Workbook

REQUIREMENT 4
40182161
Edinburgh Napier University - Fundamentals of Parallel Systems (SET09109)

Exercise 2

2.1 Multiplier Process

Code

Listing 1: "Multiplier.groovy"

```
void run()
{
   def i = inChannel.read()
   while (i > 0) {
      // write i ∗ factor to outChannel
      outChannel.write(i∗factor)
      // read in the next value of i
      i = inChannel.read()
   }
   outChannel.write(i)
}
```

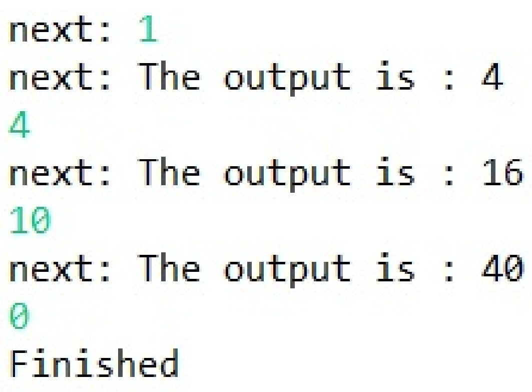
Listing 2: "Consumer.groovy"

```
while ( i > 0 )
{
   //insert a modified println statement
   println "The output is : ${i}"
   i = inChannel.read()
}
```

Listing 3: "RunMultiplier.groovy"

```
def processList = [ new Producer ( outChannel: connect1.out() ),

   //insert here an instance of multiplier with a multiplication factor of 4
   new Multiplier ( inChannel: connect1.in(),
               outChannel: connect2.out(),
               factor: 4),
   new Consumer ( inChannel: connect2.in() )
]
```

Output



```
next: 1
next: The output is : 4
4
next: The output is : 16
10
next: The output is : 40
0
Finished
```

Figure 1: **Exercise 2-1** - Output from Run Multiplier program.

Explanation The *Multiplier.groovy* process (see Listing 1) is inserted into the process list between the producer and consumer. The *Producer.groovy* process outputs an integer, that has been provided, to the Multiplier process which then outputs each integer multiplied by a constant factor which is set in the constructor of the Multiplier instance (see line 6 in Listing 3). The *Consumer.groovy* process prints a meaningful output to the console.

2.2 Integer Sets

Code

Listing 4: "ListToStream.groovy"

```
while (inList[0] != −1)
{
   // hint: output list elements as single integers
   for ( i in 0 ..< inList.size)outChannel.write(inList[i])
      inList = inChannel.read()
}
```

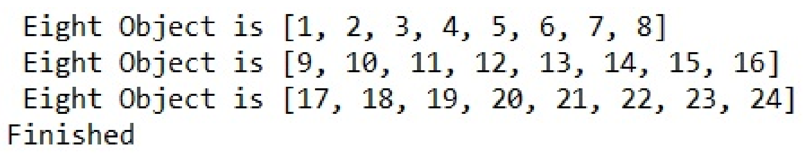
Listing 5: "CreateSetsOfEight.groovy"

```
1   while (v != −1)
2   {
3      for ( i in 0 .. 7 )
4      {
5         // put v into outList and read next input
6         outList[i] = v
7         v = inChannel.read()
8      }
9      println " Eight Object is ${outList}"
10  }
```

Listing 6: "GenerateSetsOfThree.groovy"

```
void run()
{
   def threeList = [
      [1, 2, 3],
      [4, 5, 6],
      [7, 8, 9],
      [10, 11, 12],
      [13, 14, 15],
      [16, 17, 18],
      [19, 20, 21],
      [22, 23, 24],
      [−1, −1, −1]]       // terminating list
   for ( i in 0 ..< threeList.size)outChannel.write(threeList[i])
   //write the terminating List as per exercise definition
}
```

Output



```
 Eight Object is [1, 2, 3, 4, 5, 6, 7, 8]
 Eight Object is [9, 10, 11, 12, 13, 14, 15, 16]
 Eight Object is [17, 18, 19, 20, 21, 22, 23, 24]
Finished
```

Figure 2: **Exercise 2-2** - Output from Run Three to Eight program.

Exercise Questions

What change is required to output objects containing six integers?

Within the *CreateSetsOfEight.groovy* process, change the number of iterations of the for loop. See line 3 of listing 5. The new line should read

Listing 7: "CreateSetsOfEight.groovy - Change required to output objects containing six integers"

```
//for (i in 0 .. 7) {
for (i in 0 .. 5) {
```

which will now create lists containing 6 integers. This can be improved by changing the number to a variable which is set through the constructor of the process.

How could you parameterise this in the system to output objects that contain any number of integers (e.g. 2, 4, 8, 12)?

The process can be improved by creating a parameter for the size of the output list, bearing in mind the iterations start from zero so the required size should be minus one of the input parameter. This can either be a variable that is set within the constructor of the *CreateSetsOfEight.groovy* process or by user input from the console.

What happens if the number of integers required in the output stream is not a factor of the total number of integers in the input stream (e.g. 5 or 7) ?

Some integers from the input stream will not be outputted to the console as they do not make up a full set of numbers. So the set cannot be filled and therefore the set will not display as the process cannot finish.

Exercise 3

3.1 Reversing GIntegrate

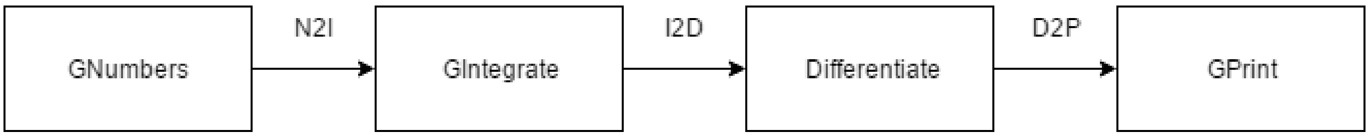


Figure 3: **Exercise 3-1** - Network for process reversing the effect of *GIntergrate.groovy*.

GNumbers.groovy outputs a stream of integers starting at zero and incrementing by one each time. *GIntegrate.groovy* increments the stream by an increasing number each time so the difference between the output is increasing. To negate this effect the output from the network should be equal to the output from the initial GNumbers process.

Minus Process

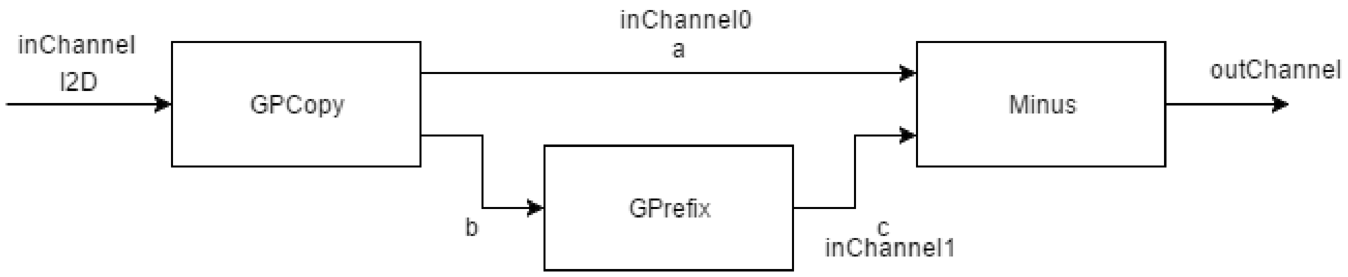


Figure 4: **Exercise 3-1** - Network showing processes for *Differentiate.groovy*.

Explanation The minus process works by copying the output value from *GIntegrate.groovy*, e.g 0, 1, 3, 6. One copy is sent straight to inChannel0 of the *Minus.groovy* process, and the other is sent to *GPrefix.groovy* which outputs the stream with a leading zero into inChannel1 - see Figure 4. The *Minus.groovy* process reads both inputs in parallel and minuses the second from the first resulting in an output stream of incrementing numbers (Figure 5).

Code

Listing 8: "Minus.groovy"

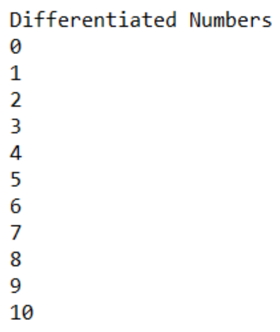
```
ProcessRead read0 = new ProcessRead ( inChannel0)
ProcessRead read1 = new ProcessRead ( inChannel1)
def parRead2 = new PAR ( [ read0, read1 ] )

while (true) {
  parRead2.run()
  // output one value subtracted from the other
  // be certain you know which way round you are doing the subtraction!!
```

```
    outChannel.write(read0.value − read1.value)
}
```

Listing 9: "Differentiate.groovy - see Figure 4 for network of this list"

```
def differentiateList = [ new GPrefix ( prefixValue: 0,
                              inChannel: b.in(),
                              outChannel: c.out() ),
                            new GPCopy ( inChannel: inChannel,
                              outChannel0: a.out(),
                              outChannel1: b.out() ),
                            // insert a constructor for Minus
                            new Minus (inChannel0: a.in(),
                              inChannel1: c.in(),
                              outChannel: outChannel)
                            ]
```

Output



```
Differentiated Numbers
0
1
2
3
4
5
6
7
8
9
10
```

Figure 5: **Exercise 3-1** - Output from the Differentiate System using the Minus Process.

Negator Process

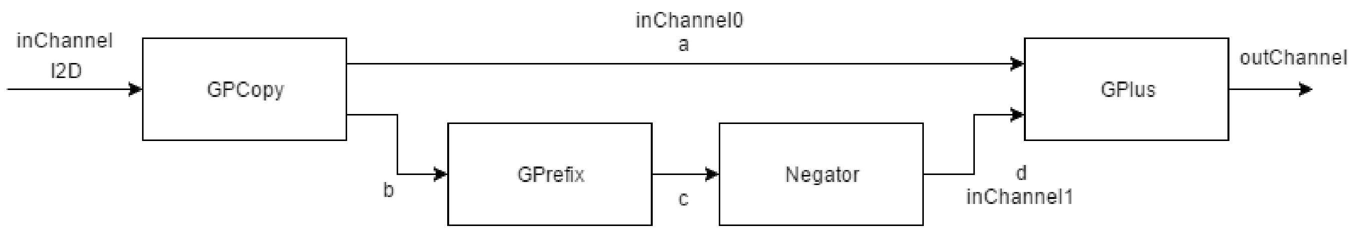


Figure 6: **Exercise 3-1** - Network showing processes for *DifferentiateNeg.groovy* (replacement for Differentiate process within Figure 3)

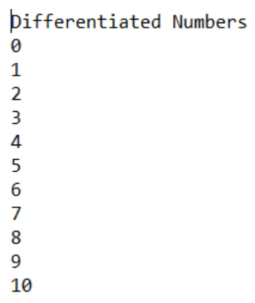
Explanation The negator process works by using the *GPlus.groovy* process which outputs the sum of the two input channels read in parallel. The first input channel is a copy of the original input, whereas the second input channel is a negative version of the input with a leading zero from *GPrefix.groovy*.

Code

Listing 10: "DifferentiateNeg.groovy"

```
def differentiateList = [ new GPrefix ( prefixValue: 0,
                              inChannel: b.in(),
                              outChannel: c.out() ),
                          new GPCopy ( inChannel: inChannel,
                              outChannel0: a.out(),
                              outChannel1: b.out() ),
                          //insert a constructor for Negator
                          new Negator ( inChannel: c.in(), outChannel: d.out()),
                          new GPlus  ( inChannel0: a.in(),
                            inChannel1: d.in(),
                            outChannel: outChannel )
                          ]
```

Output



```
Differentiated Numbers
0
1
2
3
4
5
6
7
8
9
10
```

Figure 7: **Exercise 3-1** - Output from the Differentiate System using the Negator process.

Exercise Questions

Which is the more pleasing solution and why? In this case, even though the Negator solution requires an extra process, it can be argued that this is more pleasing as there is less room for error introduced by the order of the values within the Minus solution. However, a more appropriate solution to negating the effects of GIntegrate would be not to send the values through it in the first place.

3.2 Sequential Copy Process

GSCopy Code

Listing 11: "GSCopy.groovy"

```
void run ()
{
   while (true)
   {
      def i = inChannel.read()
      // output the input value in sequence to each output channel
      outChannel0.write(i)
      outChannel1.write(i)
   }
}
```

Explanation Within the GSCopy the input value is copied by outputting the value to both output channels in sequence not in parallel.

GSPairsA

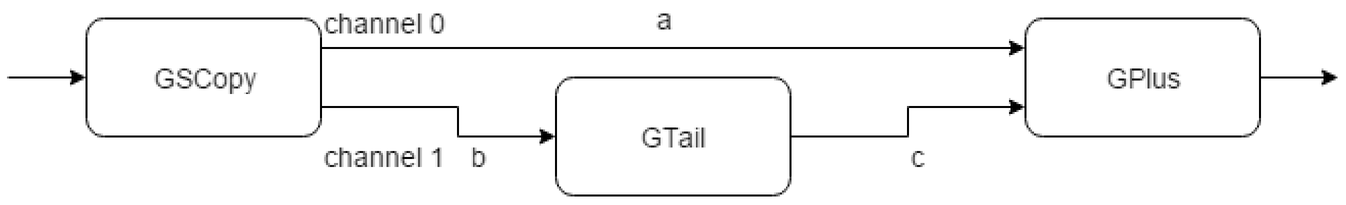
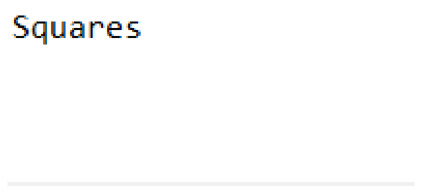


Figure 8: **Exercise 3-2** - Process Network diagram of GSPairsA, as the copy is sequential the output is written first to channel **a** then to channel **b**.

Output



```
Squares
```

Figure 9: **Excercise 3-2** - Output from the Squares system, using the *GSPairsA.groovy* process.

GSPairsB

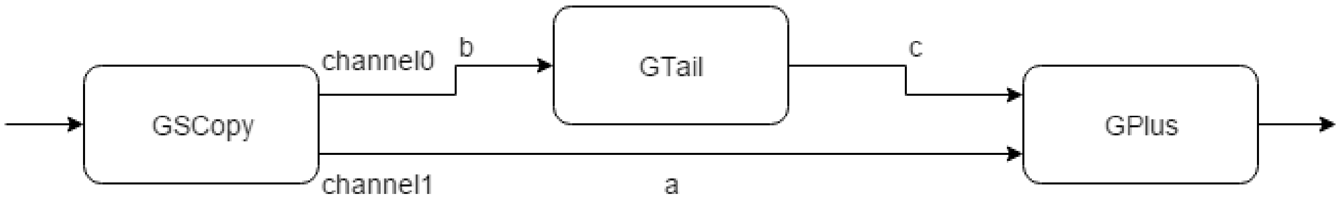


Figure 10: **Exercise 3-2** - Process Network diagram of GSPairsA, as the copy is sequential the output is written first to channel **b** then to channel **a**.

Output

```
Squares
1
4
9
16
25
36
49
64
81
100
```

Figure 11: **Excercise 3-2** - Output from the Squares system, using the *GSPairsB.groovy* process.

Questions

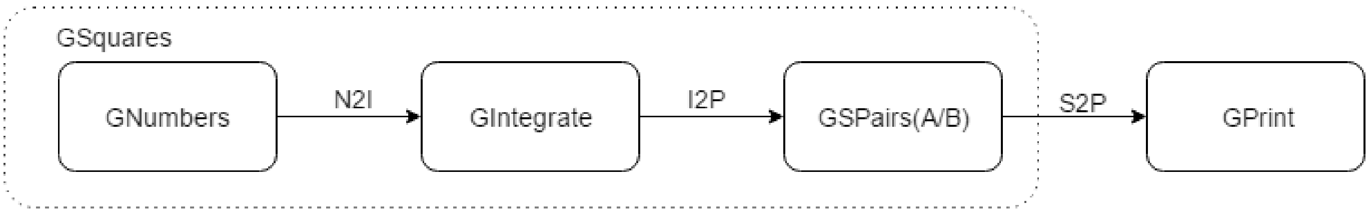


Figure 12: **Exercise 3-2** - Process Network diagram of test system, GSPairs is replaced by GSPairsA and GSPairsB to see the effect.

Determine the effect of the change between GSPairsA and GSPairsB, why does this happen?

In running *GSPairsA.groovy* the output, shown in Figure 9, is blank after the header, whereas in running *GSPairsB.groovy* the output, shown in Figure 11, displays a heading and then a stream of square numbers. The reason for this change is due to the fact *GSCopy.groovy* outputs sequentially and even though the channels are the same, the order in which they are written to is different. See Figures 8 & 10. In the instance using *GSPairsA.groovy* the system deadlocks.*GPlus.groovy* reads both inputs in parallel so cannot process the values until both channels have an input. *GTail.groovy* removes the first input it is given and then outputs the remaining numbers. However *GTail.groovy* cannot receive another input until channel a (the channel between *GSCopy.groovy* and *GPlus.groovy*) is free. The instance using *GSPairsB* does not deadlock as channel b - the channel between *GSCopy.groovy* and *GTail.groovy* - is written to first, so will be able to receive and send the next copied value on the next iteration of the sequential copy.

3.3 Parallel Print

Questions

Why was it considered easier to build GParPrint as a new process rather than using multiple instances of GPrint to output the table of results?

It is easier to build a parallel printing process rather than using multiple instances of *GPrint* as it allows for printing results from multiple processes at once. It also allows for However as a parallel print needs to wait for every input process to return a value to print, it means that it can only output as fast as the slowest process. Therefore it would only be easier to use if the processes to print output at a regular rate.

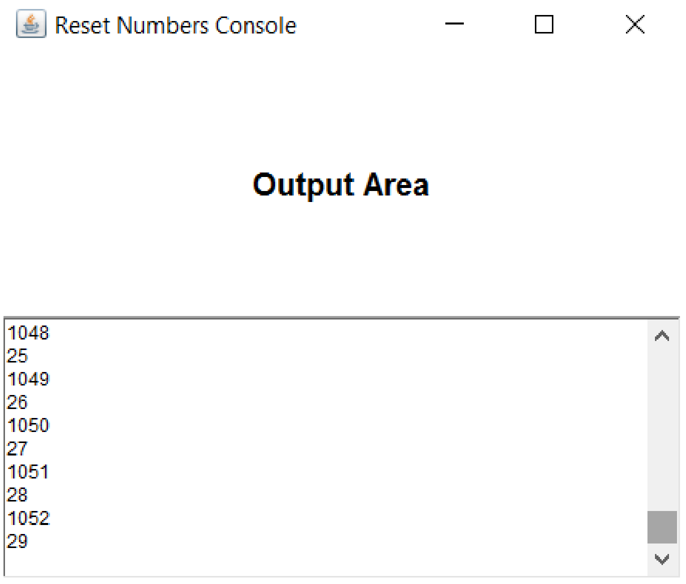
Exercise 4

4.1 ResetPrefix

Listing 12: "Line 25 of ResetPrefix.groovy"

```
        inChannel.read()
```

Output



```
1048
25
1049
26
1050
27
1051
28
1052
29
```

Figure 13: **Exercise 4-1** - Output from the ResetNumbers program when line 25 of *ResetPrefix.groovy* is removed.

Questions

What happens if line 25 of ResetPrefix Listing 4-1 is commented out? Why?

When the read line is removed the output alternates between the incrementing reset value and the original numbers as seen in figure 13. This is because the original value is not removed from the system if the channel is not read from.

Explore what happens if you try to send several reset values hence, explain what happens and provide a reason for this.

When you try to send several reset values at once deadlock occurs. This is due to the first number not being removed from channel c, shown in figure 14, and therefore the processes cannot read additional inputs from the resetChannel as there is no way for *GPCopy.groovy* to output in parallel until the channel is cleared.

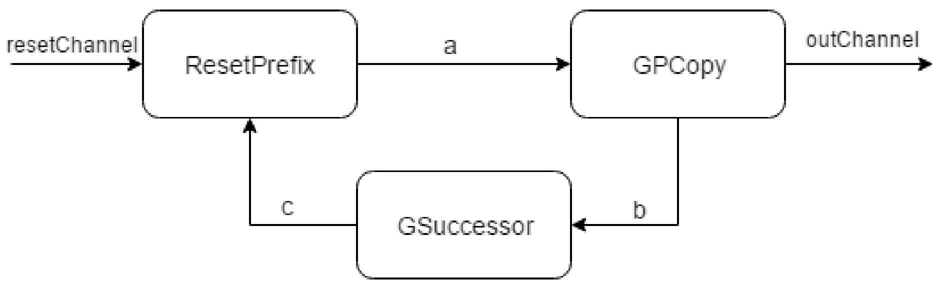


Figure 14: **Exercise 4-1** - Process network diagram for *ResetNumbers.groovy*

4.2 ResetSucessor process

Code

Listing 13: "ResetSucessor.groovy"

```
while (true)
{
    // deal with inputs from resetChannel and inChannel
    // use a priSelect
    def index = alt.priSelect();

    if (index == 0)        // reset Channel input
    {
        def resetVal = resetChannel.read()
//      inChannel.read()
        outChannel.write(resetVal)
    }
    else                   // outChannel input
    {
        outChannel.write(inChannel.read() +1)
    }
}
```

Listing 14: "ResetNumbers.groovy"

```
def testList = [
    new GPrefix ( prefixValue: initialValue,
        outChannel: a.out(),
        inChannel: c.in() ),
    new GPCopy ( inChannel: a.in(),
        outChannel0: outChannel,
        outChannel1: b.out() ),
    // requires a constructor for ResetSuccessor
    new ResetSuccessor (inChannel: b.in(),
        outChannel: c.out(),
        resetChannel: resetChannel )
]
```

Question

Does it overcome the problem identified in Exercise 1? If not, why not?

No, the reformulation does not overcome the problem as it is still is possible for the user to input multiple values, deadlocking the system. There is no way to reformulate this example without reading from the channel as the integers are never removed from the system.

Exercise 5

5.1 Varying delay for RunQueue

Questions *By varying the delay times demonstrate that the system works in the manner expected. What do you conclude from these experiments?*

Varying the delay times makes no difference to the process output, it works as expected regardless of changing the times. This happens because it uses preconditions to prevent reading and overwriting when a producer sends a new integer into the Queue. The Consumer is not able to read until it is ready, but it will continue to read in a loop regardless of whether the value is NULL so no values will be missed and the output will always be the same.

5.2 Preconditions

Code

Listing 15: "Scale.groovy"

```
while (true)
{
   switch ( scaleAlt.priSelect(preCon) )
   {
      case SUSPEND :
         // deal with suspend input
         suspend.read()
         factor.write(scaling)
         suspended = true
         println "Suspended"
         preCon[SUSPEND] = false
         preCon[INJECT] = true
         break
      case INJECT:
         // deal with inject input
         scaling = injector.read()
         println "Injected scaling is $scaling"
         suspended = false
         timeout = timer.read() + DOUBLE_INTERVAL
         timer.setAlarm(timeout)
         preCon[SUSPEND] = true
         preCon[INJECT] = false
         suspended = false
         break
      case TIMER:
         // deal with Timer input
         timeout = timer.read() + DOUBLE_INTERVAL
         timer.setAlarm ( timeout )
         scaling = scaling * 2
         println "Normal Timer: new scaling is ${scaling}"
         break
      case INPUT:
         //  deal with Input channel
         def inValue = inChannel.read()
         def result = new ScaledData()
         result.original = inValue
         result.scaled = inValue * scaling
         outChannel.write( result )
         break
   } //end-switch
} //end-while
```

Questions

Which is the more elegant formulation? Why?

The latter, using preconditions, is the more elegant solution because it avoids nested loops. Using only one switch statement and alternative makes the code more readable and possibly could increase performance due to a decrease in conditional loops.

Exercise 6

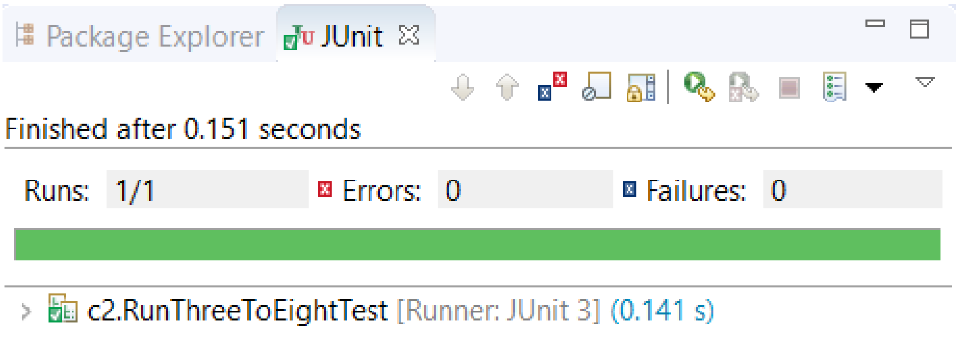
6.1 Test Case for Three-To-Eight

Code

Listing 16: "RunThreeToEightTest.groovy"

```
class RunThreeToEightTest extends GroovyTestCase
{
   void testThreeToEight()
   {
      One2OneChannel genToStream = Channel.one2one()
      One2OneChannel streamToEight = Channel.one2one()

      def gen = new GenerateSetsOfThree ( outChannel: genToStream.out())
      def list = new ListToStream ( inChannel: genToStream.in(), outChannel: streamToEight.out())
      def eight = new CreateSetsOfEight ( inChannel: streamToEight.in())
```

```

        def testRunList = [gen, list, eight]
        new PAR(testRunList).run()

        // test output is correct from eight
        def expectedList = list.inTest
        def actualList = eight.outTest

        println "exp ${expectedList} + act ${actualList}"

        assertTrue(expectedList == actualList)

        assertTrue(eight.outList.size() == 8)
    }
}
```



```
Finished after 0.151 seconds
Runs: 1/1    Errors: 0    Failures: 0
c2.RunThreeToEightTest [Runner: JUnit 3] (0.141 s)
```

Figure 15: **Exercise 6-1** - Screen capture of the results of the JUnit test performed.

Output

Explanation The performed JUnit test shown above adds each integer from the *ListToStream.groovy* process in turn and checks it against the output from *CreateSetsOfEight.groovy* to check if the values are correctly being processed through the system. It also checks that the set returned from the *CreateSetsOfEight.groovy* process correctly contains 8 integers.

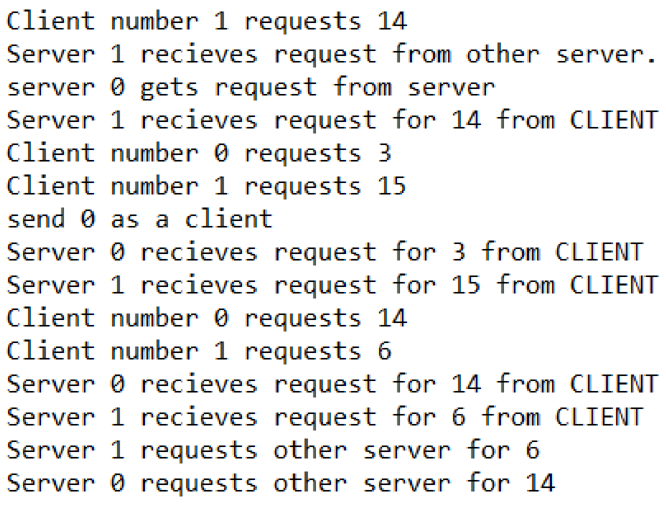
Exercise 7

7.1 Deadlock

Determine the precise nature of the deadlock in the Client Server system.

By adding a parameter to identify each server you can see from the output of the program, figure 16, that both servers request values from each other at the same time which is the cause of the deadlock in the system. Neither process can send a response as they first are expecting a response for their own request and therefore cannot continue.

Output



```
Client number 1 requests 14
Server 1 recieves request from other server.
server 0 gets request from server
Server 1 recieves request for 14 from CLIENT
Client number 0 requests 3
Client number 1 requests 15
send 0 as a client
Server 0 recieves request for 3 from CLIENT
Server 1 recieves request for 15 from CLIENT
Client number 0 requests 14
Client number 1 requests 6
Server 0 recieves request for 14 from CLIENT
Server 1 recieves request for 6 from CLIENT
Server 1 requests other server for 6
Server 0 requests other server for 14
```

Figure 16: **Exercise 7-1** - Output from the Server Client process, deadlock occurs as each server requests a response from the other.

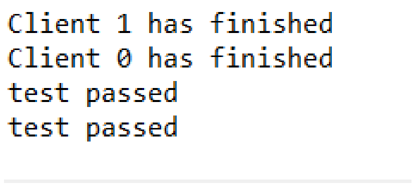
Exercise 8

8.1 Test Case for Client Server System

Code

Listing 17: "Client.groovy"

```
void run()
{
   def iterations = selectList.size
   println "Client $clientNumber has $iterations values in $selectList"

   for ( i in 0 ..< iterations) {
      def key = selectList[i]
      println "Client number $clientNumber requests $key"
      requestChannel.write(key)
      def v = receiveChannel.read()

      // add response from server to actual response list
      actualList << v
   }

   println "Client $clientNumber has finished"

   // multiply each value by ten and add in order from selectList = expected value from server
   for(i in 0 ..< iterations)expectedList << selectList[i]*10

   // check if actual equals expected
   if (actualList.equals(expectedList))
      println "test passed"
   else
      println "test failed"
}
```



```
Client 1 has finished
Client 0 has finished
test passed
test passed
```

Figure 17: **Excercise 8-1** - Output from the Server Client test.

Output

Explanation To ensure that the values returned from the Server arrive in the order expected according to their selectList property, a list was created to store the actual output from the server. The selectList was then iterated through multiplying each value by ten and storing the result into a separate list. These lists were compared to test the system.

Exercise 9

9.1 Missed Test

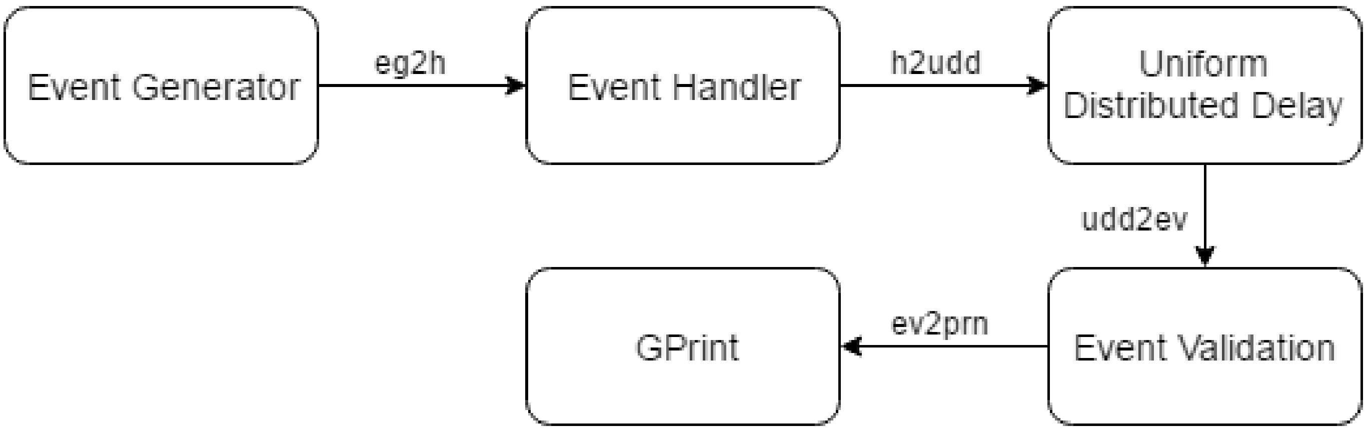


Figure 18: **Excercise 9-1** - Process network diagram for event validation system.

Explanation In constructing a test case for the event handling system, a new *EventValidation.groovy* process was implemented, see Listing 18. The process was inserted into the system between the *UniformDistributedDelay.groovy* and the *GPrint.groovy* process.

To test if the number of missed events is correct, the previous EventData object to pass through the system is stored and updated each loop. EventData instances store the number of missed events that are printed to the console. The algorithm checks whether the missed event stored by the current EventData object is correct by checking against the calculated value, the current data minus the previous data minus 1.

Code

Listing 18: "EventValidation.groovy"

```
class EventValidation implements CSProcess
{
    def ChannelInput inChannel
    def ChannelOutput outChannel
    def EventData previous

    void run()
    {
        while (true)
        {
            // read in current event data
            def currentEvent = (EventData)inChannel.read()

            // previous is null for first event
            if (previous != null)
            {

                def actualMissed = currentEvent.data − previous.data − 1

                // check if current missed event value is the same as the calculated missed value
                if (currentEvent.missed != actualMissed)
                {
                    println "Error: Missed count inaccurate.\n"
                        +"Event data: $currentEvent.data missed count = $currentEvent.missed, should be $actualMissed"
                }
            }

            // send along to GPrint and set new previous event
            outChannel.write(currentEvent)
            previous = currentEvent

        } // end while

    }

}
```

9.2 MultiStream

Exercise 9 2 (5.marks) The accompanying exercise package contains a version of the event handling system, RunMultiStream, which allows the creation of 1 to 9 event streams. By modifying the times associated with each event generation stream and also of the processing system explore the performance of the system. What do you conclude?

By altering the timings associated with the event generator, the amount of data that can be stored in the buffer can be increased. This reduces the amount of missed data. The reason that the size of the buffer is inversely proportional to the amount of missed data is that once the buffer is filled it cannot store events. Events are constantly written to the *EventProcessor.groovy* regardless of whether the buffer is full. If events are generated too quickly, the *EventProcessor.groovy* process is not ready to receive all the data resulting in missed events. Evidence of this can be seen as the initial events that are generated usually pass through the system without any loss, this is because at the start of the system the buffer is empty.

9.3 Multiplexer Variations

The process EventProcessing has three versions of multiplexer defined within it, two of which are commented out. Each process has a similar function that writes a chosen input channel to the output. The difference lies in how the input channel to be read is chosen.

FairMultiplex The *FairMultiplex.groovy* uses a "fair" select alternative if multiple guards are ready. Meaning that each channel in the list has an equal opportunity to be read from. This is similar to a FIFO queue system where each guard once read from is given the lowest priority - the first channel to be serviced is read initially and then added to the end of the list so it evens out the selection of each channel.

PriMultiplex The *PriMultiplex.groovy* uses a "priority" select alternative if multiple guards are ready. A channel's priority is determined by the order it appears within the channel list. The lower the index of the channel, the higher the priority it is given. Therefore this process reads from the channel with the lowest index of the currently ready channels.

Multiplexer The *Multiplexer.groovy* makes an arbitrary choice of ready guards. The arbitrary choice actually implements that of the fair select alternative, just renamed. This is misdirection, as the process itself does nothing unique and therefore can be said to be useless.

Summary Out of each process to choose from, the *FairMultiplex.groovy* seems to be the most useful. This is because for any system that requires to read from a selection of input channels, a process that doesn't favour any inputs makes more sense to use. The priority selection is useful but in a system where the lower priority channels will almost never be chosen, it could be argued that an alternative is not necessary at all. The *Multiplexer.groovy* has no real use as it is just a copy of the fair select.

Exercise 11

11.1 Scaling User Interface

Code

Listing 19: "UserInterface.groovy"

```
class UserInterface implements CSProcess
{

    // define inputs
    def ChannelInput toConsole
    def ChannelOutput fromConsole
    def ChannelInput clearInputArea

    def ChannelInput suspendButton
    def ChannelOutput suspend

    // title of console
    def String frameLabel = "Scaler Console"

    def void run()
    {
        // define UI frames
        def main = new ActiveClosingFrame (frameLabel)
        def root = main.getActiveFrame()
        root.setLayout ( new BorderLayout () )

        // draw labels
        def outLabel = new Label ("Scaled Output", Label.CENTER)
        outLabel.setFont(new Font("sans-serif", Font.BOLD, 20))
        def inLabel = new Label ("Input New Scale", Label.CENTER)
        inLabel.setFont(new Font("sans-serif", Font.BOLD, 20))

        // set output here
        def outText = new ActiveTextArea ( toConsole, null, "", 0, 0, java.awt.TextArea.SCROLLBARS_VERTICAL_ONLY)

        // input from console
        def inText = new ActiveTextEnterField ( clearInputArea, fromConsole )

        // setup layout
        def console = new Container()
        console.setLayout ( new GridLayout ( 5, 1 ) )
        console.add ( outLabel )
        console.add ( outText )
        console.add ( inLabel )
        console.add ( inText.getActiveTextField() )

        // add button
        def button = new ActiveButton (suspendButton, suspend, "SUSPEND")
        button.setFont(new Font("sans-serif", Font.BOLD, 20))
        console.add(button)

        root.add(console, BorderLayout.CENTER )
        root.pack()
        root.setVisible(true)

        def interfaceProcessList = [ main, outText, inText, button ]
        new PAR ( interfaceProcessList ).run()
    }
}
```

Listing 20: "Scale.groovy"

```
class Scale implements CSProcess
{
    def int scaling = 2
    def ChannelOutput outChannel
    def ChannelInput inChannel
    def ChannelInput suspend
    def ChannelInput injector

    def ChannelOutput toButton
    def ChannelOutput toConsole

    void run ()
    {
        def SUSPEND  = 0
        def INJECT   = 1
        def INPUT    = 2

        def normalAlt = new ALT ( [ suspend, injector, inChannel ] )
        def suspended = false

        while (true) {
            switch ( normalAlt.priSelect() ) {
                case SUSPEND :

                    //  deal with button input

                    // if suspend change button text and bool to stop scaling
                    if (suspend.read() == "SUSPEND")
                    {
                        toButton.write("RESTART")
                        outChannel.write("Suspended\n")
                        suspended = true
                    }
                    else // resume, change flag and button
                    {
                        toButton.write("SUSPEND")
                        outChannel.write("Resumed\n")
                        suspended = false
                    }
                    break

                case INJECT:
                    //  deal with inject input from console
                    scaling = Integer.parseInt(injector.read() ) // try parse data
                    outChannel.write("Injected scaling is $scaling \n")

                    // clear text field
                    toConsole.write("")

                    // ensure scaling is resumed
                    toButton.write("SUSPEND")
                    suspended = false
                    break

                case INPUT:
                    //   deal with Input channel

                    // read from timed data
                    def inValue = inChannel.read()
                    def result = new ScaledData()
                    result.original = inValue

                    // flag set by UI button if suspended, don't scale
                    if (suspended)
                    {
                        result.scaled = inValue
                    }
                    else
                    {
                        result.scaled = inValue * scaling
                    }

                    outChannel.write( result )
                    break
            } //end-switch
        } //end-while
    } //end-run
} // end Scale
```

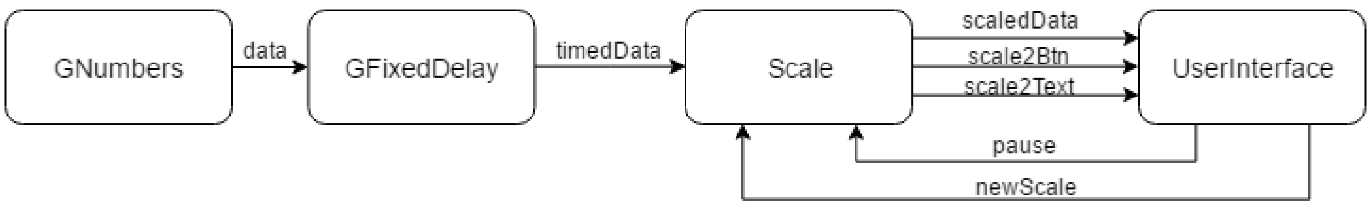
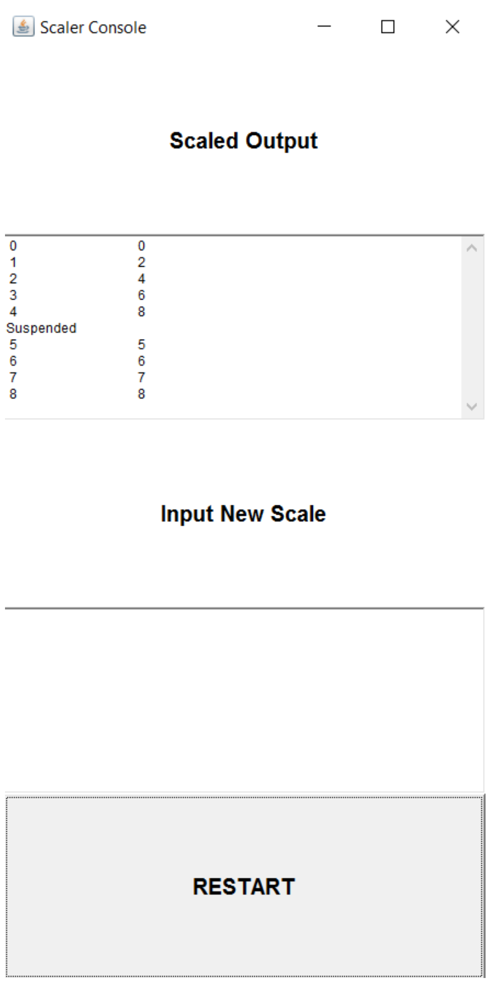


Figure 19: **Excercise 11** - Process network diagram for the scaler user interface system.

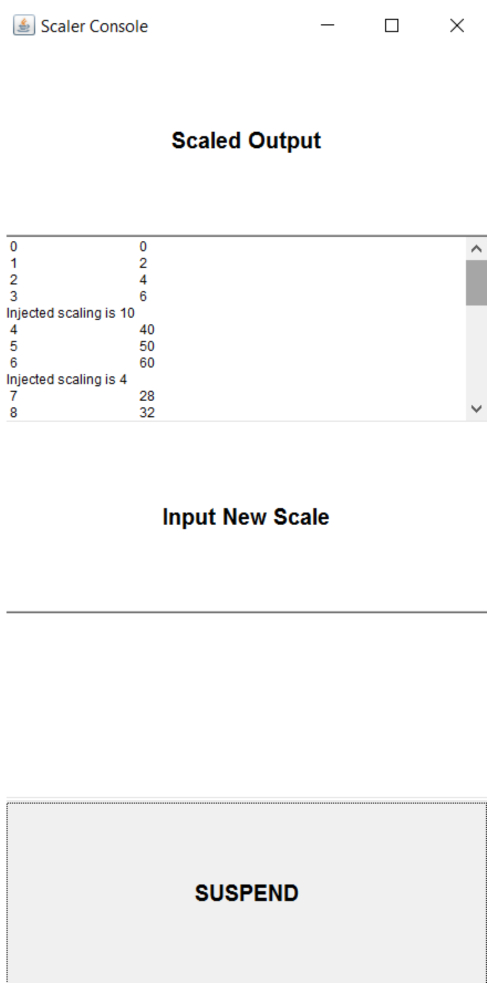
Output

Explanation



```
0           0
1           2
2           4
3           6
4           8
Suspended
5           5
6           6
7           7
8           8
```

(a) The Suspend button has been clicked



```
0           0
1           2
2           4
3           6
Injected scaling is 10
4           40
5           50
6           60
Injected scaling is 4
7           28
8           32
```

(b) Numbers have been Injected using the textbox.

Figure 20: Output captures from the Scaler User Interface.