

Fundamentals of Parallel Systems Lab Book

Jordan Stephano Gray
40087220
graybostephano@gmail.com

April 2016

1 Exercise 2.1

The Producer takes in an integer based on user input. If the integer is less than or equal to 0 the program finishes. The program will only accept integers less than or equal to 100. The result of the input is written out to the channel.

The Multiplier has a variable called factor which is defined as 2. An integer with the label "i" is then assigned whichever value was passed in through the previous channel and multiplies that value by the factor (2). This new value is then written to the next channel. The next value of i is then read.

This does the same as the previous multiplier which results in a value that has been multiplied by a factor of 2, twice. The result is then passed out to the next channel.

This takes in the value from the previous channel and sets it to a variable "i". The result is then printed in the console and the next "i" value is read.

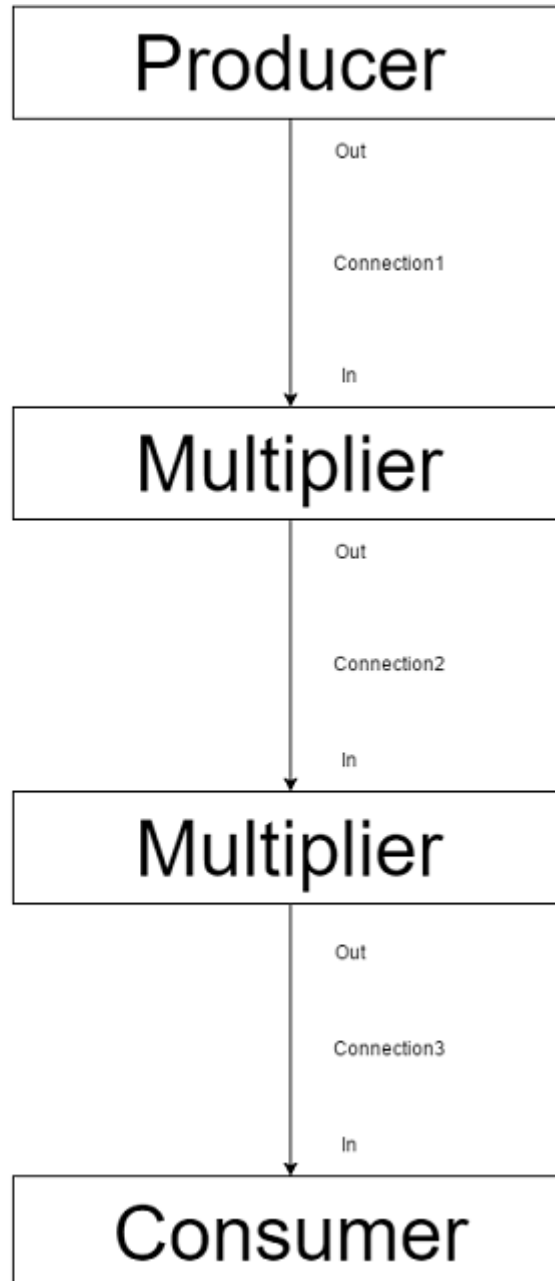


Figure 1: Diagram of the system for multiplication.

1.1 Consumer

```
void run() {
    def i = inChannel.read()
    while ( i > 0 ) {
        //Insert modified print line statement
        println "\nResult: $i"
        i = inChannel.read()
    }
    println "Finished"
}
```

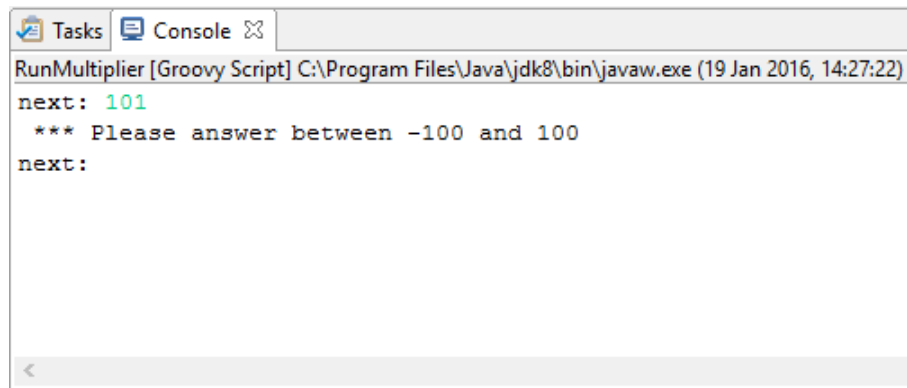
1.2 Multiplier

```
class Multiplier implements CProcess {

    def ChannelOutput outChannel
    def ChannelInput inChannel
    def int factor = 2

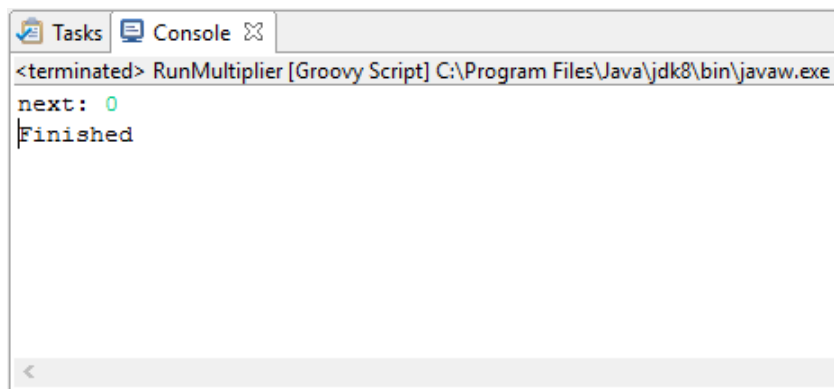
    void run() {
        def i = inChannel.read()
        while (i > 0) {
            // write i * factor to outChannel
            outChannel.write( i * factor )
            // read in the next value of i
            i = inChannel.read()

        }
        outChannel.write(i)
    }
}
```



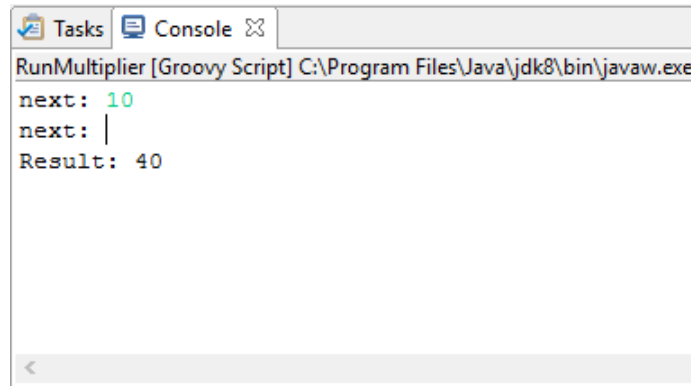
```
Tasks Console
RunMultiplier [Groovy Script] C:\Program Files\Java\jdk8\bin\javaw.exe (19 Jan 2016, 14:27:22)
next: 101
*** Please answer between -100 and 100
next:
```

Figure 2: This demonstrates what happens when an input outside of the range (-100 - 100) is used.



```
Tasks Console
<terminated> RunMultiplier [Groovy Script] C:\Program Files\Java\jdk8\bin\javaw.exe
next: 0
Finished
```

Figure 3: This demonstrates what happens when 0 is used as an input and the program terminates.



```
RunMultiplier [Groovy Script] C:\Program Files\Java\jdk8\bin\javaw.exe
next: 10
next: |
Result: 40
```

Figure 4: This demonstrates what happens when a valid input is used.

2 Exercise 2.2

Q: What change is required to output objects containing six integers?

Ans: Modify the for loop used in “CreateSetsOfEight” to loop for every index from 0 to 5 rather than 0 to 7.

Q: How could you parametrize this in the system to output objects that contain any number of integers?

Ans: Create an integer variable to hold the list parameter and substitute it into the for loop in “CreateSetsOfEight”. The variable is then easily changed in “RunThreeToEight”.

Q: What happens if number of integers required in the output stream is not a factor of total number of integers in input stream?

Ans: Any remaining numbers from input stream are added to an object but because the object is never full it is left out of the output stream and the program can’t terminate.

GenerateSetsOfThree defines a list with each index holding three numbers and writes this list to the outChannel.

ListToStream then takes this data through the inChannel and separates these groups of three into individual integers and streams them to the outChannel.

GenerateSetsOfEight then defines a new list to hold the new assortment of values, the listParameter is set to 7 which means the data will be split into sets of 8 (including index 0 – 7). After the values have been amended to the output list the list is printed of with a simple print statement.

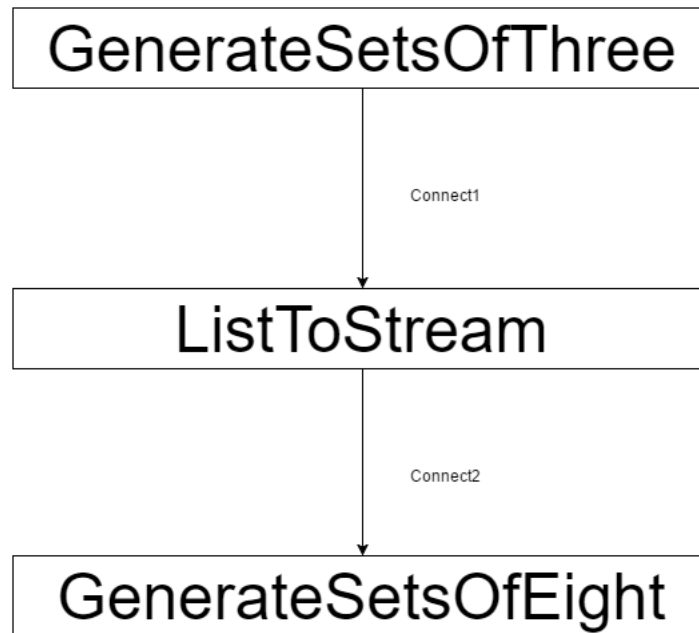


Figure 5: Diagram showing the structure of exercise 2-2.

2.1 ListToStream

```
void run (){
    def inList = inChannel.read()
    while (inList[0] != -1) {
        // hint: output list elements as single integers
        outChannel.write(inList[0])
        outChannel.write(inList[1])
        outChannel.write(inList[2])
        inList = inChannel.read()
    }
    outChannel.write(-1)
}
```

2.2 CreateSetsOfEight

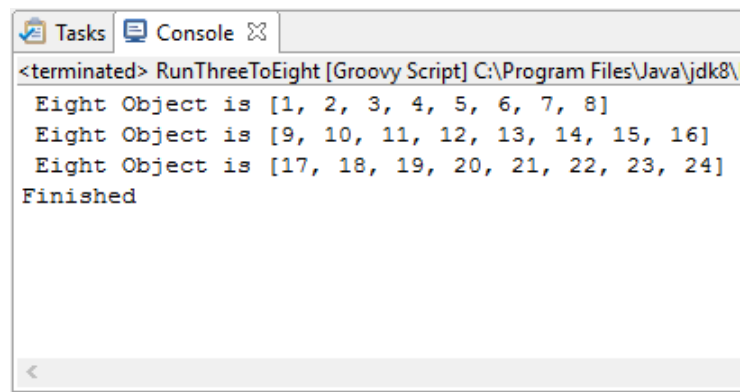
```
def ChannelInput inChannel
def int listParam = 7

void run(){
```

```

def outList = []
def v = inChannel.read()
while (v != -1){
    for ( i in 0 .. listParam ) {
        // put v into outList and read next input
        outList << v
        v = inChannel.read()
    }
    println " Eight Object is ${outList}"
    outList = []
}
println "Finished"
}

```

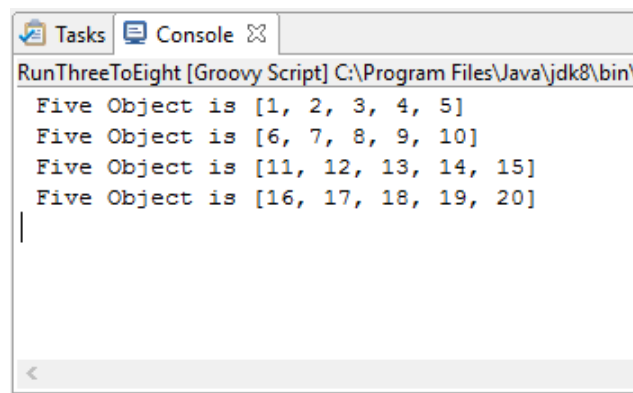


```

<terminated> RunThreeToEight [Groovy Script] C:\Program Files\Java\jdk8\
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 6: Shows the program producing eights.



```

RunThreeToEight [Groovy Script] C:\Program Files\Java\jdk8\bin\
Five Object is [1, 2, 3, 4, 5]
Five Object is [6, 7, 8, 9, 10]
Five Object is [11, 12, 13, 14, 15]
Five Object is [16, 17, 18, 19, 20]
|

```

Figure 7: Shows the program producing fives with left overs.

3 Exercise 3.1

Q: Which is the more pleasing solution and why?

Ans: I found that using the Minus method was more pleasing to me. Although using the Negator makes more sense by just inverting the value to a negative number although there is also an extra intermediate step.

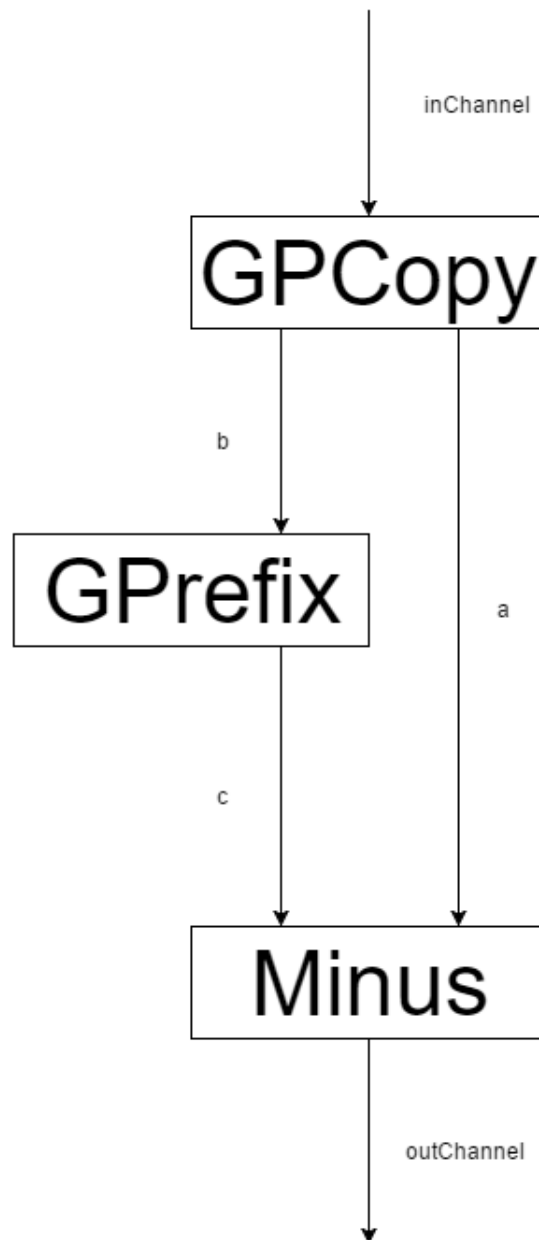


Figure 8: System using the Minus function.

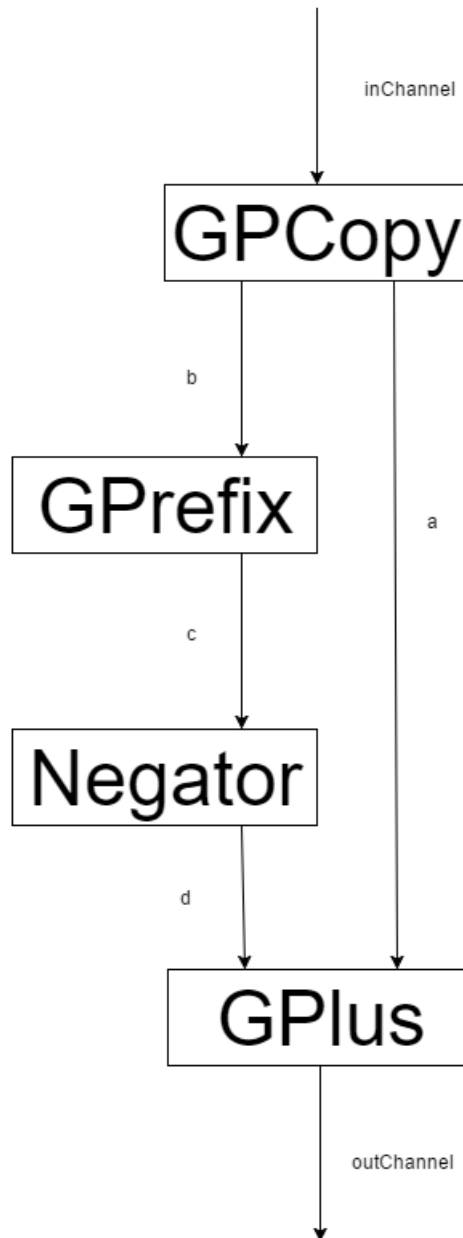


Figure 9: System using the Negator function.

3.1 Differentiate

```
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 )
]

new PAR ( differentiateList ).run()
}
```

3.2 DifferentiateNeg

```
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 )
]

new PAR ( differentiateList ).run()
}
```

3.3 Minus

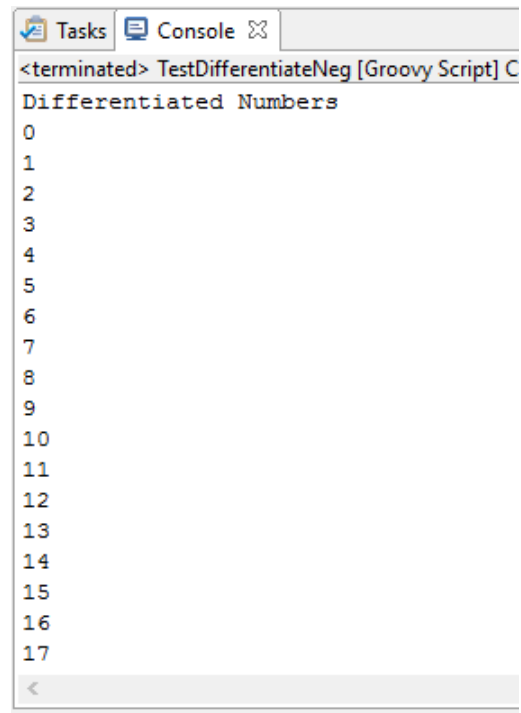
```
void run () {

    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)
    }
}
```

3.4 Negator

```
void run () {
    while (true) {
        //output the negative of the input value
        outChannel.write(-inChannel.read())
    }
}
```



```
<terminated> TestDifferentiateNeg [Groovy Script] C
Differentiated Numbers
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
<
```

Figure 10: Results from running DifferentiateNegator.

4 Exercise 3.2

Q: Determine the effect of the change and why it happens.

Ans: The output channels of pairs A and B are switched. With GSPairsA if an output isn't ready to receive data the process deadlocks as GSPairsA writes to GPlus first and GSPairsB doesn't.

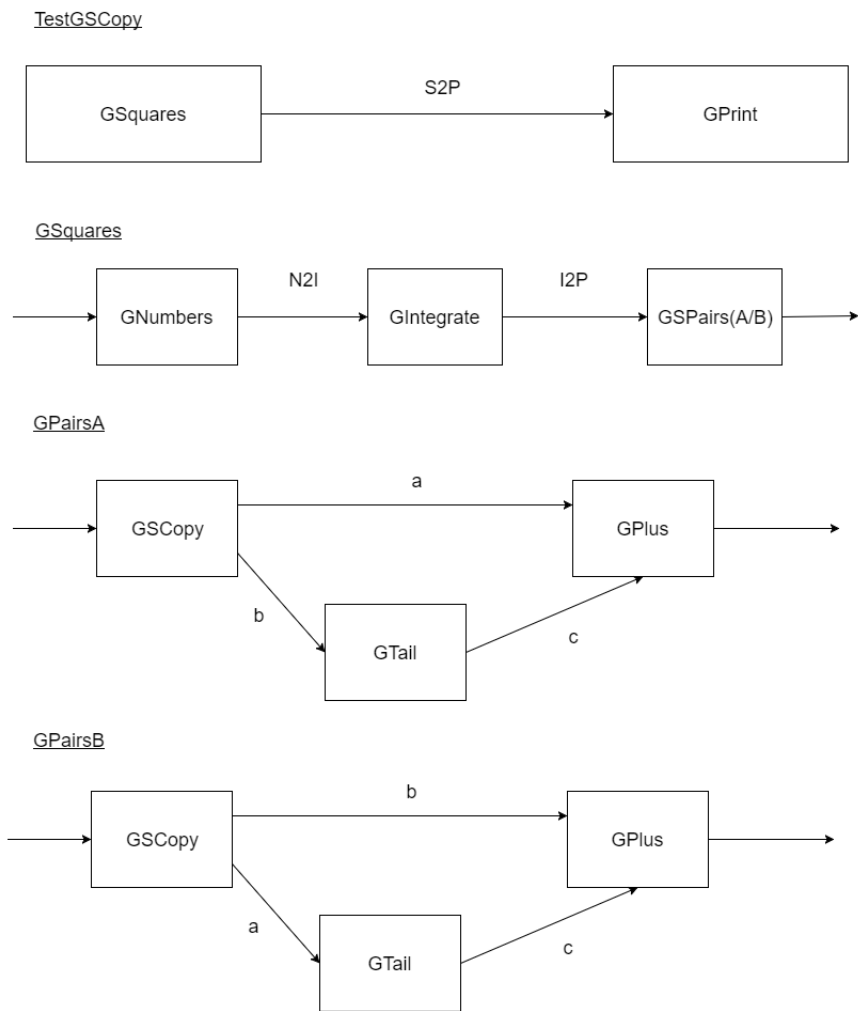


Figure 11: Process Diagrams for exercise 3-2.

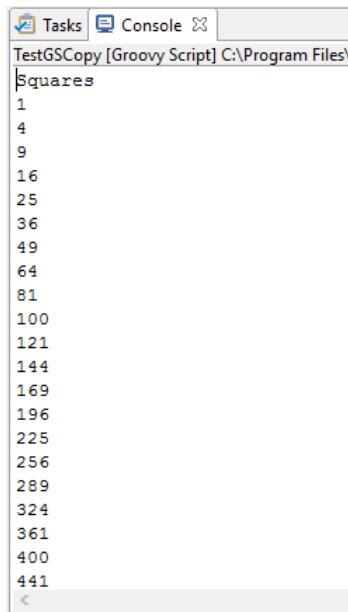


Figure 12: TestGSCopy program running.

4.1 GSCopy

```
void run () {  
  
    def write0 = new ProcessWrite(outChannel0)  
    def write1 = new ProcessWrite(outChannel1)  
    def parWrite2 = new PAR([write0, write1])  
  
    while (true) {  
        def i = inChannel.read()  
        // output the input value in sequence to each output channel  
        write0.value = i  
        write1.value = i  
        parWrite2.run()  
    }  
}
```

4.2 GSquares

```

void run () {

    One2OneChannel N2I = Channel.createOne2One()
    One2OneChannel I2P = Channel.createOne2One()

    def testList = [ new GNumbers ( outChannel: N2I.out() ),
                     new GIntegrate ( inChannel: N2I.in(),
                                      outChannel: I2P.out() ),
                     new GSPairsB ( inChannel: I2P.in(),
                                   outChannel: outChannel )
                     // you will need to modify this twice
                     //first modification is to insert a constructor for
                     //GSPairsA
                     // then run the network using TestGSCopy
                     //second modification replace the constructor for
                     //GSPairsA with GSPairsB
                     // then run the network again using TestGSCopy
                     // you will then be able to compare the behaviour
                     // and also to
                     // explain why this happens!
                     //GSPairsA does not work
                     ]
    new PAR ( testList ).run()
}

```

5 Exercise 3.3

Q: 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?

Ans: GParPrint makes for easier processing using a collection of channels. This is done using the GPCopy process at intermediate stages in the pipeline depending on what data is to be shown in the table whilst still passing on the data to the next process. Unlike GPrint, GParPrint prints the data in a table-like format with columns and makes the data more readable where GPrint prints the data in rows and makes it less readable due to the complex routing of the channels. Doing this also reduces the chances of errors by simplifying the system.

6 Exercise 4.1

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

Ans: The Alt can switch which channel it receives its input from, the resetChan-

nel and the inChannel. In this case the resetChannel is given priority over the inChannel. An initial number is set and this number is going upwards in increments of 1 and sent to the outChannel. If there is any input from the user it will come in through the resetChannel and the program should begin incrementing by 1 from this new value. Once the new input data has been read the system must wait for the outputChannel to be ready to receive data from the inChannel. This is verified by waiting for the inChannel.read (line 25) to return a value.

If line 25 is commented out the program will continue to increment the previous values whilst also outputting the new values which are being incremented by 1, if another input is entered then the system deadlocks. This happens due to the lack of processes which can take in new numbers, this could be overcome by possibly adding a buffer or simply more processes.

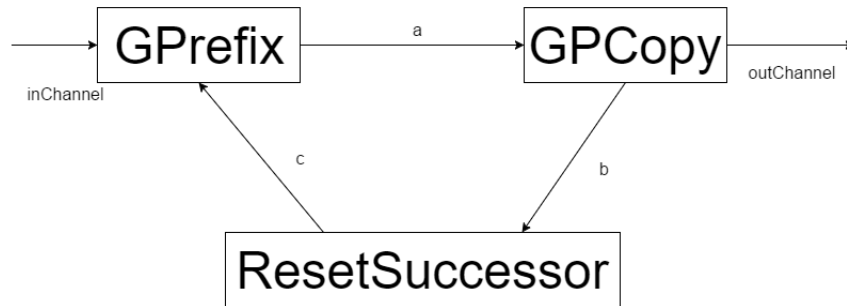


Figure 13: Diagram of the ResetNumbers process.

7 Exercise 4.2

Q: You will have to write a ResetSuccessor process. Does it overcome the problem identified in the previous exercise? If not, why not?

Ans: No this does not overcome the problem identified. This is because the ResetSuccessor process needs to read from the last iteration before continuing any further in the cycle.

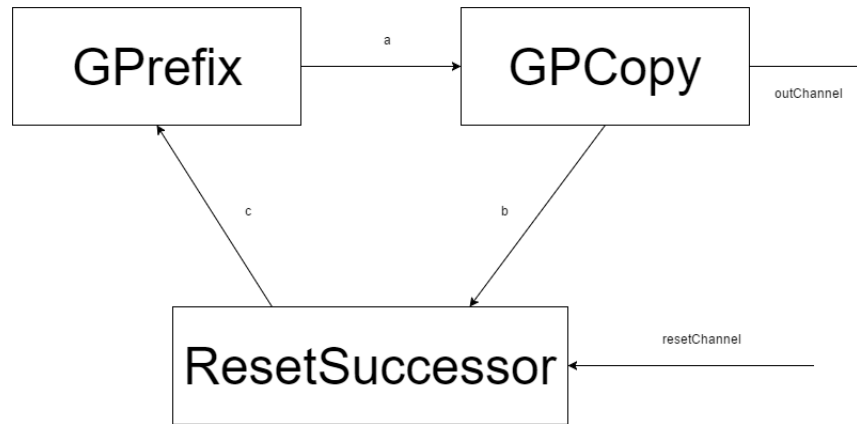


Figure 14: Diagram of the ResetNumbers process.

7.1 ResetSuccessor

```

void run () {
    def guards = [ resetChannel, inChannel ]
    def alt = new ALT ( guards )
    while (true) {
        // deal with inputs from resetChannel and inChannel
        // use a priSelect

        def index = alt.priSelect()
        if(index == 0) { //Reset channel input
            def resetValue = resetChannel.read()
            inChannel.read()
            outChannel.write(resetValue)
        }
        else { //inChannel input
            outChannel.write(inChannel.read()+1)
        }
    }
}

```

7.2 ResetNumbers

```

void run() {

    One20neChannel a = Channel.createOne20ne()

```

```

One2OneChannel b = Channel.createOne2One()
One2OneChannel c = Channel.createOne2One()

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 )
               ]
new PAR ( testList ).run()
}

```

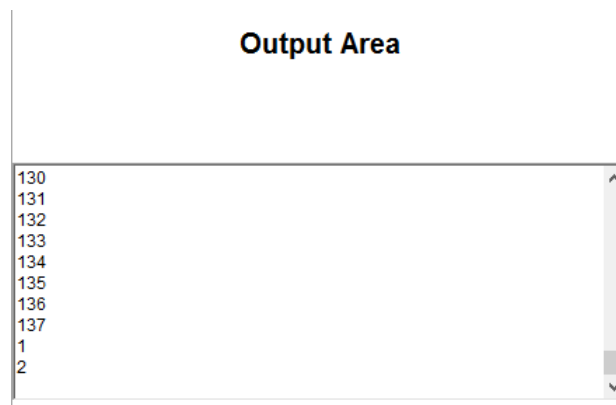


Figure 15: System output with inChannel.read()

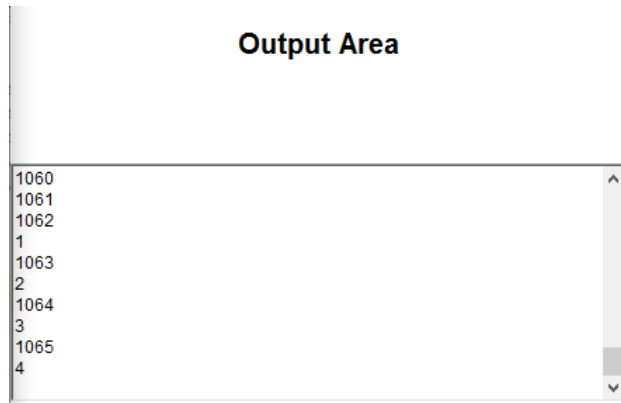


Figure 16: System output without `inChannel.read()` results in deadlock

8 Exercise 5.1

Q: What do you conclude from these experiments?

Ans: These results show that the delay effects the program equally when it is imposed on either of the processes. This happens because the two processes loop until completion and the delays begin at the same time rather than one after the other so after the initial delay the following process is ready to take in the data and process it.

This also shows that the Producer writes data to the queue based on the delay timer and doesn't need to know the state of the Consumer.

Table 1: Table of results.

Queue Delay Results		
QProducer Delay (ms)	Queue Delay (ms)	Total Time (secs)
0	0	1.85
500	0	26.77
1000	0	51.53
0	500	25.63
0	1000	47.49
500	500	26.69
1000	500	51.77
500	1000	51.53

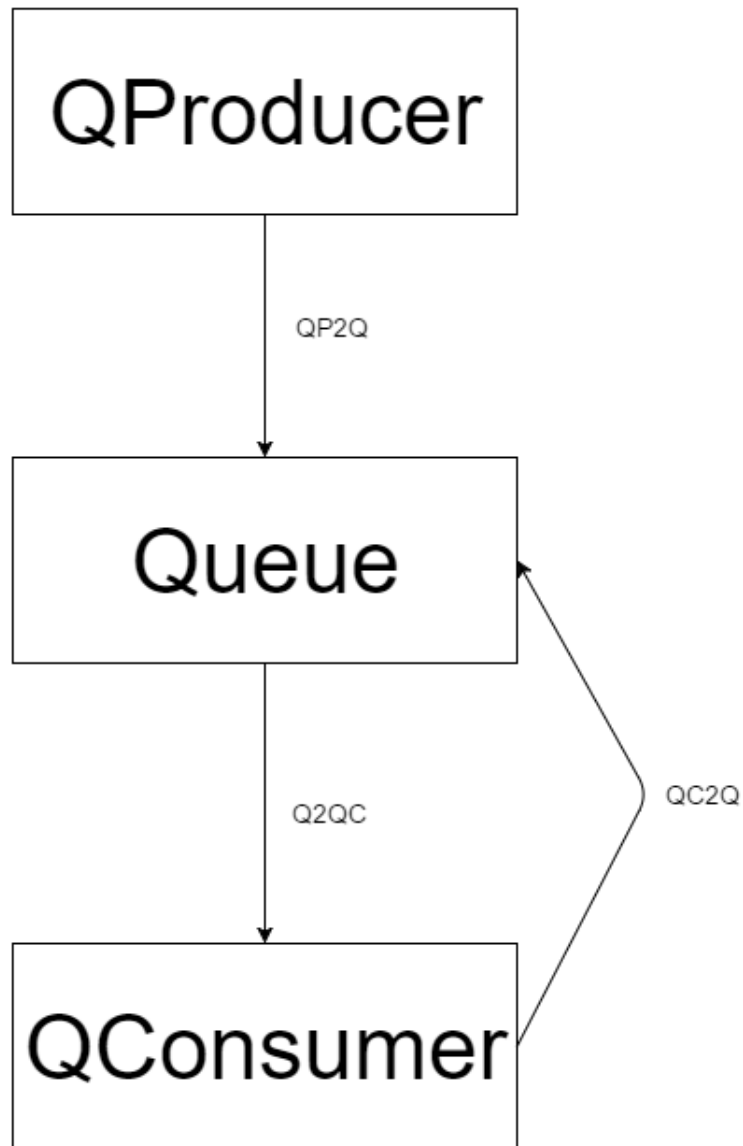
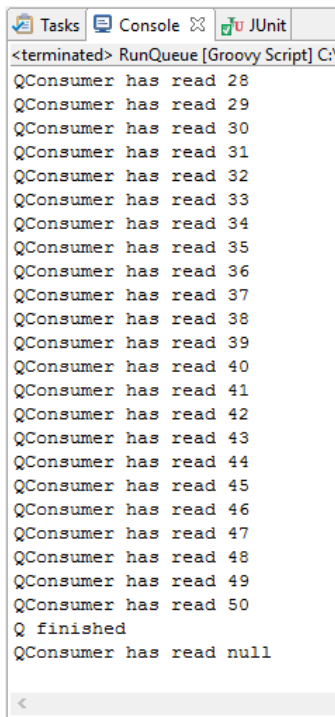


Figure 17: Queue system.



```
<terminated> RunQueue [Groovy Script] C:\
QConsumer has read 28
QConsumer has read 29
QConsumer has read 30
QConsumer has read 31
QConsumer has read 32
QConsumer has read 33
QConsumer has read 34
QConsumer has read 35
QConsumer has read 36
QConsumer has read 37
QConsumer has read 38
QConsumer has read 39
QConsumer has read 40
QConsumer has read 41
QConsumer has read 42
QConsumer has read 43
QConsumer has read 44
QConsumer has read 45
QConsumer has read 46
QConsumer has read 47
QConsumer has read 48
QConsumer has read 49
QConsumer has read 50
Q finished
QConsumer has read null
```

Figure 18: Queue system.

9 Exercise 5.2

Q: Which is the more elegant formulation? Why?

Ans: This solution is more elegant. The code is compact and far clearer/easier to read. This is because only one Alt and one switch statement is used in this solution. States can easily be switched back and forth.

9.1 Scale

```
while (true) {
    switch ( scaleAlt.priSelect(preCon) ) {
        case SUSPEND :
            // deal with suspend input
            suspend.read()
            factor.write(scaling)
```

```

println "SUSPEND"
preCon[INJECT] = true
preCon[SUSPEND] = true
    break
    case INJECT:
        // deal with inject input
        scaling = injector.read()
        println "INJECTED SCALING: $scaling"
        timeout = timer.read() + DOUBLE_INTERVAL
        timer.setAlarm(timeout)
        preCon[SUSPEND] = true
        preCon[INJECT] = false
        break
    case TIMER:
        // deal with Timer input
        timeout = timer.read() + DOUBLE_INTERVAL
        timer.setAlarm(timeout)
        scaling = scaling * 2
        println "NORMAL TIMER: NEW SCALING: ${scaling}"
        break
    case INPUT:
        // deal with Input channel
        def inValue = inChannel.read()
        def result = new ScaledData()
        result.original = inValue
        if(preCon[SUSPEND]){
            result.scaled = inValue * scaling
        }
        else{
            result.scaled = inValue
        }
        outChannel.write(result)
        break
    } //end-switch
} //end-while

```

```

<terminated> RunScaler [Groovy Script] C:\Program Fi
Original      Scaled
0             0
1             2
2             4
3             6
Normal Timer: new scaling is 4
4             16
5             20
SUSPEND
6             24
INJECTED SCALING: 5
7             35
8             40
9             45
10            50
11            55
Normal Timer: new scaling is 10
12            120
SUSPEND
13            130
INJECTED SCALING: 11
14            154
15            165
16            176
17            187
18            198
Normal Timer: new scaling is 22
19            418
SUSPEND

```

Figure 19: Scaling System output.

10 Exercise 6.1

10.1 TestCase

```

class TestCase extends GroovyTestCase {

    void test() {

        One2OneChannel connect1 = Channel.createOne2One()
        One2OneChannel connect2 = Channel.createOne2One()

        def generate = new GenerateSetsOfThree(outChannel: connect1.out())

        def List2Strm = new ListToStream(inChannel: connect1.in(),
                                         outChannel: connect2.out())
    }
}

```



```

def CreateSetsOEight = new CreateSetsOfEight(inChannel: connect2.in())

def testList = [generate, List2Strm, CreateSetsOEight]
new PAR (testList).run()

def expected = generate.sequence
def actual = CreateSetsOEight.OutSeq
assertTrue(expected == actual)

}

}

```

10.2 CreateSetsOfEight

```

class CreateSetsOfEight implements CSProcess{

    def ChannelInput inChannel
    def int listParam = 7

    def OutSeq = []

    void run(){
        def outList = []
        def v = inChannel.read()
        while (v != -1){
            for ( i in 0 .. listParam ) {
                // put v into outList and read next input
                outList << v

                OutSeq = OutSeq << v

                v = inChannel.read()
            }
            println " Eight Object is ${outList}"
            outList = []
        }
        println "Finished"
    }
}

```

10.3 GenerateSetsOfThree

```

class GenerateSetsOfThree implements CSProcess {

```

```

def ChannelOutput outChannel

def sequence = []

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] ]
  for ( i in 0 ..< threeList.size)
  {
    outChannel.write(threeList[i])

    sequence = sequence << threeList[i]
  }
  //write the terminating List as per exercise definition
  def terminatingList = [-1, -1, -1]
  outChannel.write(terminatingList)

  sequence = sequence.flatten()
  println "TERMINATING: $terminatingList $sequence "
}
}

```

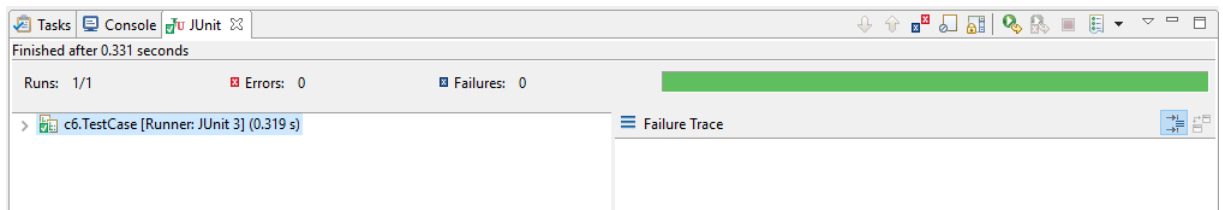


Figure 20: Test case working 6-1.

11 Exercise 7.1

Q: Determine the cause of the deadlock.

Ans: The deadlock only happens when both servers are communicating with each other at the same time and are both requesting data. They are both

waiting to receive the data and stuck in a continuous loop forever causing the deadlock.

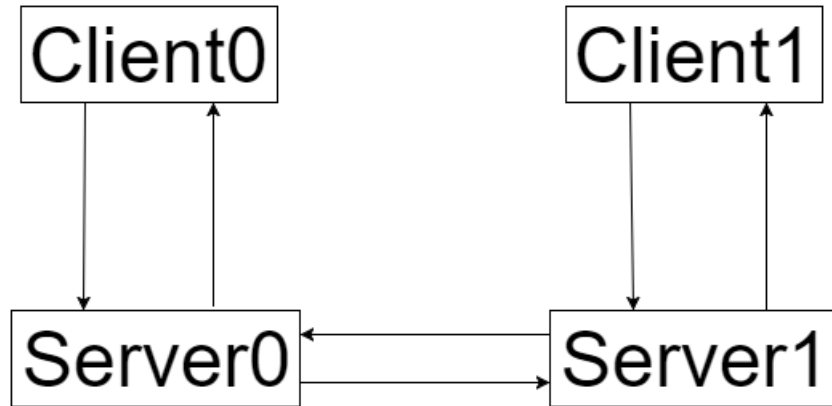
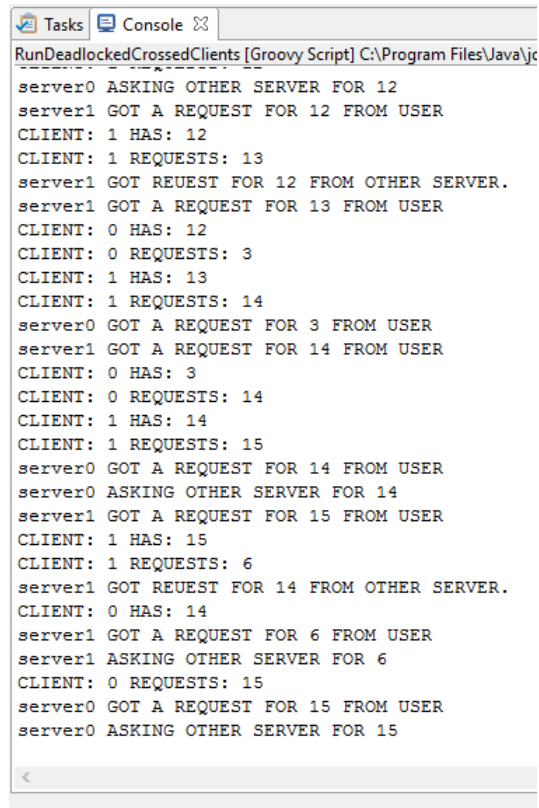


Figure 21: System diagram for exercise 7-1.



```
server0 ASKING OTHER SERVER FOR 12
server1 GOT A REQUEST FOR 12 FROM USER
CLIENT: 1 HAS: 12
CLIENT: 1 REQUESTS: 13
server1 GOT REUEST FOR 12 FROM OTHER SERVER.
server1 GOT A REQUEST FOR 13 FROM USER
CLIENT: 0 HAS: 12
CLIENT: 0 REQUESTS: 3
CLIENT: 1 HAS: 13
CLIENT: 1 REQUESTS: 14
server0 GOT A REQUEST FOR 3 FROM USER
server1 GOT A REQUEST FOR 14 FROM USER
CLIENT: 0 HAS: 3
CLIENT: 0 REQUESTS: 14
CLIENT: 1 HAS: 14
CLIENT: 1 REQUESTS: 15
server0 GOT A REQUEST FOR 14 FROM USER
server0 ASKING OTHER SERVER FOR 14
server1 GOT A REQUEST FOR 15 FROM USER
CLIENT: 1 HAS: 15
CLIENT: 1 REQUESTS: 6
server1 GOT REUEST FOR 14 FROM OTHER SERVER.
CLIENT: 0 HAS: 14
server1 GOT A REQUEST FOR 6 FROM USER
server1 ASKING OTHER SERVER FOR 6
CLIENT: 0 REQUESTS: 15
server0 GOT A REQUEST FOR 15 FROM USER
server0 ASKING OTHER SERVER FOR 15
```

Figure 22: System output for exercise 7-1.

12 Exercise 8.1

12.1 Client

```
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: $clientNumber REQUESTS: $key"
        requestChannel.write(key)
        def v = receiveChannel.read()
        println "CLIENT: $clientNumber HAS: $key"
```

```

def testVal = key*10

if(testVal == v) {
    println("KEY $clientNumber READ $v")
    println "TEST SUCCESS"
}
else {
    println("KEY $clientNumber READ $v")
    println "TEST FAILED"
}
}

println "CLIENT: $clientNumber FINISHED."
}

```

```

CLIENT: 1 HAS: 18
KEY 1 READ 180
TEST SUCCESS
CLIENT: 1 REQUESTS: 19
CLIENT: 0 HAS: 18
KEY 0 READ 180
TEST SUCCESS
CLIENT: 0 REQUESTS: 9
servernull GOT A REQUEST FOR 19 FROM USER
servernull GOT A REQUEST FOR 9 FROM USER
CLIENT: 1 HAS: 19
KEY 1 READ 190
TEST SUCCESS
CLIENT: 1 REQUESTS: 20
CLIENT: 0 HAS: 9
KEY 0 READ 90
TEST SUCCESS
CLIENT: 0 REQUESTS: 10
servernull GOT A REQUEST FOR 20 FROM USER
servernull GOT A REQUEST FOR 10 FROM USER
CLIENT: 1 HAS: 20
KEY 1 READ 200
TEST SUCCESS
CLIENT: 1 FINISHED.
CLIENT: 0 HAS: 10
KEY 0 READ 100
TEST SUCCESS
CLIENT: 0 FINISHED.

```

Figure 23: Output for exercise 8-1.

13 Exercise 9.1

13.1 MissCheck

```
class MissCheck implements CSPProcess {

    def ChannelInput inChannel
    def ChannelOutput outChannel
    int preVal = 99

    void run() {
        while (true)
        {
            def inStream = inChannel.read()
            outChannel.write(inStream)

            def missed = ( inStream.data - preVal) - 1
            if(missed != inStream.missed ){
                println "EVENT W/ DATA: $inStream.data MISSED:
                        $inStream.missed CORRECTION: $missed"
            }
            preVal = inStream.data
        }
    }
}
```

14 Exercise 9.2

Q: What do you conclude?

Ans: When the program is first run, for the most part the first few events pass through without being missed. As seen in the figure below (Fig:24) during the first few cycles the limited buffers will be empty. If the buffers were increased in size this would reduce the amount of missed events.

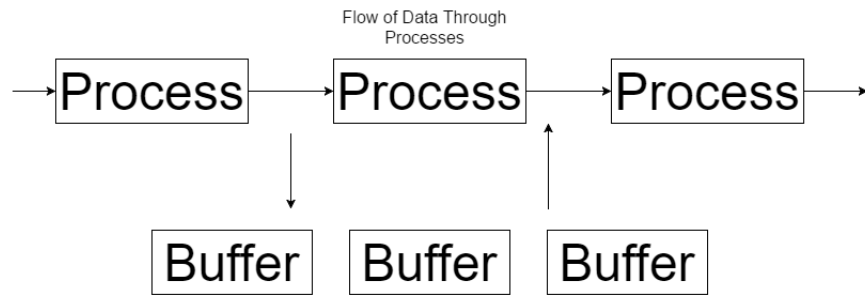


Figure 24: An abstract diagram to help illustrate the point.

15 Exercise 9.3

Fairmultiplex: This method chooses a channel based on what priority it is set to. It works similarly to a queuing system, after a channel has been selected it is then allocated a low priority during the next selection process. This gives consistent output.

Primultiplex: This method chooses a channel based on which channel is ready first. This results in the prioritization of streams with the a lower index i.e stream 1. This may result in any events in streams with a higher index not being read.

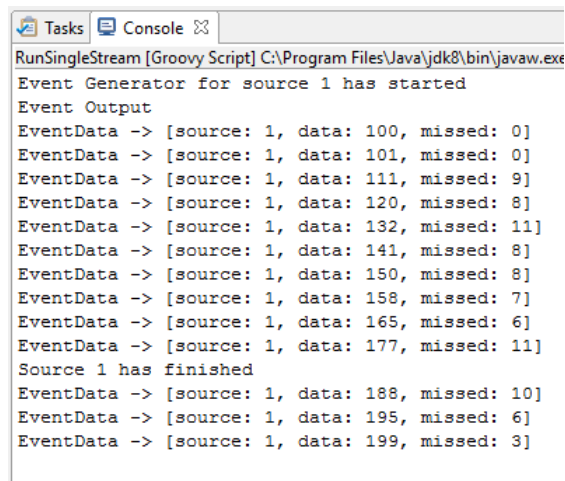
Multiplexer: This method randomly chooses between the different channels that are currently ready. As a result this method produce the most inconsistent output.

```
RunSingleStream [Groovy Script] C:\Program Files\Java\jdk8\bin\javaw.exe
Event Output
Event Generator for source 1 has started
EventData -> [source: 1, data: 100, missed: 0]
EventData -> [source: 1, data: 101, missed: 0]
EventData -> [source: 1, data: 108, missed: 6]
EventData -> [source: 1, data: 119, missed: 10]
EventData -> [source: 1, data: 128, missed: 8]
EventData -> [source: 1, data: 140, missed: 11]
EventData -> [source: 1, data: 155, missed: 14]
EventData -> [source: 1, data: 164, missed: 8]
Source 1 has finished
EventData -> [source: 1, data: 177, missed: 12]
EventData -> [source: 1, data: 190, missed: 12]
EventData -> [source: 1, data: 199, missed: 8]
```

Figure 25: Results using Fairmultiplex.

```
RunSingleStream [Groovy Script] C:\Program Files\Java\jdk8\bin\javaw.exe
Event Generator for source 1 has started
Event Output
EventData -> [source: 1, data: 100, missed: 0]
EventData -> [source: 1, data: 101, missed: 0]
EventData -> [source: 1, data: 106, missed: 4]
EventData -> [source: 1, data: 119, missed: 12]
EventData -> [source: 1, data: 132, missed: 12]
EventData -> [source: 1, data: 141, missed: 8]
EventData -> [source: 1, data: 154, missed: 12]
EventData -> [source: 1, data: 165, missed: 10]
EventData -> [source: 1, data: 173, missed: 7]
Source 1 has finished
EventData -> [source: 1, data: 184, missed: 10]
EventData -> [source: 1, data: 192, missed: 7]
EventData -> [source: 1, data: 199, missed: 6]
```

Figure 26: Results using Primultiplex.

A screenshot of a console window titled 'RunSingleStream [Groovy Script] C:\Program Files\Java\jdk8\bin\javaw.exe'. The console output shows the following:

```
Event Generator for source 1 has started
Event Output
EventData -> [source: 1, data: 100, missed: 0]
EventData -> [source: 1, data: 101, missed: 0]
EventData -> [source: 1, data: 111, missed: 9]
EventData -> [source: 1, data: 120, missed: 8]
EventData -> [source: 1, data: 132, missed: 11]
EventData -> [source: 1, data: 141, missed: 8]
EventData -> [source: 1, data: 150, missed: 8]
EventData -> [source: 1, data: 158, missed: 7]
EventData -> [source: 1, data: 165, missed: 6]
EventData -> [source: 1, data: 177, missed: 11]
Source 1 has finished
EventData -> [source: 1, data: 188, missed: 10]
EventData -> [source: 1, data: 195, missed: 6]
EventData -> [source: 1, data: 199, missed: 3]
```

Figure 27: Results using Multiplexer.

16 Exercise 11.3

17 Requirement 2

After looking at the code it is apparent there is an abundance of nested while loops and if statements, this makes the code relatively clunky and could be made more readable. When looking at how the system is structured and after running it, it is apparent that the system actually deadlocks. The deadlock appears to occur between the matcher and mouse buffer and this problem could probably be solved by introducing a queue process which acts as an intermediate buffer and would regulate the flow of data.