**Lab Book – Glenn Wilkie-Sullivan (40208762)**

***Exercise 2-1:***

(Multiplier Class)

// write i \* factor to outChannel

outChannel.write(i \* factor);

// read in the next value of i

i = inChannel.read();

(Consumer Class)

//insert a modified println statement

println ("New Value: " + i)

i = inChannel.read()

(RunMultiplier Class)

//insert here an instance of multiplier with a multiplication factor of 4

**new** Multiplier ( inChannel : connect1.in(),

outChannel : connect2.out() ),

Pipeline Diagram:



Output:



Network Operation Commentary:

Using this network is fairly trivial – upon running, the user can enter numbers into the console, and the program will return that number multiplied by a factor, ready for another input.

***Exercise 2-2:***

(GenerateSetsOfThree Class)

//write the terminating List as per exercise definition

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

(ListToStream Class)

// hint: output list elements as single integers

**for**(i **in** 0 ..< inList.size)outChannel.write(inList[i])

inList = inChannel.read()

(CreateSetsOfEight Class)

**for** ( i **in** 0 .. 7 ) {

// put v into outList and read next input

outList.add(v);

v = inChannel.read();

}

println " Eight Object is ${outList}"

outList.clear();

Pipeline Diagram:



Output:



**Question 1 (What change is required to output objects containing six integers?)**

Within our “CreateSetsOfEight” class, we read each element of the incoming tuples and add them to the outlist, which has the given range 0-7. Simply changing this to 0-5 generates list of six.

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

As before, we change the range parameter to whatever size of tuple we require.

**Question 3 (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) ?)**

If a new list of full size cannot be created, the remainder of integers are added to the previous list.

Network Operation Commentary:

Like before, using this network is fairly trivial. The user can create a large amount of tuples/containers for three integers on the backend of the program – upon running, the program will read in the sets of three integers until there is enough to make a set of eight integers, then print it to the console.

***Exercise 3-1:***

(Minus Class)

// output one value subtracted from the other

// be certain you know which way round you are doing the subtraction!!

outChannel.write(read1.value - read0.value)

(Negator Class)

//output the negative of the input value

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

outChannel.write(i \*= -1)

(Differentiate Class)

// insert a constructor for Minus

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

inChannel1: a.in(),

outChannel: outChannel)

(DifferentiateNeg Class)

//insert a constructor for Negator

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

outChannel: d.out()),

Pipeline Diagram (Minus):



Pipeline Diagram (Negator):



Output (Both Functions):



**Question 1 (Implement both approaches and test them. Which is the more pleasing solution? Why?)**

While both of the approaches seem very similar, and in operation they are identical, the Minus operation seems far more elegant and robust, as it uses ProcessReads instead of writing directly to the outChannel. It seems likely that the Minus operation is a lot less likely to crash or deadlock, and the process is very easily implementable.

Network Operation Commentary:

Running this network requires no external knowledge of the backend. It simply requires the user to start the network, and the program will handle the rest of the logic and output.

***Exercise 3-2:***

(GSCopy Class)

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

outChannel0.write(i)

outChannel1.write(i)

(GSquares Class)

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

outChannel: outChannel)

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

outChannel: outChannel)

Piepline Diagram (FOR GSPairsA & GSPairsB):



Output (FOR GSPairsB):



**Question 1 (Replace GPairsB with GPairsA and determine the effect of the change. Why does this happen?)**

GTail only outputs what is read into it. As channels a/b are switched, GPlus then expects an input of channels b/c, but is instead fed an input of a/c.

Network Operation Commentary:

Running this network requires no external knowledge of the backend. It simply requires the user to start the network, and the program will handle the rest of the logic and output.

***Exercise 3-3:***

**Question 1 (Why was it considered easier to build GParPrint as a new process rather than using multiple instances of GPrint to output a table of results?)**

GParPrint contains logic to do all the printing and formatting at once, which is considerably faster than instantiating two GPrints, which will have to be formatted into columns, spaced, etc. which takes a lot of time.

***Exercise 4-1:***

If one reset value is input, the program will use that value but doesn’t increment it. Thus, the reset value and prefix value are output side by side. However, if the user inputs multiple values, the program deadlocks and stops the output.

***Exercise 4-2:***

(ResetSuccessor Class)

// deal with inputs from resetChannel and inChannel

// use a priSelect

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

**if** (i.value == 0) {

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

inChannel.read()

outChannel.write(j + 1)

} **else** {

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

outChannel.write(k + 1)

}

(ResetNumbers Class)

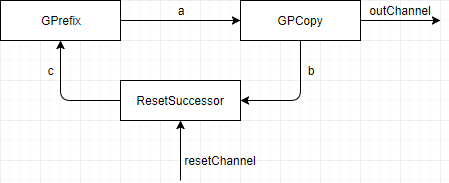
// requires a constructor for ResetSuccessor

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

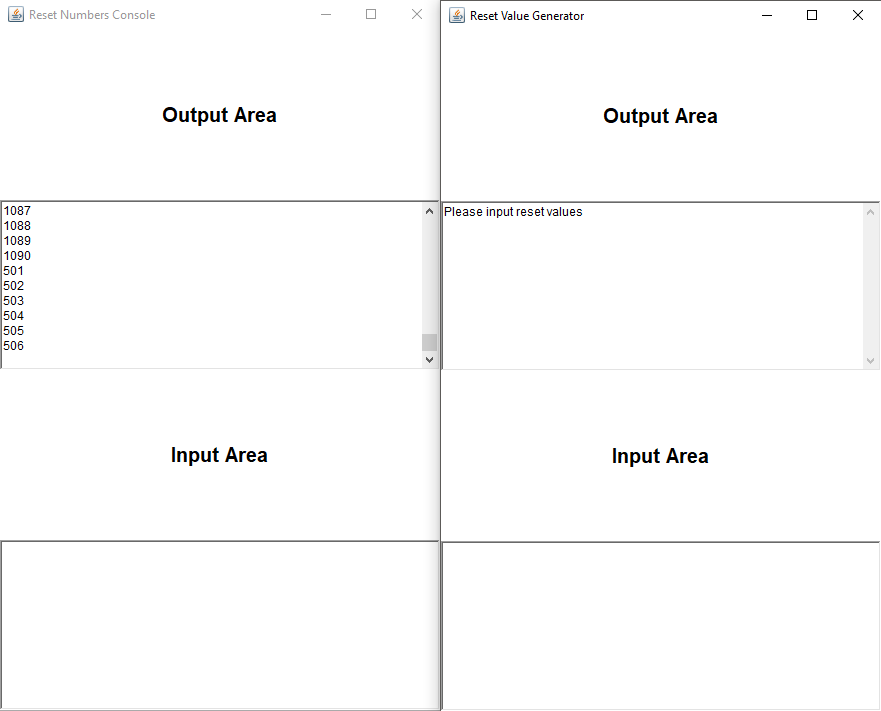
outChannel: c.out(),

resetChannel: resetChannel)

Pipeline Diagram:



Output:



**Associated Question (Does it overcome the problem identified in Exercise 1? If not, why not?)**

It doesn’t overcome the problem in exercise 1, as the functionality is almost identical except for the prefix functions are now in successor.

Network Operation Commentary:

Upon running the program, the user should be in control of two windows – one window to show the values being printed, and another window to control the reset of the values. On the control window, if the user enters a number into the input area and presses enter, the values on the output window should reset and start incrementing at whatever value was specified.

***Exercise 5-1:***

The delays do in fact work on the QProducer/Consumer process, as the program runs in variable time according to the value of the delay – the larger the delay, the longer the program takes to execute. It appears only one delay needs to be modified for the delay to work – as expected. This means there can either be a delay from values being written, or a delay from values being read. Upon further examination, I believe the delay is scaled from 1000 to 1. This means for each 1000 units added to the delay value, the producer/consumer will take an extra second to process a read or write signal. It also appears there is no limit on the delay – I tested from a 0 to 500,000,000-unit delay, but the program still ran (may be delimited by the size of primitive variables).

***Exercise 5-2:***

(Scale Class)

**case** SUSPEND:

// deal with Suspend input

preCon[SUSPEND] = **false**

preCon[INJECT] = **true**

preCon[INPUT] = **false**

suspend.read()

factor.write(scaling)

suspended = **true**

println "Stream Suspended"

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

**def** result = **new** ScaledData()

result.original = inValue

result.scaled = inValue

outChannel.write(result)

**break**

**case** INJECT:

// deal with Inject input

preCon[SUSPEND] = **true**

preCon[INJECT] = **false**

preCon[INPUT] = **true**

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 \* 2

println "Normal Timer: new scaling is ${scaling}"

**break**

**case** INPUT:

// deal with Input channel

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

**def** result = **new** ScaledData()

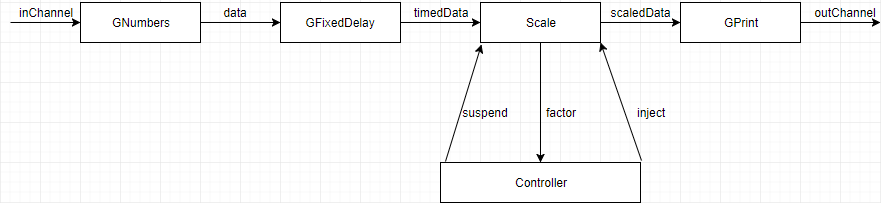
result.original = inValue

result.scaled = inValue \* scaling

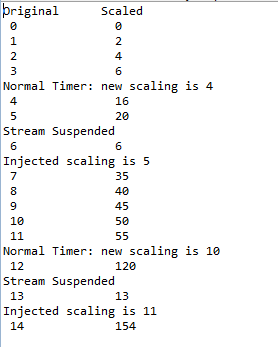
outChannel.write(result)

**break**

Pipeline Diagram:



Output:



**Associated Question (Which is the more elegant formulation? Why?)**

The precondition formulation is naturally more robust, or elegant, as it specifies which behaviours can be dealt with at which specific times during the course of the program running. It stops any unexpected input from happening, which could break the first case-statement formulation if the input is not dealt with correctly. The second formulation also allows more control over the variables effecting the eventual output, as there is more insight into the suspension and injection of the values.

Network Operation Commentary:

Running this network requires no external knowledge of the backend. It simply requires the user to start the network, and the program will handle the rest of the logic and output.

***Exercise 6-1:***

(Amended ListToStream Class)

**def** ChannelInput inChannel

**def** ChannelOutput outChannel

**def** testList = []

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

outChannel.write(inList[i])

testList = testList << inList[i];

}

(Amended CreateSetsOfEight Class)

**def** ChannelInput inChannel

**def** testList = []

**for** ( i **in** 0 .. 7 ) {

// put v into outList and read next input

outList.add(v);

testList = testList << v

v = inChannel.read();

}

(TestThreeToEight Class)

**class** TestThreeToEight **extends** GroovyTestCase {

**void** testMethod() {

One2OneChannel connect1 = Channel.*createOne2One*()

One2OneChannel connect2 = Channel.*createOne2One*()

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

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

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

**def** testList = [genSetOfThree, listStream, createSetOfEight]

**new** PAR (testList).run()

**def** expected = listStream.testList

**def** actual = createSetOfEight.testList

**for** (i **in** 0..22) {

*assertTrue*(expected == actual)

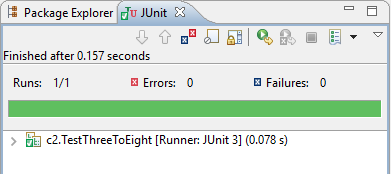
*assertFalse*(expected != actual)

}

}

}

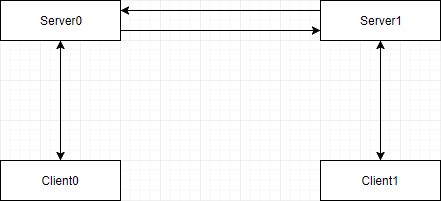
Output:



***Exercise 7-1:***

The deadlock lies when two pairs of identical numbers fall in parallel between the two clients. The first working system can run smoothly because it robustly deals with one pair of identical numbers being sent at the same time, and can rely on the next input, but upon receiving two pairs of identical numbers (the server tries to rely on another identical pair) the system deadlocks as the server has to go between requesting identical numbers for two pairs instead of one, which can be dealt with.

Pipeline Diagram:



***Exercise 8-1:***

(Amended Client Class)

**class** Client **implements** CSProcess {

**def** ChannelInput receiveChannel

**def** ChannelOutput requestChannel

**def** clientNumber

**def** selectList = [ ]

**def** propertyOrder = **false**

**void** run () {

**def** iterations = selectList.size

println "Client $clientNumber has $iterations values in $selectList"

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

**def** key = selectList[i]

requestChannel.write(key)

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

println "Key: " + key

println "V: " + v

**if** (v == key \* 10) {

propertyOrder = **true**;

}

}

println "Client $clientNumber has finished"

**if** (propertyOrder == **true**) {

println "Client $clientNumber in order"

}

**else** {

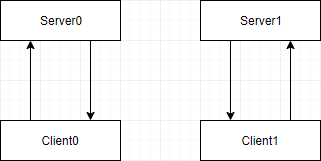
println "Client $clientNumber not in order"

}

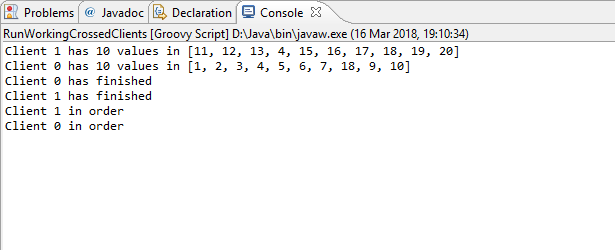
}

}

Pipeline Diagram:



Output:



Network Operation Commentary:

Upon running the program, the user will be prompted to enter the amount of clients they’d like to see assigned per server – subsequently, the backend of the program will then handle the rest.

***Exercise 9-1:***

(EventFix Class)

**class** EventFix **implements** CSProcess {

**boolean** passed = **true**

**def** ChannelInput inChannel

**def** ChannelOutput outChannel

**void** run() {

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

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

**if** (eventData.data != 100 && eventData.data != eventData.prev +

eventData.missed + 1)

{

println "Missed"

}

outChannel.write(eventData)

}

}

}

(New EventData Class)

**class** EventData **implements** Serializable, JCSPCopy {

**def** **int** source = 0

**def** **int** data = 0

**def** **int** missed = -1

**def** **int** prev

**def** copy() {

**def** e = **new** EventData ( source: **this**.source,

data: **this**.data,

missed: **this**.missed,

prev: **this**.prev )

**return** e

}

**def** String toString() {

**def** s = "EventData -> [source: "

s = s + source + ", data: "

s = s + data + ", missed: "

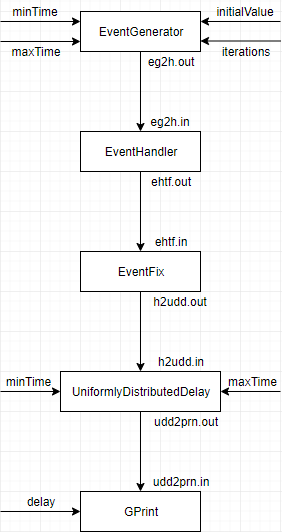
s = s + missed + "]"

**return** s

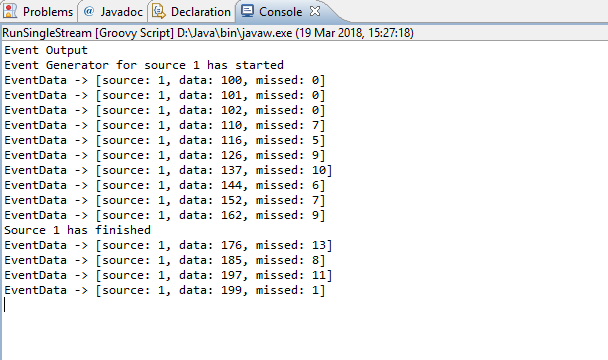
}

}

Pipeline Diagram:



Output:



Network Operation Commentary:

Running this network requires no external knowledge of the backend. It simply requires the user to start the network, and the program will handle the rest of the logic and output.

***Exercise 9-2:***

The times associated with the event generation stream seem almost directly proportional to the number of missed events. i.e, if the minimum and maximum times are increased, more events are missed – but if the times are decreased, less events are missed. By modifying the times of the processing system, increasing the times doesn’t seem to have much effect, but if the times are decreased, the system misses more events as time goes on, and the sources take much longer to finish. From this, we can assume that decreasing the time on the processing system is directly linked to how long the sources take to send/receive data.

***Exercise 9-3:***

Out of the three multiplexer, the basic multiplexer seems to be the slowest – it has an effect on each source, and seems to miss a lot of data (albeit the least of the three). The ‘PriMultiplex’ isn’t much faster, but seems to miss massive chunks of data at a time each time, possibly to compensate for a lack of processing speed. The ‘FairMultiplex’ doesn’t miss nearly as much data as the PriMultiplex, and seems to be a bit faster than the basic multiplexer.

***Exercise 11-1:***

(UserInterface Class)

**class** UserInterface **implements** CSProcess {

**def** ActiveCanvas scalerCanvas

**def** **int** canvasSize

**def** ChannelInput scaleValueConfig

**def** ChannelInput suspendButtonConfig

**def** ChannelInput printValueConfig

**def** ChannelOutput buttonEvent

**void** run() {

**def** root = **new** ActiveClosingFrame ("SCALER")

**def** mainFrame = root.getActiveFrame()

**def** printText = **new** ActiveTextArea(printValueConfig, **null**)

**def** suspendButton = **new** ActiveButton(suspendButtonConfig, buttonEvent, "SUSPEND")

**def** scaleLabel = **new** Label ("Current Scale")

**def** scale = **new** ActiveLabel (scaleValueConfig)

**def** newScaleLabel = **new** Label("Enter New Scale")

**def** newScale = **new** ActiveTextEnterField(**null**, buttonEvent)

Panel newScalePanel = **new** Panel (**new** GridLayout (1, 2));

newScalePanel.add (newScale.getActiveTextField ());

**def** scalerContainer = **new** Container()

scalerContainer.setLayout(**new** GridLayout (1, 1))

scalerContainer.add(suspendButton)

scalerContainer.add(scaleLabel)

scalerContainer.add(scale)

scalerContainer.add(newScaleLabel)

scalerContainer.add(newScalePanel)

scalerContainer.add(printText)

scalerCanvas.setSize(canvasSize, canvasSize)

mainFrame.setLayout(**new** BorderLayout())

mainFrame.add(scalerCanvas, BorderLayout.***CENTER***)

mainFrame.add(scalerContainer, BorderLayout.***SOUTH***)

mainFrame.pack()

mainFrame.setVisible(**true**)

**def** network = [ root, scalerCanvas, printText, scale, newScale, suspendButton]

**new** PAR (network).run()

}

}

(RunScaler Class)

**new** ControllerInterface ( inChannel: oldScale.in(),

suspend: pause.out(),

inject: newScale.out(),

print: scaledData.in(),

initialScale: 2 )

(ControllerInterface Class)

**class** ControllerInterface **implements** CSProcess {

**def** ChannelInput inChannel

**def** ChannelInput print

**def** ChannelOutput suspend

**def** ChannelOutput inject

**def** **int** canvasSize = 100

**def** **int** initialScale

**void** run() {

**def** controllerCanvas = **new** ActiveCanvas()

**def** scaleConfig = Channel.*one2one*()

**def** suspendConfig = Channel.*one2one*()

**def** uiEvents = Channel.*any2one*( **new** OverWriteOldestBuffer(5) )

**def** network = [ **new** ControllerManager ( fromScale: inChannel,

toScaleSuspend: suspend,

toScaleInject: inject,

fromUIButtons: uiEvents.in(),

toUISuspend: suspendConfig.out(),

toUILabel: scaleConfig.out(),

CANVASSIZE: canvasSize,

START\_SCALE: initialScale ),

**new** UserInterface ( controllerCanvas: controllerCanvas,

canvasSize: canvasSize,

scaleValueConfig: scaleConfig.in(),

suspendButtonConfig: suspendConfig.in(),

printValueConfig: print,

buttonEvent: uiEvents.out() )

]

**new** PAR ( network ).run()

}

}

(ContollerManager Class)

**class** ControllerManager **implements** CSProcess {

**def** ChannelInput print

**def** ChannelOutput toUIPrint

**def** ChannelInput fromScale

**def** ChannelOutput toScaleSuspend

**def** ChannelOutput toScaleInject

**def** **int** CANVASSIZE

**def** **int** START\_SCALE

**def** ChannelInput fromUIButtons

**def** ChannelOutput toUILabel

**def** ChannelOutput toUISuspend

**void** run() {

// Send initial scale value to UI

**def** scale = START\_SCALE

toUILabel.write( scale.toString() )

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

// Read direction from UI

**def** direction = fromUIButtons.read()

// If SUSPEND button pushed...

**if** (direction == "SUSPEND") {

// Change button to SUSPENDED (does nothing when clicked)

toUISuspend.write("SUSPENDED")

// Suspend Scale process

toScaleSuspend.write(0)

// Get current scale value and send to UI

scale = fromScale.read()

toUILabel.write(scale.toString())

// Wait till direction is an integer value

**def** integerScale = **false**

**while** (!integerScale) {

direction = fromUIButtons.read()

**try** {

// Get new scale value from direction

direction = Integer.*parseInt*(direction);

scale = direction

integerScale = **true**

} **catch** (NumberFormatException e) {

System.***out***.println("Incorrect Value!");

}

}

// Change button to SUSPEND and update scale in UI and Scale

toUISuspend.write("SUSPEND")

toUILabel.write(scale.toString())

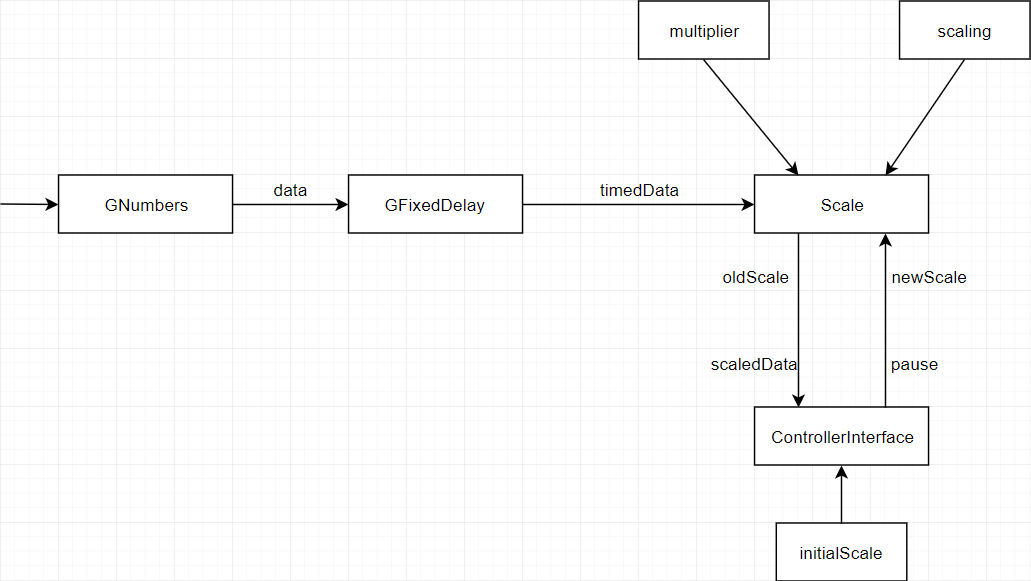
toScaleInject.write(scale)

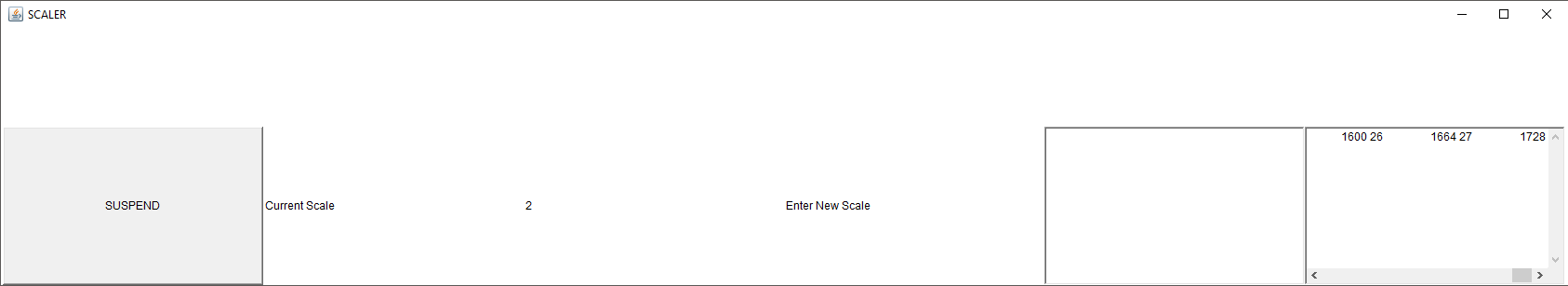
}

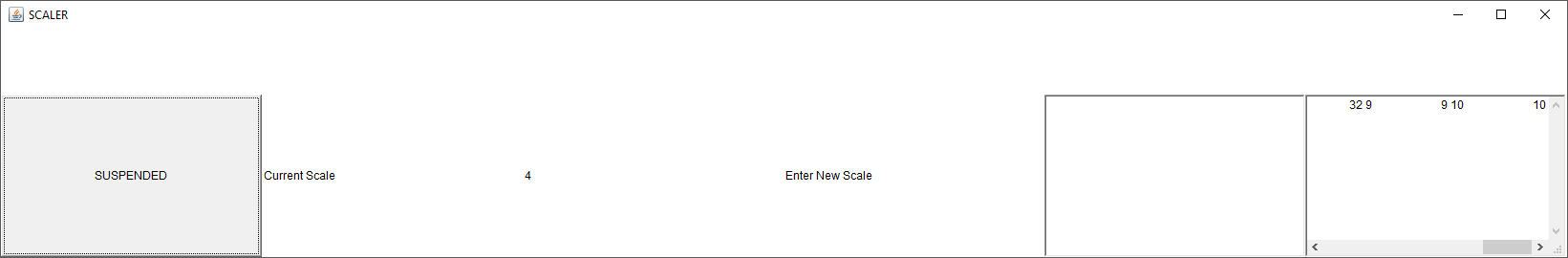
}

}

}

Pipeline Diagram:

Output:



Network Operation Commentary:

Upon running the network, the user will be presented with a window, which has a suspend button, the current scale, as well as the values being scaled (along with console output). The user can then suspend the operation of the network and inject a new scaler by inputting a value next to the scaled value box, and that value will become the new scaler for the network.

Coursework Challenge

***Requirement 2:***

Initially, we had decided on using a token and preconditions to settle most of the turn functions - when it came to development, we soon realised that preconditions were too long-winded to implement, as we had to account for any number of possibilities at any point in the game with *N* amount of players. However, within the design document, we talked about implementing a ‘token’ system, wherein we would make a system which passes around either a boolean/integer value between the players whenever it was their turn. This was done using a ‘currentTurn’ variable within the GameDetails class, which can be accessed by any of the other classes through a constructor.

However, instead of using preconditions, we decided to use a basic 2D array to track the details of selected pairs. Our system keeps a constant track of the current amount of players, as well as the current cards/pairs selected and the turn order. Therefor, it was fairly trivial to create a system which used the communication structures we were provided without having to make any changes to them, as the system basically tracks every detail of the chosen cards.

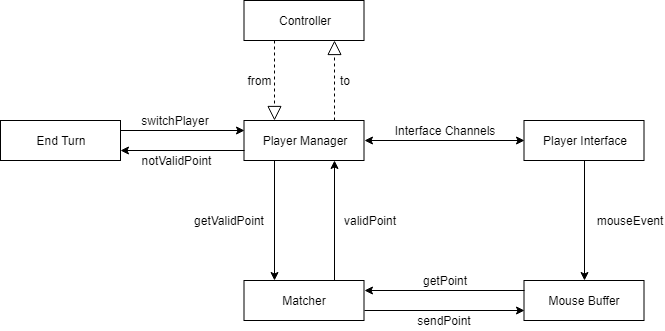
There was also a strange flickering issue which occurred in the project due to the constant refreshing of the board. To fix this, we implemented a basic timer delay, which was constantly in effect. Instead of refreshing every 1/6th of a second or so, the board would now refresh every 2 seconds.

***Requirement 3:***

The challenge in question is to create a turn system for a pairs game. The turn system should require roughly 3 maps (or lists) to contain the squares of the board. A map would be more beneficial as the contained values can be recursively searched for and matched. An example would be giving each player an empty list at the game start, and placing the values or squares (or board coordinate) into the respective player’s map. This way, each turn, the maps can be searched through and if a square belongs to player A, do not show it to player B. If possible, this could be through an attached boolean value if all else fails.

The turns can be done through a token system, where a value is passed around players. This token could be a boolean value where ‘true’ refers to one specific player, and ‘false’ for the other - this way, the game could constantly check within the while loop if the turn is with the valid player. The turn system should only require 3 channels - playerAtoEnd, playerBtoEnd and endTurn. This way, the channels could send a signal for ending a turn (when two squares do not match), and the endTurn channel could send a signal to pass along the token (respectfully passing on the turn), as well as for player A or B to start their turn.

*Pipeline Diagram:*



***Requirement 4-1: Group M***

This solution showed great understanding of the brief and structure behind it, the game was fully functioning to a high standard. The multiple players were able to clearly see the squares being selected by the playing user and once the tiles were claimed they were shown as plain white squares to all users. This solution was also very clear in showing which player was currently playing and kept track of the scores throughout all games, he game also renewed automatically with a new randomised game board. However, the game had no means of withdrawal - players didn’t seem to be able to leave a game in any way.

In their design document, the group talked about identifying a way to blend the ‘TurnNumber’ or ‘TurnProperty’ with the Player IDs. To approach this, they seeked to use the following formula:

PlayerID = ((TurnNumber – 1) mod NUMBER\_OF\_PLAYERS)

But unfortunately, the final product didn’t seem to use this, or it was modified. The group also wanted to make two canvases communicate in the player interface, but we believe this was approach differently in the final version, likely by means of an array or map to contain pair values.

***Requirement 4-2: Group G***

The solution to this team’s game was not fully functional, not being able to play with multiple users and showed minor issues with single player games. Within the user interface one player was able to join the controller and enrol onto a game, however for this player the game board didn’t fully work to the expected standard. With the game not being fully functional we were unable to determine the behaviour between multiple users, we were also unable to see the game getting renewed so cannot state if the game board randomised automatically after the current game had finished.

This is likely down to the way the communication structures were operating. It’s possible, but unlikely that the group may have caused a deadlock somewhere within the system when they were trying to implement a feature.

From the design document, the group seemed to believe they could implement turns by adding ‘EndTurn’ and ‘EndGame’ functions to the controller manager. If the group ended up using this structure with a basic queue, it’s possible the game did not work as a result of bad communication between the channels, or the queue doesn’t work correctly.

*Challenge Development Team:*

40288424

40208762