**Exercise 2-1**

**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() )

]

**Multiplier.groovy**

// write i \* factor to outChannel

outChannel.write (i \* factor)

// read in the next value of i

i = inChannel.read()

**Consumer.groovy**

//insert a modified println statement

println "Next integer multiplied by 4 is: ${i}"

i = inChannel.read()

**Output**

next: 2

next: Next integer multiplied by 4 is: 8

3

next: Next integer multiplied by 4 is: 12

4

next: Next integer multiplied by 4 is: 16

5

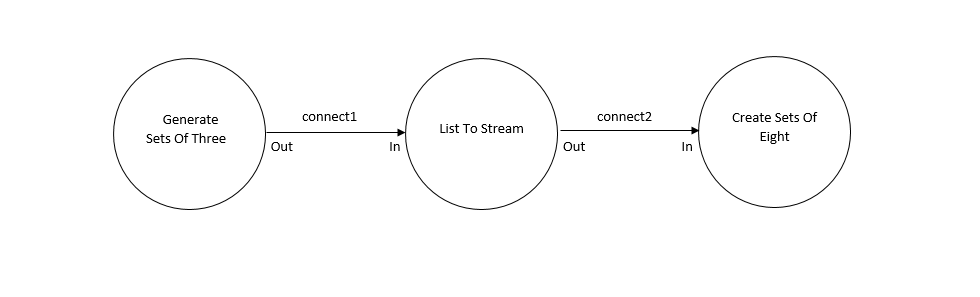
next: Next integer multiplied by 4 is: 20

2

next: Next integer multiplied by 4 is: 8

0

Finished

**Exercise 2-2**

**GenerateSetsOfThree.groovy**

//write the terminating List as per exercise definition

outChannel.write([-1,-1,-1])

**ListToStream.groovy**

// hint: output list elements as single integers

**for**(i **in** 0 ..< inList.size){

outChannel.write(inList[i])

}

inList = inChannel.read()

**CreateSetsOfEight.groovy**

// put v into outList and read next input

outList[i] = v

v = inChannel.read()

**Output**

Eight Object is [1, 2, 3, 4, 5, 6, 7, 8]

Eight Object is [9, 10, 11, 12, 13, 14, 15, 16]

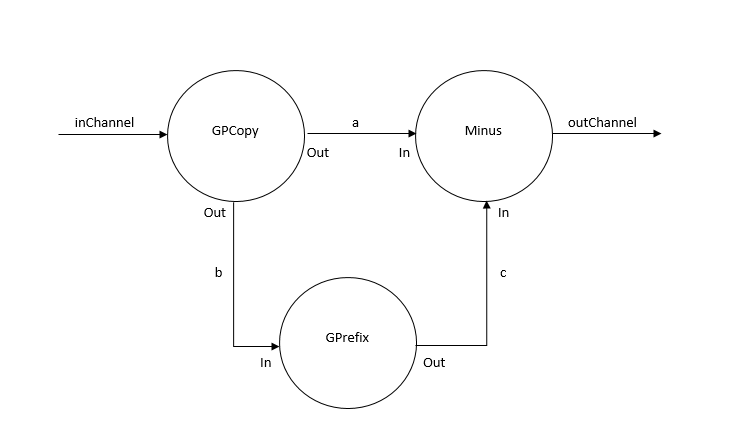
Eight Object is [17, 18, 19, 20, 21, 22, 23, 24]

Finished

**Questions**

1. To output 6 integers, change the line **for(i in 0 ..7)** to **for(i in 0 ..5)** in CreateSetsOfEight.groovy
2. Create a variable **size** that can be adjusted to any number and then pass it to the process, which will decide the size of each list generated.
3. The process will just get stuck, it will eventually read the termination number of -1 but then will have no number to read and will be stuck.

**Exercise 3-1**

**Differentiate**

**Minus.groovy**

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)

**Differentiate.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 Minus

**new** Minus ( inChannel0: a.in(),

inChannel1: c.in(),

outChannel: outChannel )

]

**Output - Minus**

Differentiated Numbers

0

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

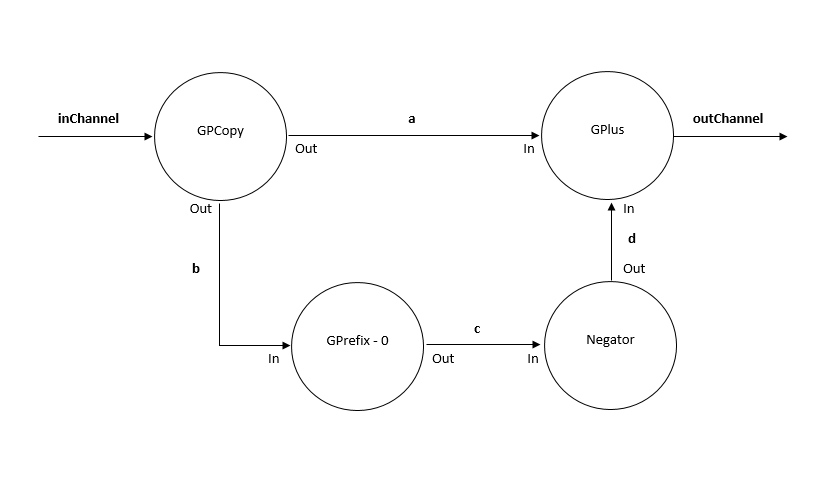
16

17

18

19

20

**DifferentiateNeg**

**Negator.groovy**

//output the negative of the input value

outChannel.write(-inChannel.read())

**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 - negator**

Differentiated Numbers - Negator

0

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

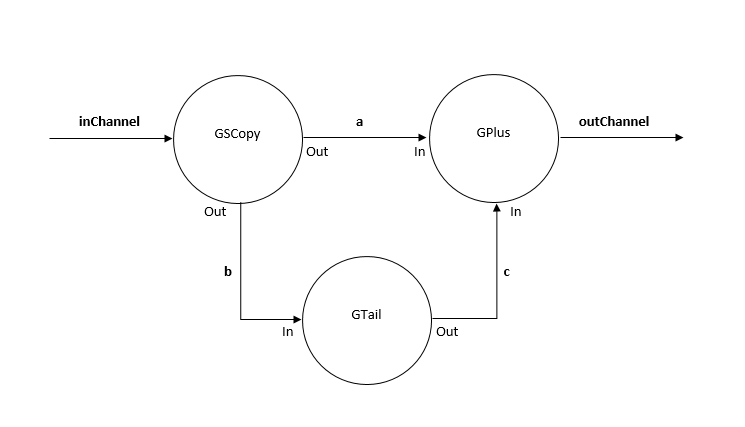
18

19

20

**Questions**

I believe the Negator approach is much better, as it lets us reuse the GPlus process just by building a very simple processes to connect to GPlus to make it work as minus. It is also much easier to connect to other processes as you do not have to worry about which channel connects to which to come out with the accurate answer. If you used the Minus version, then you have to make sure the two inputs are in correct order since subtraction is not cumulative.

**Exercise 3-2**

**GSCopy.groovy**

**def** i = inChannel.read()

// output the input value in sequence to each output channel

outChannel0.write(i)

outChannel1.write(i)

**GSquares.groovy Version A**

**def** testList = [ **new** GNumbers ( outChannel: N2I.out() ),

**new** GIntegrate ( inChannel: N2I.in(),

outChannel: I2P.out() ),

// you will need to modify this twice

//first modification is to insert a constructor for GSPairsA

// then run the network using TestGSCopy

**new** GSPairsA ( inChannel: I2P.in(),

outChannel: outChannel)

//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 to

// explain why this happens!

]

**Output**

Squares

**GSquares.groovy Version B**

**def** testList = [ **new** GNumbers ( outChannel: N2I.out() ),

**new** GIntegrate ( inChannel: N2I.in(),

outChannel: I2P.out() ),

// you will need to modify this twice

//first modification is to insert a constructor for GSPairsA

// then run the network using TestGSCopy

**new** GSPairsB ( inChannel: I2P.in(),

outChannel: outChannel )

//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 to

// explain why this happens!

]

**Output**

Squares

1

4

9

16

25

36

49

64

81

100

**Questions**

GSPairsA has returned no output, whereas GSPairsB has returned the output for square numbers correctly. I believe that this is the case because when GSPairsA runs, the first value is sent to be read in by GPlus and the next value is then sent to GTail which ignores the first value sent to it. This creates a problem since now it is time for another value to be sent to GPlus through the **a** channel, but a value is already occupying that spot so the program enters a deadlock and cannot continue since GPlus requires 2 values and GTail has not sent any value through channel **c**. All of this is due to the fact that the GSCopy processes runs sequentially. GSPairsB manages to work since it sends its first value to GTail which ignores it and then the processes can continue smoothly.

**Exercise 3-3**

**Questions**

GPrint prints the output in a not ordered, non-tabular way which looks really messy, whereas GParPrint will print each processes at the correct stage in a neat, order manner. Therefore, it is much easier to just build this new process that is far more dynamic and pleasant in its printing manner.

**Exercise 4-1**

**ResetPrefix.groovy**

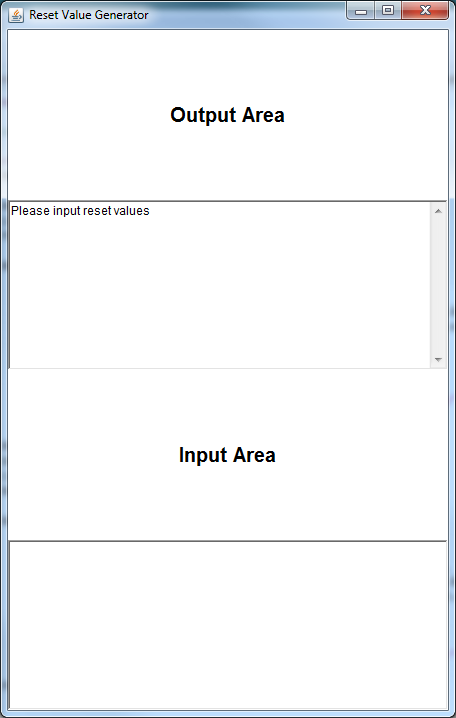
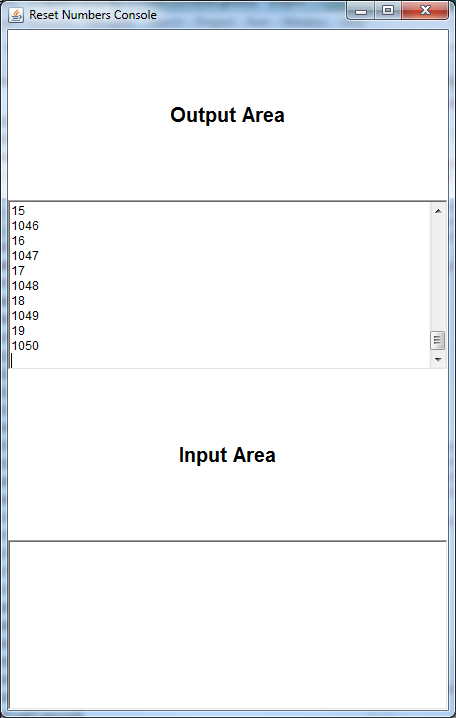
**if** (index == 0 ) { // resetChannel input

**def** resetValue = resetChannel.read()

//inChannel.read()

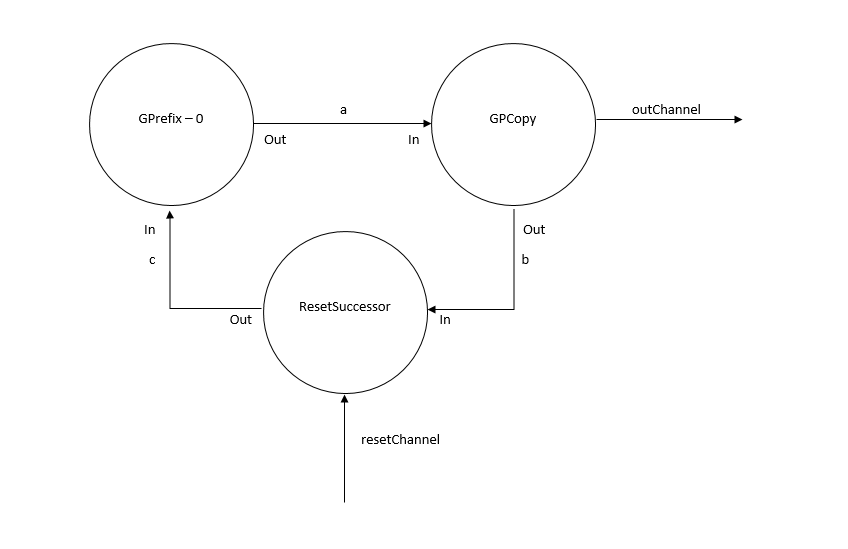
outChannel.write(resetValue)

}

**Output**

**Questions**

Removing the **inChannel.read()** line will make it so the original value will not be passed into ResetPrefix, so it will keep GSuccessor process busy, after the reset value has been sent out to GCopy, GSuccessor can finally send out the original number to ResetPrefix which will then continue circulating alongside the reset value. The system now has 2 numbers circulating about it. Adding a third value to the system will cause it to deadlock as the other 2 processes will be busy and will not be able to read in any values and the only ready process ResetPrefix will now also be busy due to a new number being passed in.

**Exercise 4-2**

**ResetSuccessor.groovy**

// deal with inputs from resteChannel and inChannel

// use a priSelect

**def** index = alt.priSelect()

**if** (index == 0){

**def** resetValue = resetChannel.read()

inChannel.read()

outChannel.write(resetValue)

}

**else** {

**def** inputValue = inChannel.read()

outChannel.write(inputValue + 1)

}

**ResetNumbers.groovy**

**def** testList = [ **new** GPrefix ( prefixValue: initialValue,

outChannel: a.out(),

inChannel: c.**in**() ),

**new** GPCopy ( inChannel: a.**in**(),

outChannel0: outChannel,

outChannel1: b.out() ),

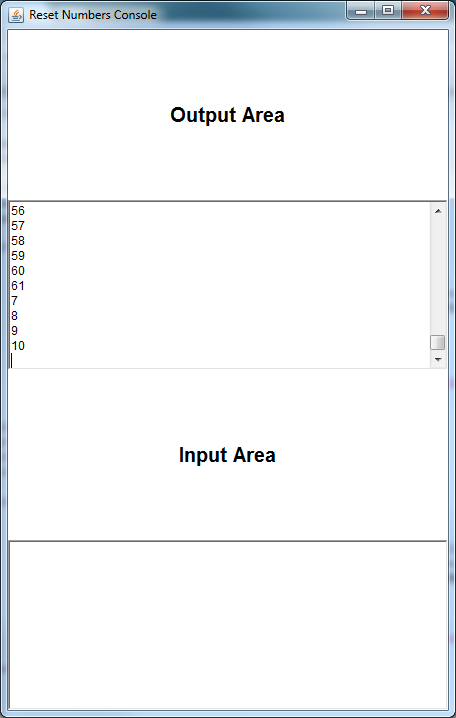
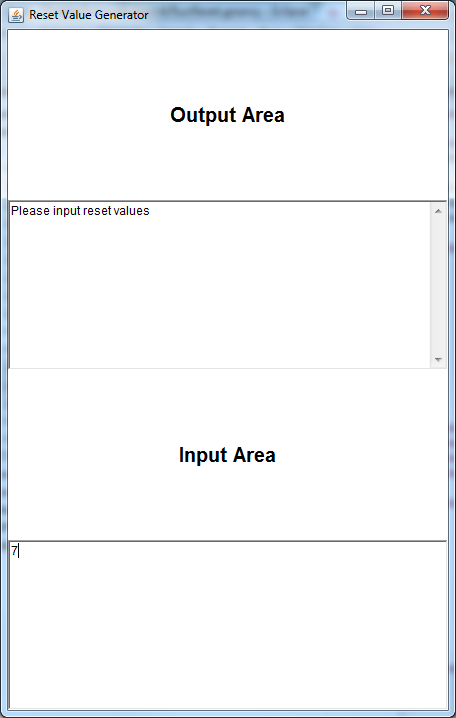
// requires a constructor for ResetSuccessor

**new** ResetSuccessor ( inChannel: b.**in**(),

outChannel: c.out(),

resetChannel: resetChannel )

]

**Output**

**Questions**

Changing the position of resetChannel did not fix the issue encountered in the previous exercise, the system deadlocks after 2 reset values have been added in. I believe that is because we haven’t really added a precaution in the processes themselves and just changing the position of resetChannel will not fix an issue of having too many numbers circulating in the system. This system behaves just like the previous one except for what the output of the system will look like since the resetValue you enter will be changed in the output due to it being sent to the GSuccessor process.

**Exercise 5-1**

|  |  |  |
| --- | --- | --- |
| Test Number | What was changed | Result |
| 1. | Delay of 5 added to QProducer | The output was a tiny bit slower than original but it has returned the correct output |
| 2. | Delay of 5 added to QConsumer | The output was a tiny bit slower than original but it has returned the correct output |
| 3. | Delay of 10 added to QProducer | The output was a tiny bit slower than original but it has returned the correct output |
| 4. | Delay of 10 added to QConsumer | The output was a tiny bit slower than original but it has returned the correct output |

**Questions**

In conclusion, changing the delay does not affect the validity of the system, all it affects is the amount of time taken to print the output. The pre-conditions make sure that even if QConsumer is way slower, it will not cause any problems with the system.

**Exercise 5-2**

**Scale.groovy**

**switch** ( scaleAlt.priSelect(preCon) ) {

**case** SUSPEND :

// deal with suspend input

suspend.read()

factor.write(scaling)

suspended = **true**

*println* "Suspended"

**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 )

**break**

**case** TIMER:

// deal with Timer input

timeout = timer.read() + DOUBLE\_INTERVAL

timer.setAlarm( timeout )

scaling = scaling \* multiplier

*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

**if**(suspended) result.scaled = inValue

outChannel.write( result )

**break**

} //end-switch

preCon[TIMER] = (!suspended) //If not suspended, timer is available

preCon[INJECT] = suspended //If suspended, injector is available

**Output**

Original Scaled

0 0

1 2

2 4

3 6

Normal Timer: new scaling is 4

4 16

5 20

Suspended

6 6

Injected scaling is 5

7 35

8 40

9 45

10 50

11 55

Normal Timer: new scaling is 10

12 120

Suspended

13 13

Injected scaling is 11

14 154

15 165

16 176

17 187

18 198

Normal Timer: new scaling is 22

19 418

Suspended

20 20

Injected scaling is 23

21 483

22 506

23 529

24 552

25 575

Normal Timer: new scaling is 46

26 1196

Suspended

27 27

Injected scaling is 47

28 1316

29 1363

**Questions**

Pre-conditions version of scale is much more elegant in my opinion. It is a much easier to understand process and it is much easier to expand its alternatives, you just have to add a new pre-condition and a switch case and work around the system. I personally do not like using nested loops or in this case nested alternatives that much because they do not look that good.

**Exercise 6-1**



**ListToStreamForTest.groovy**

// hint: output list elements as single integers

**for**(i **in** 0 ..< inList.size){

outChannel.write(inList[i])

//To be used for testing

testList = testList << inList[i]

}

**CreateSetsOfEightTest.groovy**

**class** CreateSetsOfEightTest **extends** GroovyTestCase {

**void** testSetsOfEight(){

One2OneChannel connect1 = Channel.*createOne2One*()

One2OneChannel connect2 = Channel.*createOne2One*()

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

**def** ListToStreamForTest = **new** ListToStreamForTest ( inChannel: connect1.in(), outChannel: connect2.out() )

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

**def** processList = [ GenerateSetsOfThree, ListToStreamForTest, CreateSetsOfEight ]

**new** PAR (processList).run()

// Expected is just the last 8 values of the list

**def** expected = ListToStreamForTest.testList.subList( ListToStreamForTest.testList.size - 8,

ListToStreamForTest.testList.size )

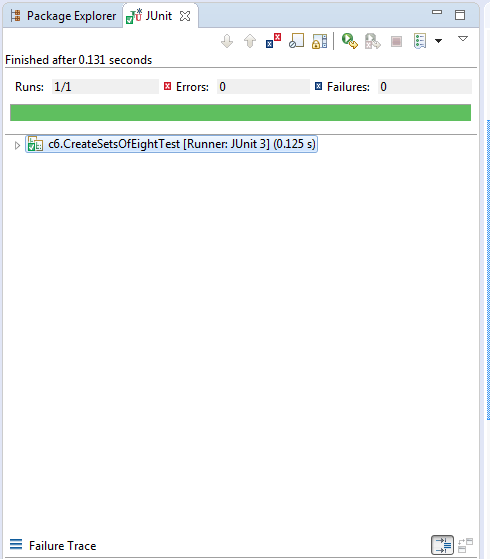
**def** actual = CreateSetsOfEight.outList

*assertTrue*( expected == actual)

}

}

**Test Output**



**Exercise 7-1**

**Server.groovy**

**switch** (index) {

**case** CLIENT :

**def** key = clientRequest.read()

println "Server $serverNumber received a request from client $serverNumber for value at location $key"

**if** ( dataMap.containsKey(key) ){

clientSend.write(dataMap[key])

println "Server $serverNumber is sending value at location $key to client $serverNumber"

}

**else**{

println "Server $serverNumber is requesting a value from the other server at location $key"

thisServerRequest.write(key)

}

//end if

**break**

**case** OTHER\_REQUEST :

**def** key = otherServerRequest.read()

println "Other server is requesting a value from server $serverNumber at location $key"

**if** ( dataMap.containsKey(key) )

otherServerSend.write(dataMap[key])

**else**

otherServerSend.write(-1)

//end if

**break**

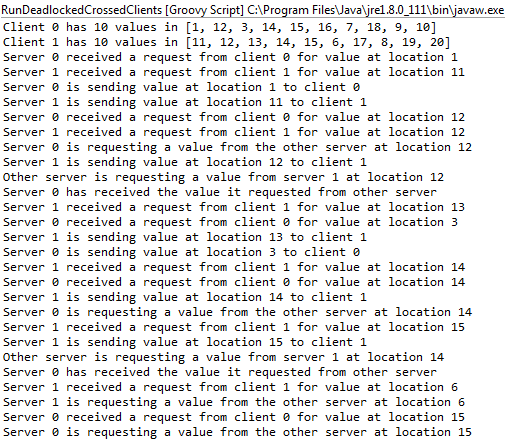
**case** THIS\_RECEIVE :

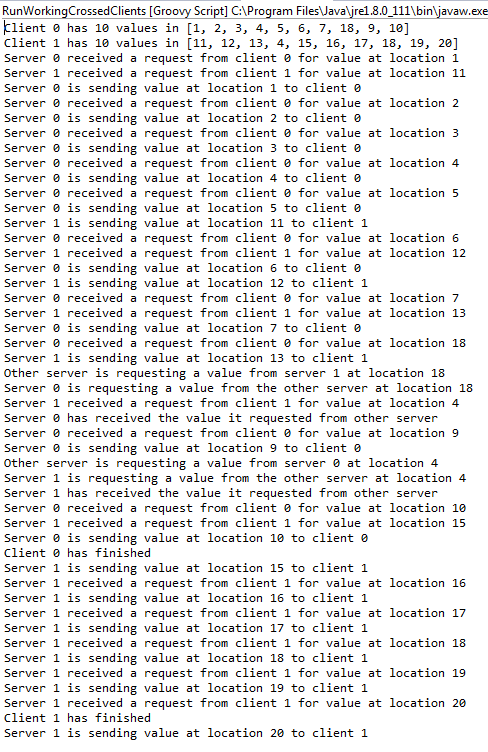
clientSend.write(thisServerReceive.read() )

println "Server $serverNumber has received the value it requested from other server"

**break**

} // end switch

 **Working Output Deadlock Output**



**Questions**

If you look at the last 4 lines of the deadlock output, you can see that both servers received a value from their clients that is outside their data map. This means that server 0 will request a value from server 1 and server 1 will request a value from server 0 at the same time. This is just like the previous example of deadlock in this chapter, server 1 is writing a value to server 0 and server 0 is writing a value to server 1, but none of them are reading the values in so they enter a state of deadlock. The working server client version works because it has less chances for both servers to request a value at the same time.

**Exercise 8-1**

**Client.groovy**

**for** ( i **in** 0 ..< iterations) {

**def** key = selectList[i]

**def** expected = key \* 10

**def** test = "WRONG"

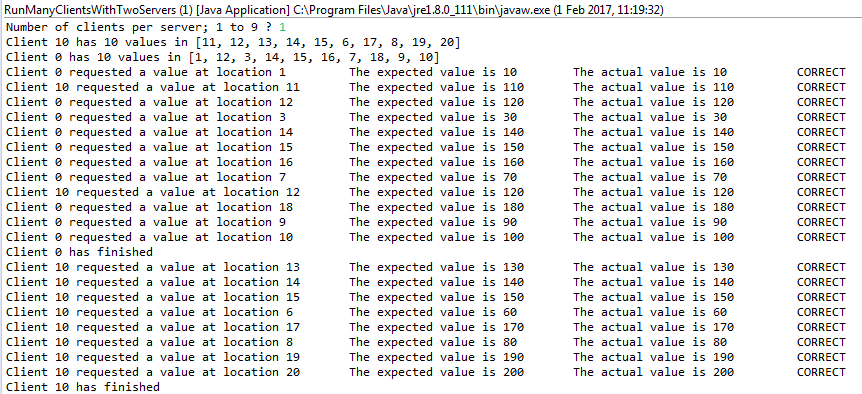
requestChannel.write(key)

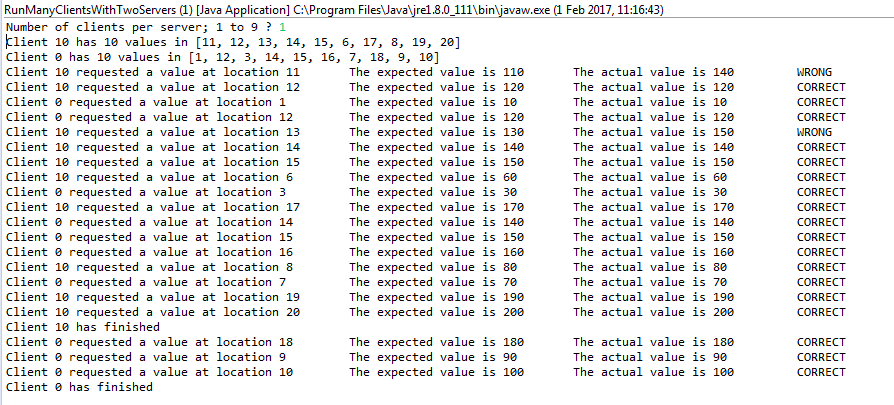
**def** v = receiveChannel.read()

**if** (expected == v) test = "CORRECT"

println "Client $clientNumber requested a value at location $key \t The expected value is $expected \t The actual value is $v \t $test"

}

**Output with default values, shows the test is correct**

**Output with wrong values put in to test if system is correct**

**Exercise 9-1**

E:\All stuff\Downloads\Untitled Diagram (2).png **EventHandler**

**EventHandler.groovy**

**def** handlerList = [ **new** EventReceiver ( eventIn: inChannel,

eventOut: toBuffer.out()),

**new** EventOWBuffer ( inChannel: toBuffer.in(),

getChannel: get.in(),

outChannel: transfer.out() ),

**new** EventPrompter ( inChannel: transfer.in(),

getChannel: get.out(),

outChannel: toTest.out() ),

//Added new process here, so the test is done at the very end of the handler network

**new** EventMissedTest ( inChannel: toTest.in(), outChannel: outChannel )

]

**EventData.groovy**

Line 15 -- **def** test = "Incorrect"

Lines 18-21 -- **def** e = **new** EventData ( source: **this**.source,

data: **this**.data,

missed: **this**.missed,

test: **this**.test )

Line 30 -- s = s + " -- Testing missed data: " + test

**EventMissedTest.groovy**

**class** EventMissedTest **implements** CSProcess {

**def** ChannelInput inChannel

**def** ChannelOutput outChannel

**public** **void** run() {

**def** e = inChannel.read().copy()

//This is a real hacky way of creating previous data for the first value in

**def** previous = e.data - 1

**while**(**true**){

//Reversed the formula given in book to find missed

**def** test = (e.data - previous) - 1

**if**(test == e.missed) e.test = "Correct!"

previous = e.data

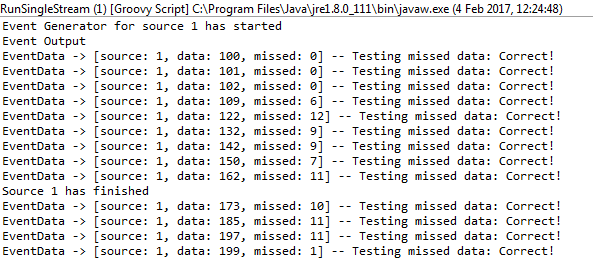
outChannel.write(e)

e = inChannel.read().copy()

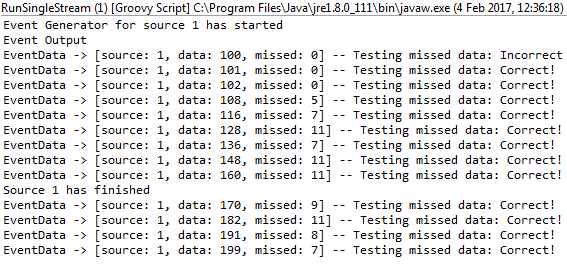
}

}

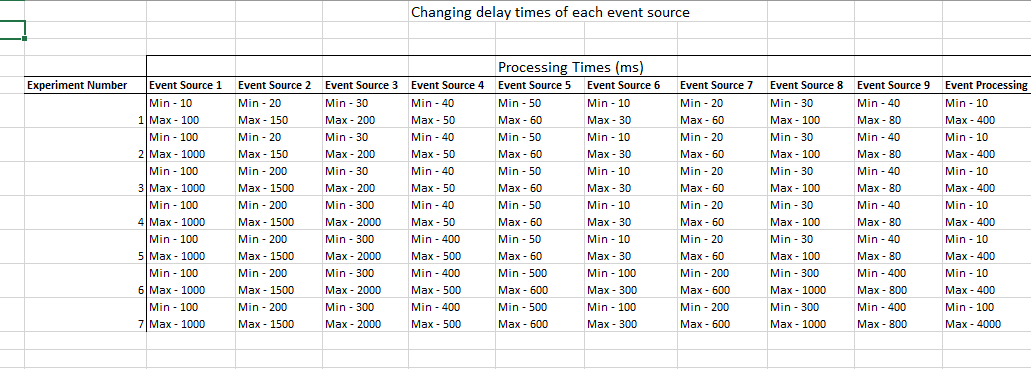
}

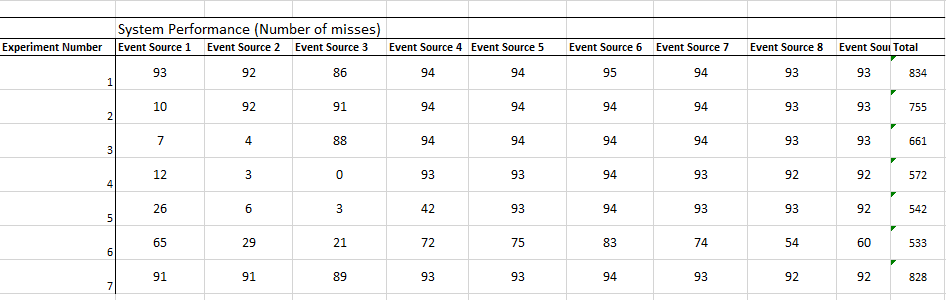
**Correct Output**

**Output with first event being wrong**



**Exercise 9-2 (**[**Click here to open excel spreadsheet**](Exercise%209-2%20and%209-3%20Tests.xlsx)**)**

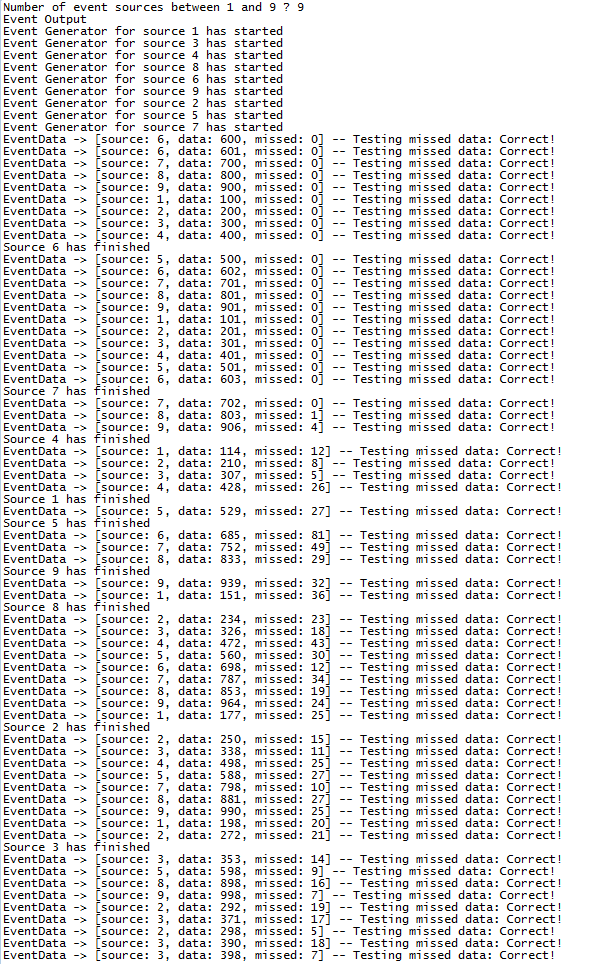




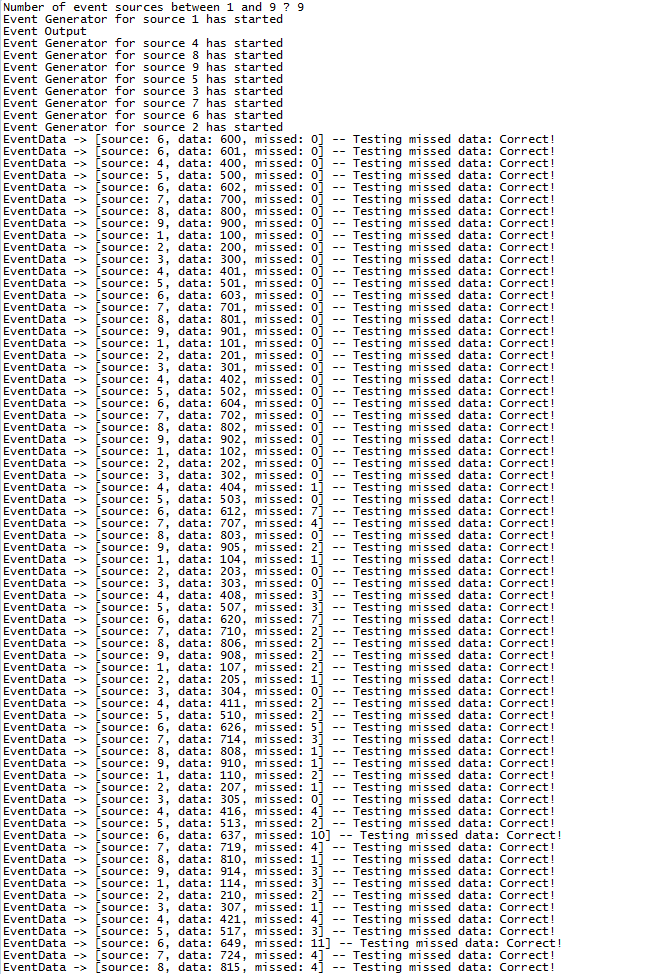
E:\All stuff\Downloads\Untitled Diagram (2).png**Event Handler (From previous exercise)**

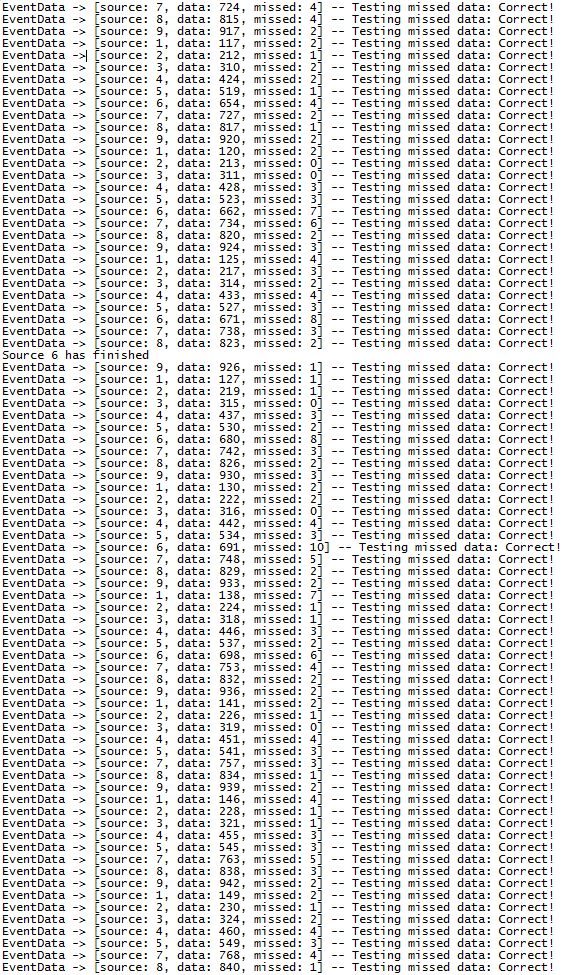
**Output**

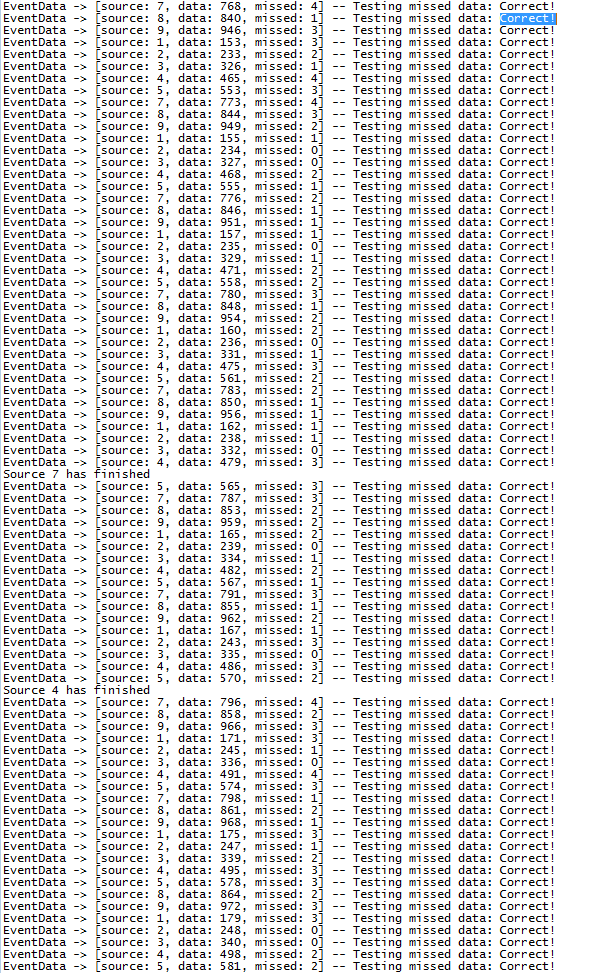
Experiment 1 (Default delay)



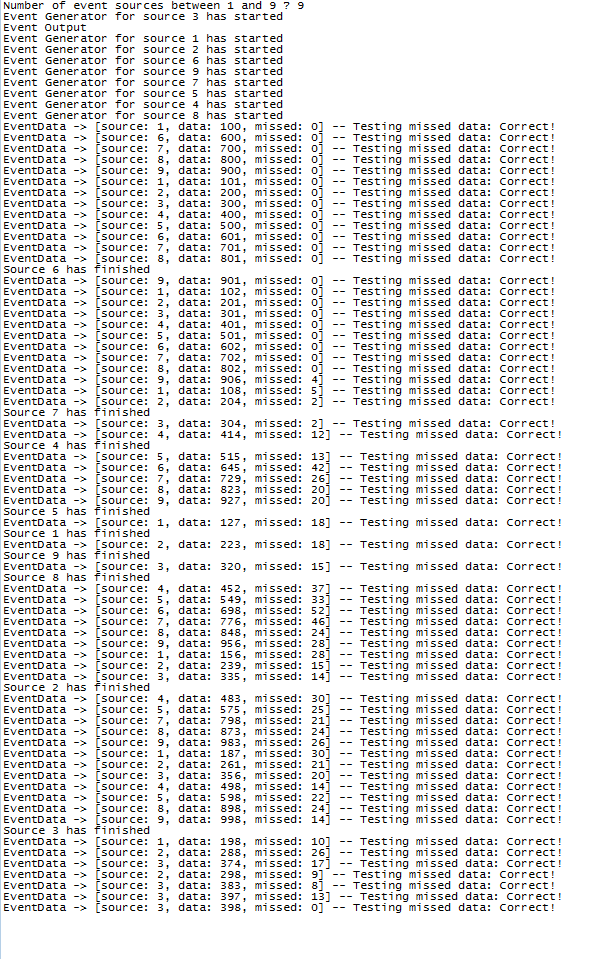
Experiment 6 (Delay \* 10 for all sources)

Part 1

Part 2

Part 3

Experiment 7(Delay \* 10 for all event sources and for event processing)



**Questions**

As can be seen in experiment results, the default delay that has been put has a total of 834 missed events, this is a huge number of events that has been missed but this can probably be blamed on the fact that the buffer used is rather small, so of course it will miss large amounts of data when 9 event sources are trying to write to it concurrently.

As the delay is increased on some of the events, not all, then we can see a decrease in events missed, this is because less events will be coming in to the buffer at one time, since now some events will be staying back due to a much larger delay they have been given. This allowed the number of events to decrease down to 533. It seems that increasing the delay does improve the amount of data saved, however the system takes longer to finish.

Another thing I noticed is that if you increase the delay by 10 times to all 9 event sources, the total number of missed events is small, but individual event sources will still have a rather large number of missed events but that may be because the difference between event source delays is the same as in Experiment 1 which has the default delays in place, just increased tenfold.

The final experiment is the same as experiment 6, however the processing time has been increased tenfold. This has returned the total value of missed events to that of the first experiment and that may be because even though the delays have improved the number of missed events, it does not have much of an effect if the processing time is also increased to a big amount since the events will still have to wait for processing to be done so the buffer will still get a lot of events at the same time.

In conclusion, it seems that if increase the delays placed on event sources, you can get an increase in performance as the number of missed events is much smaller, however the price is that the process will run for a much longer time. Also, increasing processing time does not help the system that much.

**Exercise 9-3**

**Test data**

**EventProcessing.groovy – FairMultiplex version**

**def** pList = [

**new** FairMultiplex ( inChannels: eventStreams,

outChannel: mux2udd.out() ),

**new** UniformlyDistributedDelay ( inChannel:mux2udd.**in**(),

outChannel: udd2prn.out(),

minTime: minTime,

maxTime: maxTime ),

**new** GPrint ( inChannel: udd2prn.**in**(),

heading : "Event Output",

delay: 0)

]

**EventProcessing.groovy – PriMultiplex version**

**def** pList = [

**new** PriMultiplex ( inChannels: eventStreams,

outChannel: mux2udd.out() ),

**new** UniformlyDistributedDelay ( inChannel:mux2udd.**in**(),

outChannel: udd2prn.out(),

minTime: minTime,

maxTime: maxTime ),

**new** GPrint ( inChannel: udd2prn.**in**(),

heading : "Event Output",

delay: 0)

]

**EventProcessing.groovy – Multiplexer version**

**def** pList = [

**new** Multiplexer ( inChannels: eventStreams,

outChannel: mux2udd.out() ),

**new** UniformlyDistributedDelay ( inChannel:mux2udd.**in**(),

outChannel: udd2prn.out(),

minTime: minTime,

maxTime: maxTime ),

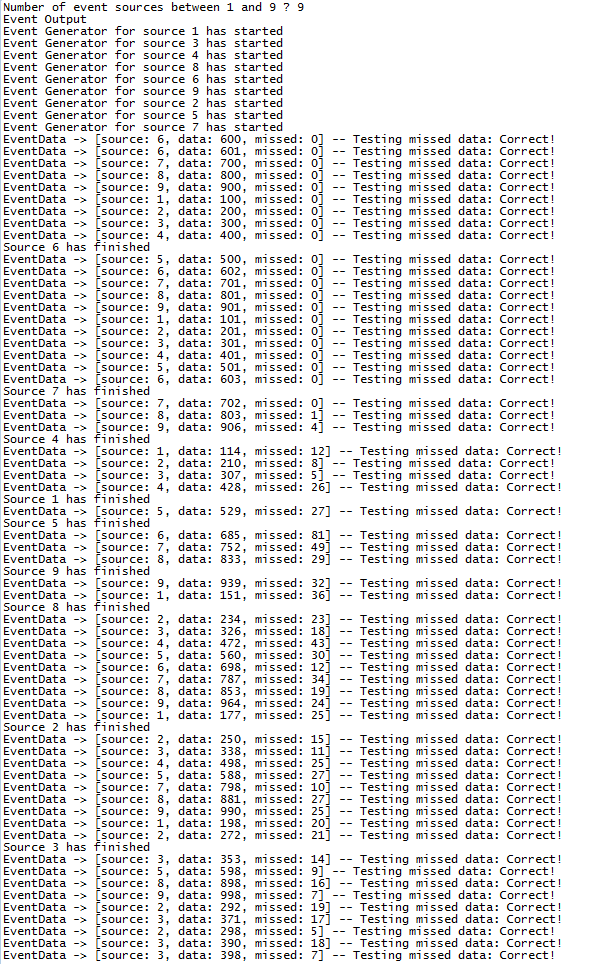
**new** GPrint ( inChannel: udd2prn.**in**(),

heading : "Event Output",

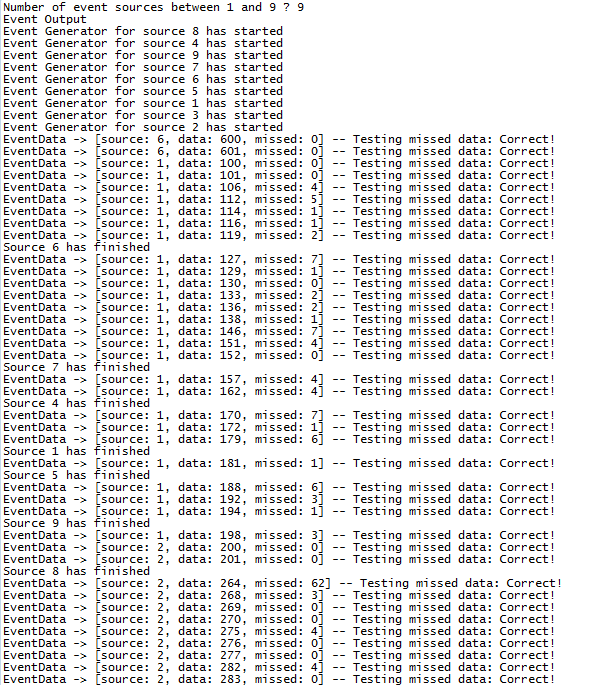
delay: 0)

]

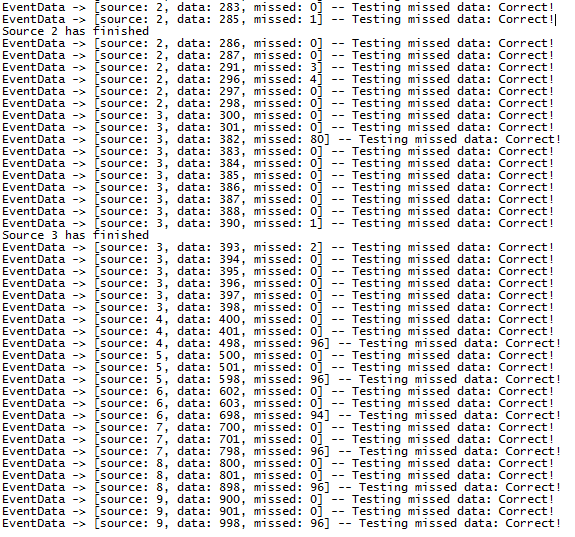
**Output**

Experiment 1(Fair)

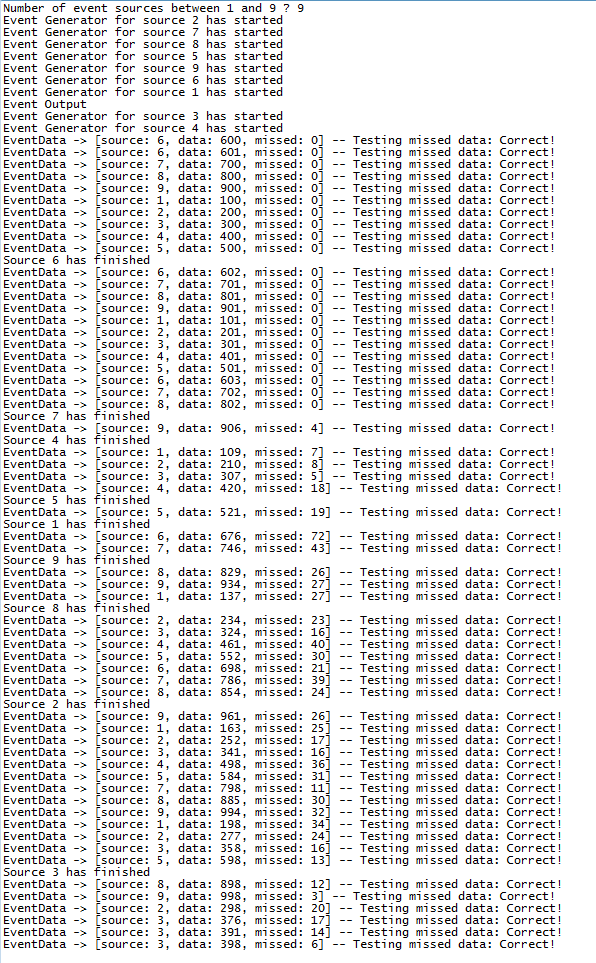
Experiment 2(Pri)

Part 1

Part 2



Experiment 3(Normal)



**Questions**

FairMultiplex selects events in a fair manner, so each event source will be given a fair amount of chances to output their data. This is the multiplexer used for majority of the tests in the previous exercise so the performance of the system is the base line for other multiplexers to be judged upon. Using this multiplexer resulted in 834 total misses which are spread out over the 9 events.

PriMultiplex selects based on the index of the event source. The highest priority will be given to the event sources closest to 0, which can be seen in the test data since the first 3 event sources had a better performance over the others. This did result in the total missed events to be reduced down to 811, which is a minor increase in performance.

Multiplexer selects whatever event source is ready. It has no bias and just whoever comes first gets in first. Interestingly, there is barely any difference between this and FairMultiplex, they behaved in the same way and if there is a difference it is incredibly minor and I have not spotted it. Using this multiplexer resulted in 832 total events missed, but that could just be due to external interference.

In conclusion, I believe changing the multiplexer doesn’t change the performance as much as changing the delay times for the event sources, but if the system is more built around the specific behaviours of the multiplexers, I’m sure the system would be much more efficient.