15 Communication over Networks: Process Parallelism

JCSP, together with JCSPNet provide a transparent means of connecting processes, whether they be 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 7 and 10, 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.

15.1 Channel Name Service

The Channel Name Service (`CNS`) is a process that runs on one node of the network. It runs as a single server process in its own JVM. The `CNS` is specific to the network communications technology. It provides a means of connecting named networked channels between nodes executing on the underlying communications network. The basic JCSPNet package is supplied with a `CNS` for TCP/IP based networks and it is this version that is described here. Other communications infrastructures, such as Bluetooth require the implementation of a `CNS` for the specific technology.

Previously, we have used `Net2One` and `One2Net` networked channels to create `NetChannelInputs` and `NetChannelOutputs` respectively. It is however not possible to create some of the shared '`Any`' versions of channels, the specific variants being determined by the underlying Java `Socket` capability. Implicitly, a `Socket` provides an `Any` input to which many processes can connect, provided they know the address of the `Socket`. Thus the simplest architecture is a networked any-to-one connection allowing many writer processes to output to a single reader process. A networked one-to-any connection can be constructed but the multiple reader processes have to be placed on the same node of the network. It is not possible to construct a networked any-to-any architecture.

15.2 Multiple Writers to One Reader

Figure 15-1 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. The `CNS` is assumed to be running on a node of the network. 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.

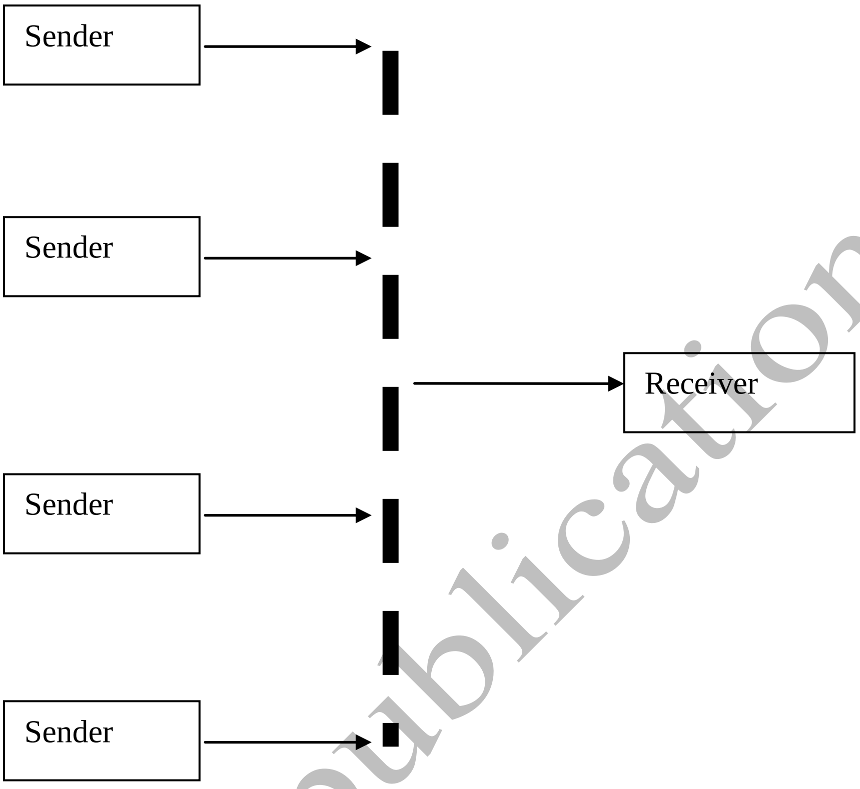

Figure 15-1 The Multi-Sender Single Receiver System

The `Receiver` process simply reads in a currently value `v` {5} from any of the `Sender` processes and prints it on the console window {6} as shown in Listing 15-1.

```
01    class Receiver implements CSProcess {

02      def ChannelInput inChannel

03      void run() {

04        while (true) {
05          def v = inChannel.read()
06          println "${v}"
07        }
08      }
09    }
```

Listing 15-1 The Receiver Process

A `Sender` process simply waits for 10 seconds {16}, outputs an identity `String id` {17}, and then repeats the sequence forever, as shown in Listing 15-1.

```
10     class Sender implements CSProcess {

11       def ChannelOutput outChannel
12       def String id

13       void run() {
14         def timer = new CSTimer()
15         while (true) {
16           timer.sleep(10000)
17           outChannel.write ( id )
18         }
19       }
20     }
```

Listing 15-2 The Sender Process

The system shown in Figure 15-1 is invoked by initially running the CNS, after which the user has a choice because the order of invoking `Receiver` and `Sender` processes is immaterial. It is perhaps easier to create a few `Sender` processes first because all processes write to the same console window and creating another `Sender` is a bit more difficult once the system is running. `Sender` processes can be created and destroyed at any time. The creation of a Sender node is shown in Listing 15-3.

```
21     def v= Ask.string ("Sender identity string ? ")

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

23     def comms = CNS.createAny2Net ("comms")

24     def pList = [ new Sender ( outChannel: comms, id: v ) ]

25     new PAR ( pList ).run()
```

Listing 15-3 A Sender Node

The user is asked to supply the `Sender`'s identity {21}, after which a network `Node` is created {22}. An `Any2Net` channel `comms` is created {23}. The `Sender` process is then constructed {24} and `run` {25}. Provided each `Sender` accesses the same `comms` network channel then they will all be able to write to the `Receiver` process running in its own `Node` as shown in Listing 15-4.

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

27     def comms = CNS.createNet2One ("comms")

28     def pList = [ new Receiver ( inChannel: comms ) ]

29     new PAR ( pList ).run()
```

Listing 15-4 The Receiver Node

At the receiving end the network channel `comms` is defined as `Net2One` {27}. Implicitly, an `Any2Net` 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.

15.3 A Single Writer Connected to Multiple Readers

A single writer process `Puts` data to one of many reader processes that `Get` data from the network. The network structure is shown in Figure 15-2.

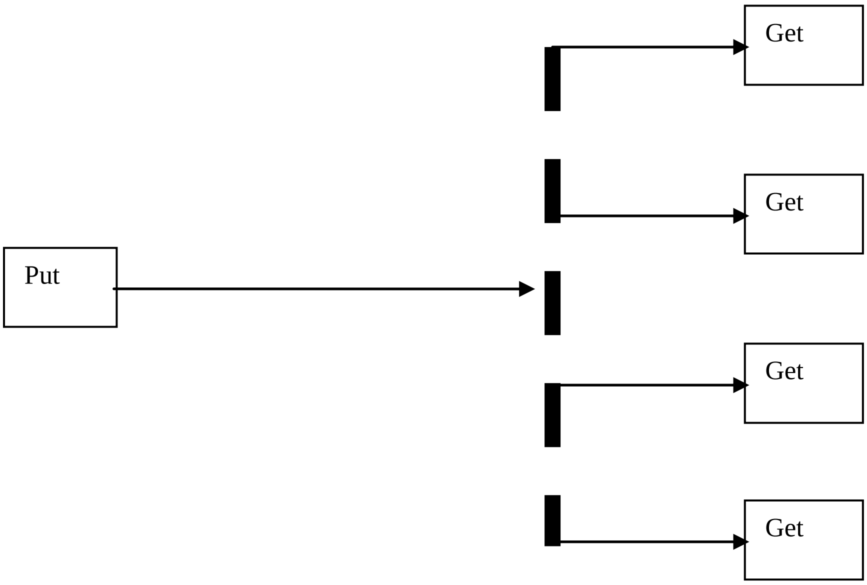

Figure 15-2 One Writer Multiple Reader Process Network

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.

The `Put` process shown in Listing 15-5 shows a process that simply writes a sequence of increasing integers to its `outChannel` {35}. This is done as quickly as possible with no delay between each write operation.

```
30    class Put implements CSProcess {

31      def ChannelOutput outChannel

32      void run() {
33        def i = 1
34        while (true) {
35          outChannel.write ( i )
36          i = i + 1
37        }
38      }
39    }
```

Listing 15-5 The Put Process

The `Get` process, Listing 15-6, has two properties, its `inChannel` {41} and the identity, `id` {42}, of the `Get` process instance.

```
40     class Get implements CSProcess {

41       def ChannelInput inChannel
42       def int id

43       void run() {
44         def timer = new CSTimer()
45         while (true) {
46           def v = inChannel.read()
47           println "${id} .. ${v}"
48           timer.sleep(200 * v)
49         }
50       }
51     }
```

Listing 15-6 The Get Process

A `timer` is defined {44} and within the processes' main loop a value `v` {46} is `read` from the `inChannel` and then printed {47} after which the process `sleeps` for a time proportional to the value read, `v` {48}.

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

53     def comms = CNS.createOne2Net ("comms")

54     def pList = [ new Put ( outChannel: comms ) ]

55     new PAR ( pList ).run()
```

Listing 15-7 The Node Running the Put Process

Listing 15-7 shows how the `Node` that runs the `Put` process is created by defining a single `One2Net` channel `comms` {53} that enables `Put` to `write` values to the network. 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. The `CNS` generates an error indicating that a channel with the name `comms` has already been registered with the CNS when an attempt is made to register it for the second time. Effectively, an attempt to create a second `Socket` with the same name is being undertaken and not surprisingly this causes an error.

Listing 15-8 shows the way in which multiple copies of the `Get` process can be created, within the same node. The channel `comms` {57} is created as a `Net2Any` channel and then passed as a property parameter to a collection of `Get` process {58}.

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

57     def comms = CNS.createNet2Any ("comms")

58     def pList = (0 .. 5).collect{i -> new Get ( inChannel: comms, id: i ) }

59     new PAR ( pList ).run()
```

Listing 15-8 The Creation of Many Get processes Within a Single Node

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.

15.4 Networked Dining Philosophers

As a final demonstration of shared networked channels we implement the canteen based version of the dining philosophers discussed previously in Chapter 14. The process definition for the `Canteen`, `Philosophers` and `Chef` are taken directly from those presented in Chapter 14. All that has changed is the manner of their invocation.

Inspection of Listing 14-9 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 15-3. The network is represented by the central elipse.

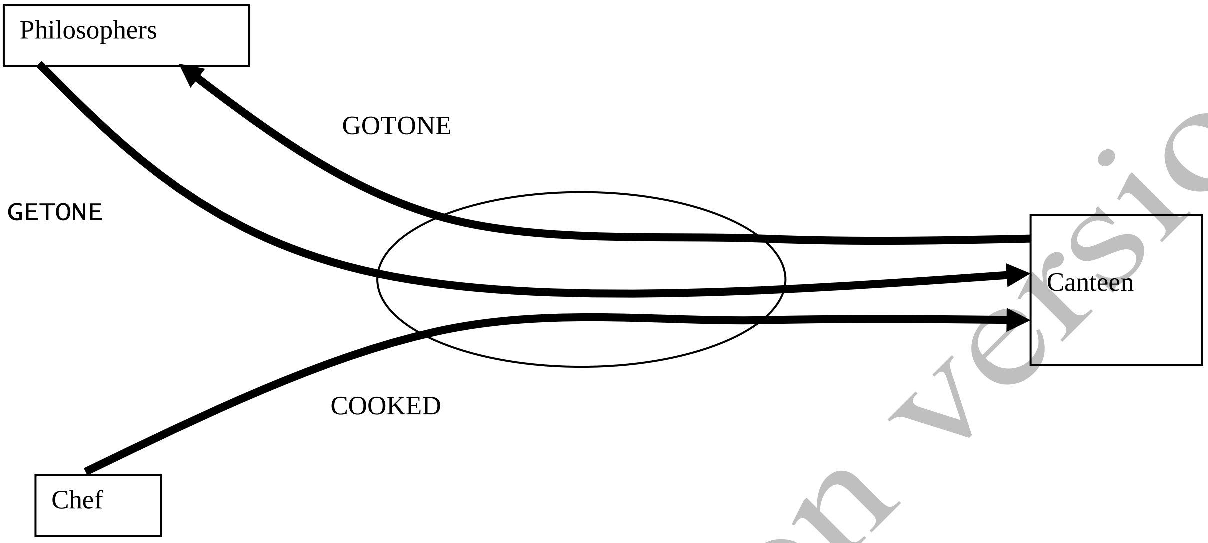

Figure 15-3 The Networked Structure for the Dining Philosophers Canteen Based Solution

The collection of `Philosopher` processes is invoked by means of the script shown in Listing 15-9. The channels `GOTONE` and `GETONE` are defined as net shared channels {61, 62}. The collection of `Philosopher` processes is then created {63-66} and executed {67}.

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

61     def gotOne = CNS.createNet2Any ("GOTONE")
62     def getOne = CNS.createAny2Net ("GETONE")

63     def philList = ( 0 .. 4 ).collect{ i ->
64                         return new Philosopher( philosopherId:i,
65                                                 service:getOne, deliver:gotOne)
66                         }
67     new PAR ( philList ).run()
```

Listing 15-9 The Node Running the Collection of the Philosopher Processes

The `Canteen` is run on a node as shown in Listing 15-10.

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

def cooked = CNS.createNet2One ("COOKED")
def getOne = CNS.createNet2One ("GETONE")
def gotOne = CNS.createOne2Net ("GOTONE")

def processList = [
                  new ClockedQueuingServery( service: getOne,
                                             deliver: gotOne,
                                             supply: cooked)
                  ]
new PAR ( processList ).run()
```

Listing 15-10 The Invocation of the Canteen 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 {69-71}. The `ClockedQueuingServery` version of the canteen is invoked {72-76} and run {77}.

The `Chef` is invoked as shown in Listing 15-11, 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.

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

def cooked = CNS.createOne2Net ("COOKED")

def processList = [ new Kitchen ( supply: cooked) ]

new PAR ( processList ).run()
```

Listing 15-11 The Kitchen Node

15.5 Summary

This chapter has explored the creation of networks of processes connected by shared network channels. The effect the underlying Java Socket based implementation has on the nature of the networks that can be created has also been demonstrated. It is obvious that the creation of unique names for each networked channels will become a problem if a large number of such channels have to be created. Further if we wish to create dynamic systems with networked channels being created as needed then this will be even more of a problem. The creation of anonymous network channels is discussed in the next chapter.

15.6 Exercise

1. Make the CREW system of Chapter 13 run over a network.