibles	4bibles
1	6,359	13,102	27,069
2	6,359	11,917	23,353
3	5,291	8,731	15,831
4	3,678	5,521	9,341
5	2,557	3,609	5,792
6	1,915	2,587	3,981
Total	26,159	45,467	85,367
ratio to bible		1.74	3.26
ratio to 2bibles			1.88

Table 24-3 Comparative Output File Sizes for Different Input Sources

The minimum sequence length for each of the source files was adjusted so that the difference between the sizes of the output files generated was not too large. This length is the minimum number of occurrences of the word string in the file. If this had not been done the output size for 4bibles would have been 263,321 Kbytes. As can be seen the ratio of the sizes is in proportion to the size of the input files which are in the ratio 1:2:4.

The times in milliseconds for each of the processes on a system comprising 1 Reader process, 6 Merger processes (for N = 6) and 4, 8 and 12 Nodes is given in Table 24-4. The times were averaged over 8 runs.

The Table is subdivided horizontally into three parts corresponding to the number of Node processes (4, 8 and 12). The table is subdivided vertically into two parts the first showing the times for each processor for the source,; bible, 2bibles and 4bibles and the ratio relative to the number of bibles in the source text. In order to show improvement due to the number of processes the ratio should be less than 2 in the column 1 to 2 and less than 4 in column 1 to 4. Using this as a basis for comparison it can be seen that the version using 4 Nodes does not perform as well as the other versions.

Process Times		bible	2bibles	4bibles		Ratio	
	Nodes	4	4	4		1 to 2	1 to 4
Reader		9,750	21,123	44,585		2.17	4.57
Node		103,580	156,686	271,222		1.51	2.62
Merge		103,669	156,722	271,265		1.51	2.62
	Nodes	8	8	8			
Reader		6,349	12,432	23,758		1.96	3.74
Node		128,355	171,378	229,194		1.34	1.79
Merge		127,903	171,060	228,582		1.34	1.79
	Nodes	12	12	12			
Reader		5,662	8,970	19,110		1.58	3.38
Node		145,906	186,436	286,397		1.28	1.96
Merge		145,450	186,400	285,910		1.28	1.97

Table 24-4 Process Times (milliseconds) for various Combinations of Sources and Nodes Processes

TURN TO THE EXPERTS FOR SUBSCRIPTION CONSULTANCY

Subscribe is one of the leading companies in Europe when it comes to innovation and business development within subscription businesses.

We innovate new subscription business models or improve existing ones. We do business reviews of existing subscription businesses and we develop acquisition and retention strategies.

Learn more at [linkedin.com/company/subscribe](https://www.linkedin.com/company/subscribe) or contact Managing Director Morten Suhr Hansen at mha@subscribe.dk

SUBSCRIB✓**BE** - to the future



Click on the ad to read more

In terms of the actual times it can be seen that bible and 2bibles runs quickest on 4 Nodes and that 4bibles run fastest on 8 Nodes. The use of 12 Nodes cannot be justified for these data sets for any of the sources. In all cases the process that shows most improvement is the Reader process. It has to be recalled that the single machine parallel version of the algorithm completed the whole task for bible in 3,961 milliseconds and thus none of these solutions is better than this.

The problem with the application is that it is I/O bound. The main effect is that as we add more Node processes the number of data records that are transmitted over the interconnect increases and this has a detrimental effect on overall processing time. This can be seen more clearly in Table 24-5, which shows the speed up achieved using the various numbers of Node processes and sources.

Speedup	Nodes	bible	2bibles	4bibles		Ideal
Reader	4 to 8	1.54	1.70	1.88		2
	4 to 12	1.72	2.35	2.33		3
Node	4 to 8	0.81	0.91	1.18		2
	4 to 12	0.71	0.84	0.95		3
Merge	4 to 8	0.81	0.92	1.19		2
	4 to 12	0.71	0.84	0.95		3

Table 24-5 Speedup: Based on Times in Table 24-4

The Ideal column shows the value if optimum speedup were to be achieved. The reader process consistently shows the closest approach to the optimum, ideal, value. The other processes often show speedups that are less than 1.0 which means that the overall processing time gets longer with more processors.

24.8.1 Analysis of Overall Processing Times

Once the algorithm has been distributed over a network, rather than running in a single multi-core processor, it is vital to consider the amount of data communication that result from the parallelisation on a distributed system. In the Concordance example data is communicated from the Reader process to each of the Processing Nodes and then from each of these Processing Nodes to each of the Merger Nodes. This data is transferred as a `Serializable` object and as such incurs a cost in terms of the amount of storage required to hold such a serialized object. A method was created `defs.sizeof()` which attempts to estimate the size of each object that is transferred across the network. This can then be used to determine the total amount of data sent between each of the nodes. Table 24-6 shows the number of bytes that were communicated for each text for the various numbers of nodes.

Nodes	Bible	2Bibles	4Bibles
4	343,262,248	580,965,093	1,046,447,697
8	366,888,432	611,272,041	1,090,260,161
12	385,214,256	711,856,998	1,342,869,549

Table 24-6 Number of Bytes Generated by Object Serialization

Perhaps of more interest is the time, in seconds, it takes to transfer this number of bytes over the 100mbit/sec Ethernet connection that was used to connect the processing nodes, as shown in Table 24-7.

Nodes	Bible	2Bibles	4Bibles
4	26.2	44.3	79.8
8	28.0	46.6	83.2
12	29.4	54.3	102.5

Table 24-7 Time to Transfer Serialized Objects in Seconds

We can thus observe that a substantial proportion of the overall processing time is taken up with the transfer of data between the nodes. The times shown in Table 24-7 do not include the processing time required to both serialize and de-serialize the objects once they have been transferred. A further contribution to the overall time is the time it takes to actually write the out files to disc and even though these are on separate Merger processes this still takes time.

24.9 Summary

This chapter has explored the manner in which large data sets can be processed. Initially it demonstrated how multiple cores could be utilised once the design had been finalised ensuring that some parameter was chosen upon which the algorithm can be scaled. This was followed by a discussion of the techniques that can be used to scale the algorithm to a network of workstations.

Necessarily, such designs are very dependent upon the specifics of the data being processed. There is no specific proforma that can be followed to achieve such a parallelisation. It has to be undertaken on a case by case basis. The crucial aspect of such a design is that data structures have to be found which can be separated into a writing and then a reading phase. At all costs it is advisable to avoid creating data structures that mix writing and reading in some random order as this is very difficult to parallelise.

In many situations it is better to return to first principles for an algorithm rather than to try to parallelise a highly tuned sequential algorithm. Over the years much effort has been expended many people to improve sequential algorithms to run on single processor machines. Many of these optimisations are inherently hard to run in a parallel mode. Therefore it is often much easier to start afresh if it is desired to produce a scalable parallel solution.

If an algorithm is to be distributed over a number of workstations or nodes within a cluster then a balance has to be achieved between the amount of computation and communication. Various communication infrastructures are available from a basic Ethernet to specially designed networks for High Performance Computer Clusters. The choice of distribution mechanism will depend on the available network capability and thus no hard and fast rule can be provided. In the example described above, distribution actually caused the application to run more slowly but did mean that we had a means of dealing with any size of data set. It is a trade-off that each developer will have to consider for each application that they choose to distribute over a cluster of workstations.

A final optimisation would be to transform the data objects that are transmitted over the network using the `net2 filter` capability. This would then yield a simpler and small data structure that should be quicker to serialise. That is left as an exercise for the interested reader!



Losing track of your leads?

Bookboon leads the way
Get help to increase the lead generation on your own website. Ask the experts.

bookboon.com

Interested in how we can help you?
email ban@bookboon.com 



25 Concluding Remarks

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 (Formal Systems Europe Ltd, 2013) (Holzmann G.J., 2013).

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 (Welch & Barnes, 2013). All the examples presented are demonstrable and where necessary operate over a network and employ aspects of mobility. Yet this is achieved in a 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 that 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 system designer 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.

25.1 The Initial Challenge – A Review

In Chapter 1 the ideas of parallel system design were introduced by means of a simple game. The description of the game stopped at the point where we needed to introduce some basic concepts. We now return to the game with the knowledge gained as a result of assimilating the presented material. The key to designing parallel systems has been the use of the client-server design pattern. We therefore return to the original process design and present the interactions undertaken between the processes to see that they obey the client-server design pattern. Figure 24-1 shows the process architecture with the channels named and the client-server interactions labelled. The Mouse Buffer is a pure server, which makes it much easier to undertake the deadlock analysis.

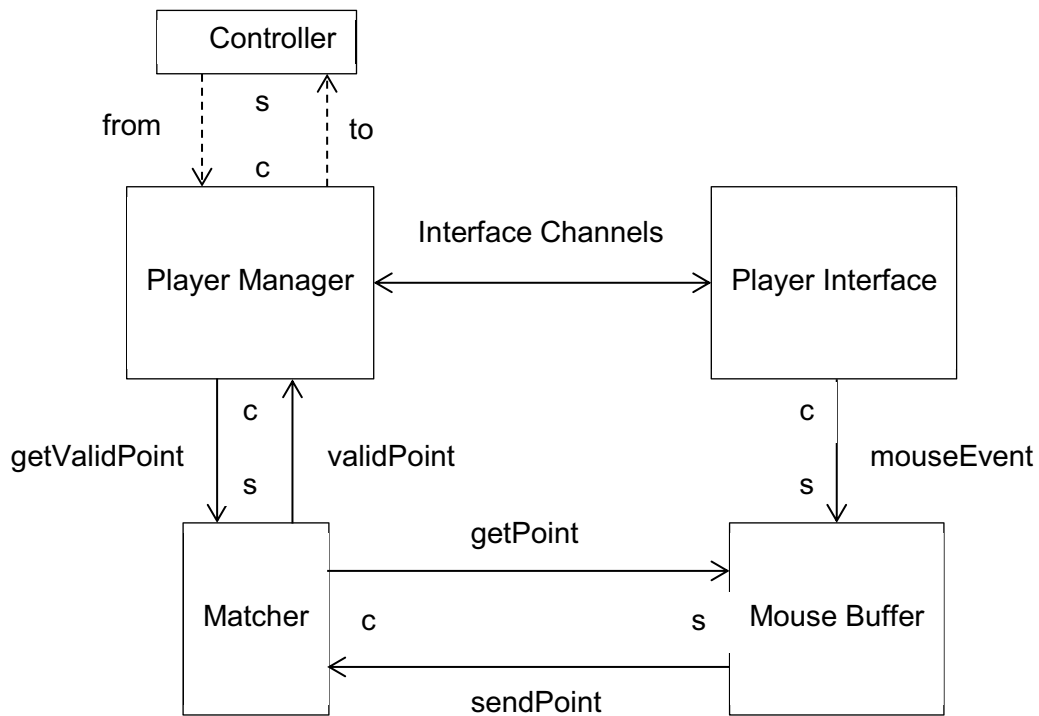


Figure 25-1 The Player Process Network

"I studied English for 16 years but...
...I finally learned to speak it in just six lessons"
Jane, Chinese architect

ENGLISH OUT THERE

Click to hear me talking before and after my unique course download



There is still a problem with the interaction between the Player Manager and the Player Interface, which is bi-directional. It is thus necessary to understand these interactions to ensure that deadlock cannot occur.

Figure 24-2 shows the actual messages and how they are sent and received by the processes. The crucial interactions are those within the Player Manager. In all cases, the Player Manager acts as the client apart from the ‘withdraw from game’ button press. In order to deal with this, a non-deterministic choice is required in the main loop of the process to ensure that this input can be processed.

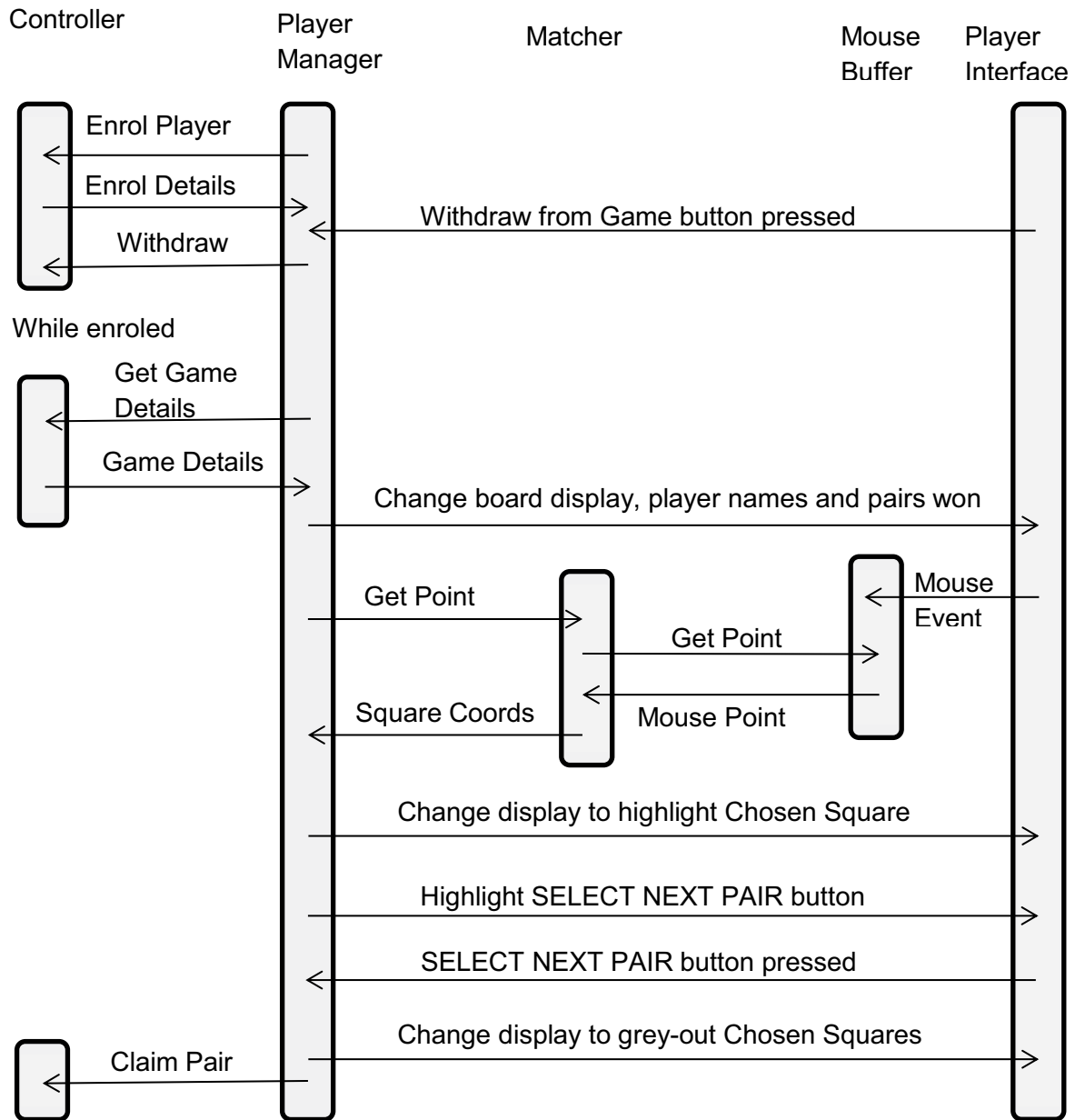


Figure 25-2 Communication Interactions

The implementation has some limitations, in particular, that the players do not have to take turns and they cannot see the cards other players have turned over. This implementation is provided as a working system in the accompanying software.

The challenge is to modify the game implementation so that players have to take it in turn to reveal squares and all players can see the squares that have been revealed. If a player reveals a pair they can continue with their turn by revealing more squares. A player's turn ceases as soon as they reveal two squares that do not match.

There are many ways of solving this challenge and it is left to the interested reader to solve the challenge in one or more ways!

25.2 Final Thoughts

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 it 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!

26 References

Anon., 2009. *Writing Portable Groovy Scripts with A Magic Runnable Jar*. [Online] Available at: <http://www.weheartcode.com/2009/01/14/writing-portable-groovy-scripts-with-a-magic-runnable-jar/> [Accessed 30 May 2013].

Barclay, K. & Savage, J., 2006. *Groovy Programming: An Introduction for Java Developers*. San Francisco: Morgan Kaufmann.

Belapurkar, A., 2013. *IBM DeveloperWorks Technical Library*. [Online] Available at: <http://www.ibm.com/developerworks/java/library/j-csp2/> [Accessed 22 3 2013].

Brinch Hansen, P., 1973. *Operating System Principles*. s.l.: Prentice Hall.

Chalmers , K., 2008. Introducing JCSP Networking 2.0. In: *Communicating Process Architecture 2008*. s.l.:IOS Press Amsterdam.



This e-book
is made with
SetaPDF

SETASIGN

PDF components for PHP developers

www.setasign.com

Chalmers, K., 2009. *Investigating communicating sequential processes for Java to support ubiquitous computing (PhD thesis)*, Edinburgh: Edinburgh Napier University.

Chalmers, K. & Kerridge, J., 2005. jcsp.mobile: A Package Enabling Mobile Processes and Channels. In: *Communicating Process Architectures 2005*. s.l.:IOS Press, Amsterdam.

Chalmers, K., Kerridge, J. & Romdhani, I., 2007. Mobility in JCSP: New Mobile Channel and Mobile Process Models. In: *Communicating Process Architectures 2007*. s.l.:IOS Press Amsterdam.

Formal Systems Europe Ltd, 2013. *Formal Systems*. [Online]
Available at: <http://www.fsel.com/software.html>
[Accessed 22 3 2013].

Hoare, C., 1978. Communicating Sequential Processes. *Communications of the ACM*, 17(10).

Hoare, C., 1985. *Communicating Sequential Processes*. available from <http://www.usingcsp.com/> ed. s.l.:Prentice Hall.

Holzmann G. J., 2013. *ON-THE-FLY, LTL MODEL CHECKING with SPIN*. [Online]
Available at: <http://spinroot.com/spin/whatispin.html>
[Accessed 22 3 2013].

Inmos Ltd, 1988. *occam programming refence manual*. s.l.:Prentice Hall.

JUnit, 2013. *JUnit*. [Online]
Available at: <http://junit.org/>
[Accessed 22 3 2013].

Kerridge, J., 2007. Testing and Sampling Parallel Systems. In: *Communicating Process Architectures 2007*. s.l.:IOS Press Amsterdam.

Kerridge, J., Barclay, K. & Savage, J., 2005. Groovy Parallel! A Return to the Spirit of occam?. In: *Communicating Process Architectures 2005*. s.l.:IOS Press 2005, Amsterdam.

Kerridge, J. & Chalmers, K., 2006. Ubiquitous Access to Site Specific Services by Mobile Devices: the Process View.. In: *Communicating Process Architectures 2006*. s.l.:IOS Press Amsterdam.

Kerridge, J., Haschke, J.-O. & Chalmers, K., 2008. Mobile Agents and Processes using Communicating Process Architectures. In: *Communicating Process Architectures 2008*. s.l.:IOS Press Amsterdam.

Kerridge, J., Welch, P. & Wood, D., 1999. Synchronisation Primitives for Highly Parallel Discrete Event Simulations.. In: *In: 32nd Hawaii International Conference on Systems Science – HICSS-32*. s.l.:IEEE Computer Society Press..

Kosek, A., Kerridge, J., Syed, A. & Armitage, A., 2009. JCSP Agents-Based Service Discovery for Pervasive Computing. In: *Communicating Process Architectures 2009*. s.l.:IOS Press Amsterdam.

Lea, D., 2003. *Concurrent Programming in Java: Design Principles and Pattern (2nd Edition)*. s.l.:Addison-Wesley.

Magedanz, T., Rothermel, K. & Krause, S., n.d. Intelligent agents: An emerging technology for next generation telecommunications?. In: *9. INFOCOM'96. Fifteenth Annual Joint Conference of the IEEE Computer Societies. Networking the Next Generation*. s.l.:IEEE, pp. 464–472.

Martin, J. & Welch, P., 1997. A Design Strategy for Deadlock-Free Concurrent Systems. *Transputer Communications*, 3(4), pp. 215–232.

Nwana, H., 1996. Software Agents: An Overview. *Knowledge Engineering Review*, 11(3), pp. 205–244.

Pham, V. & Karmouch, A., 1998. Mobile Software Agents: An Overview. *IEEE Communication Magazine*, Issue July, pp. 26–37.

Project Gutenberg, 2014. *Free ebooks – Project Gutenberg*. [Online]

Available at: <http://www.gutenberg.org/>

[Accessed 9 June 2014].

Ritson, C. & Welch, P., 2007. A Process-Oriented Architecture for Complex System Modelling. In: *Communicating Process Architectures 2007*. s.l.:IOS Press, Amsterdam, pp. 249–266.

Welch, P., 2002. Concurrency For All. In: *Lecture Notes in Computer Science – 2330*. s.l.:Springer-Verlag, pp. 678–687.

Welch, P., 2013. *Communicating Sequential Processes for Java (JCSP)*. [Online]

Available at: <http://www.cs.kent.ac.uk/projects/ofa/jcsp/>

[Accessed 22 3 2013].

Welch, P. & Barnes, F., 2013. *occam-pi: blending the best of CSP and the pi-calculus*. [Online]

Available at: <http://www.cs.kent.ac.uk/projects/ofa/kroc/>

[Accessed 22 3 2013].

Welch, P. et al., 2007. Integrating and Extending JCSP. In: *Communicating Process Architectures 2007*. s.l.:IOS Press Amsterdam, pp. 349–369.

Welch, P., Brown, N., Moores, J. & et al, 2010. Alting Barriers: Synchronisation with Choice in Java using JCSP, 22. pp. 182–196. *Concurrency and Computation: Practice and Experience*, Volume 22, pp. 182–196.

Welch, P., Justo, G. & Willcock, C., 1993. High-level paradigms for deadlock-free high-performance systems. In: *Proceedings of the 1993 World Transputer Congress on Transputer Applications and Systems*. s.l.:IOS Press, Amsterdam.

Wikipedia, 2013. *Transputer*. [Online]
Available at: <http://en.wikipedia.org/wiki/Transputer>
[Accessed May 2013].

gaiteye[®]
Challenge the way we run

**EXPERIENCE THE POWER OF
FULL ENGAGEMENT...**

**RUN FASTER.
RUN LONGER..
RUN EASIER...**

**READ MORE & PRE-ORDER TODAY
WWW.GAITEYE.COM**

Index

A

Access Server, 86
AccessProcess, 88
Active Network, 95
Adaptive Agent, 123
ALT, 20, 40, 66, 90, 97, 98, 100, 107, 130, 137, 138
alternative, 20, 41, 44, 65, 97, 107, 108, 115, 132, 137,
151, 218
any2net, 16, 17, 19, 20, 27, 36, 45, 77, 78, 82, 85, 90, 92,
111, 113, 117, 126, 128, 129, 143, 161, 162, 170, 174,
176, 177, 178
any2one, 72, 78, 120, 140, 178
Automatic Class Loading, 147

B

BackAgent, 63
BackRoot, 64
Big Data, 191

C

Canteen, 24, 25, 26, 28
channel, 218
ChannelInputList, 31, 33, 173, 177, 179, 181, 182, 183,
184, 185, 212, 213
ChannelOutputList, 31, 33, 137, 138, 171, 172, 173, 177,
179, 181, 182, 183, 184, 185, 205, 207, 208
Chef, 26
Collector, 154, 175
concordance, 191
CREW, 15, 28, 29, 31, 33
CSTimer, 18, 22, 39, 50, 137, 138, 154, 172, 176, 194,
207, 208, 210, 211, 213

D

data parallel architecture, 148
DataGenerator, 120
DataGenerator process, 136
deadlock, 63, 65, 104, 105, 112, 113, 114, 115, 118, 219,
223, 224

Dining Philosophers, 15, 24, 25
DoWork, 165

E

Emitter, 157, 175
EmitterNet, 158

F

Forward and Back Agent, 69

G

Gatherer, 121
Gatherer process, 140
GConsole, 29, 31, 97, 101
Get, 22
GetInput, 163
GroovyLauncher, 189
GroovyTestCase, 46, 48, 49, 53
Group Location Service, 90
guard, 97, 132, 137

H

High Performance Cluster, 167
HPC. See High Performance Clusters

K

Kitchen, 24, 26

M

McPiCollector, 185
McPiCore, 180
McPiEmitter, 183
McPiManager, 181
McPiWorker, 182
Merger, 204, 212
mobile agent, 54, 79, 95, 120
MobileAgent, 54, 55, 56, 64, 73, 103, 104, 123
Montecarlo Pi, 179
Multiple Instruction Multiple Data (MIMD), 147

N

net channel, 15, 16, 23, 28, 31, 35, 36, 41, 45, 63, 64, 65,
69, 70, 72, 73, 74, 75, 76, 78, 82, 85, 93, 95, 102, 124,
127, 128, 137, 153, 154, 168, 170, 171, 177, 178, 205
net channels, 15, 16, 17, 35, 70, 105, 145, 182

net2any, 16, 22, 23, 27, 28

net2one, 16, 17, 20, 25, 36, 38, 39, 40, 44, 48, 53, 61, 62,
67, 69, 75, 78, 92, 117, 128, 129, 144, 145, 154, 157,
161, 162, 177, 178

NetChannelLocation, 37, 41, 64, 89, 90, 92, 103, 107, 111

node, 15, 16, 17, 18, 19, 20, 22, 23, 25, 26, 28, 29, 30, 31,
33, 36, 39, 44, 45, 48, 53, 54, 55, 57, 58, 60, 61, 62,
63, 64, 65, 66, 68, 69, 70, 72, 73, 74, 75, 76, 77, 79,
80, 82, 84, 85, 88, 91, 94, 95, 96, 101, 102, 103, 104,
105, 107, 108, 110, 111, 112, 113, 114, 115, 116, 117,
118, 119, 120, 121, 123, 124, 125, 126, 127, 128, 131,
132, 134, 135, 136, 137, 140, 142, 143, 144, 145, 146,
148, 152, 153, 154, 156, 157, 158, 159, 160, 161, 162,
164, 166, 167, 168, 169, 170, 171, 173, 174, 175, 177,
178, 179, 182, 183, 185, 186, 189, 190, 204, 205, 206,
208, 209, 210, 211, 212

NodeProcess, 128

numberedNet2One, 19, 28, 29, 31, 32, 85, 86, 90, 172,
174, 176, 177

O

one2any, 208

one2net, 16, 23, 25, 26, 28, 31, 32, 33, 38, 39, 42, 43, 48,
53, 61, 62, 64, 67, 69, 70, 73, 74, 76, 87, 90, 91, 138,
158, 160, 170, 172

P

PAR, 19, 20, 22, 23, 25, 26, 27, 29, 31, 33, 44, 45, 46, 47,
48, 49, 52, 53, 61, 63, 67, 78, 83, 84, 89, 117, 143, 144,
145, 154, 157, 162, 163, 183, 195, 208, 210

Philosopher, 24, 26, 27, 28

Player Interface, 219

Player Manager, 219

PrintSpooler, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45

PrintUser, 36, 37, 38, 39, 43, 44, 45, 81

priSelect, 98, 100

process, 218

Process discovery, 119

Process Farms, 147

process mobility, 80

process network under test (PNUT), 46

ProcessManager, 54, 57, 59, 66, 70, 75, 76, 85, 86, 110,
113, 122, 129, 130, 131, 132, 133, 134, 135, 136, 174,
175, 177, 179

ProcessNode, 58

Prompter, 98

pure clients, 150, 157

pure server, 150, 219

Put, 21

Q

Queue, 96, 99

R

Reader, 205

Receiver, 17

Redirecting Channels, 95

RestartAgent, 104

RestartAgent processing, 112

Ring Element, 101, 103, 107

RingAgentElement, 105

Root, 56

S

Scalable Architecture, 205

Self-Monitoring Process Ring, 95

Sender, 17

SendOutput, 165

Service Architecture, 82

sharedData, 161

Single Instruction Multiple Data (SIMD), 147

Sorter, 211

Sorter - Output, 204

StateManager, 100

StateManager processing, 115

StopAgent, 101

StopAgent processing, 110

T

task parallel architecture, 152

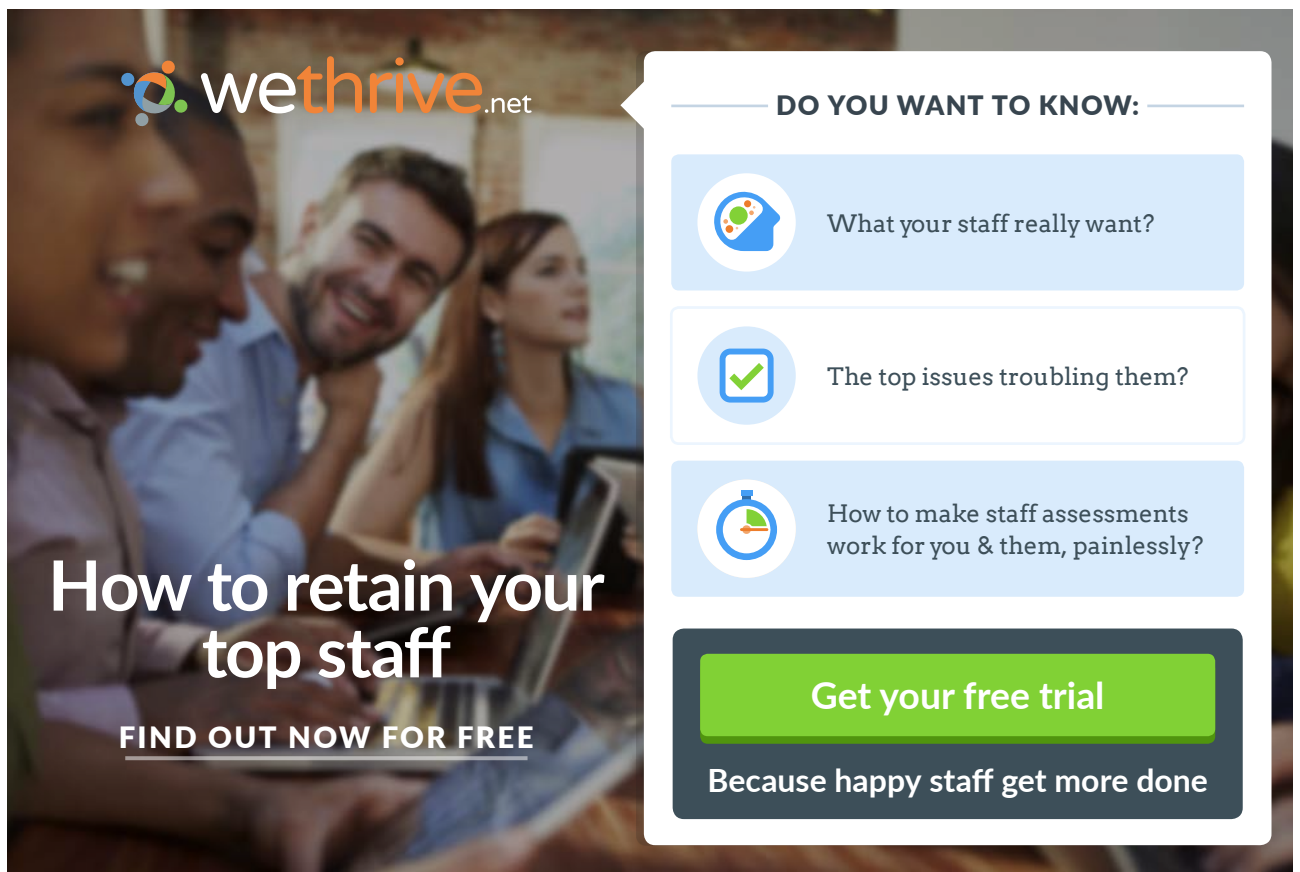
TCP/IP, 15, 16, 19, 28, 34, 46, 47, 60, 88

Test-Network, 46, 47, 48, 49

timer, 18, 22, 39, 42, 50, 137, 138, 154, 155, 171, 172,
173, 174, 175, 176, 177, 179, 193, 194, 195, 205, 207,
208, 210, 211, 213, 218
Travellers' Meeting System, 81
TripAgent, 72
TripNode, 74
TripRoot, 75

U
Universal Client, 83

W
Worker, 208
Worker Object interface, 167
Worker process, 149, 161
WorkerObject, 173



wethrive.net

How to retain your top staff

FIND OUT NOW FOR FREE

DO YOU WANT TO KNOW:

- What your staff really want?
- The top issues troubling them?
- How to make staff assessments work for you & them, painlessly?

Get your free trial

Because happy staff get more done

