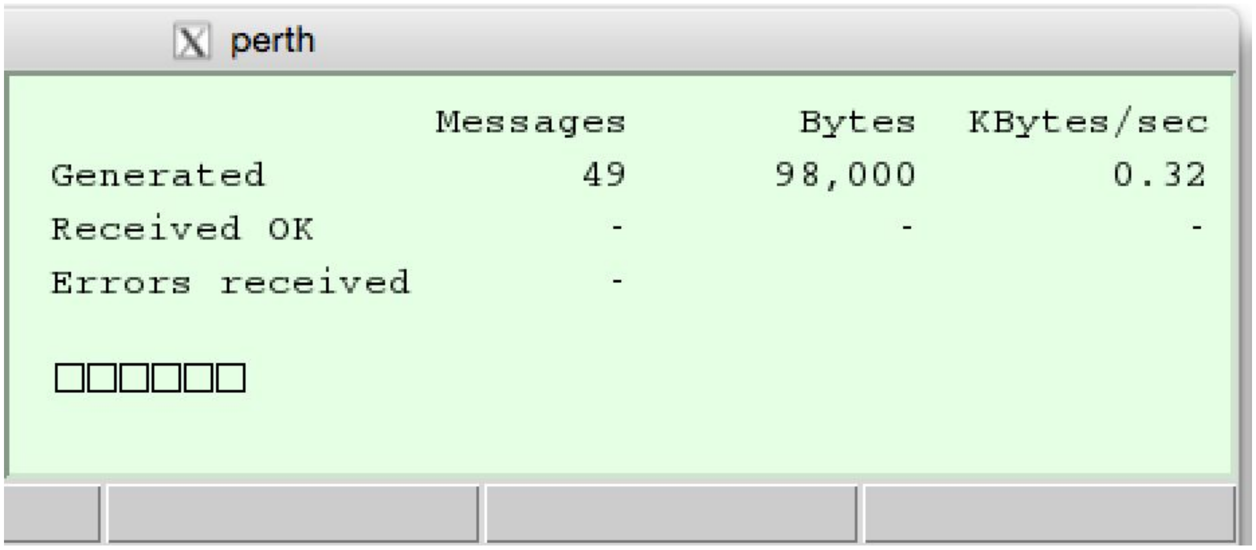
CMPUT 313
Lab 2
Jonathan Cairo
jcairo

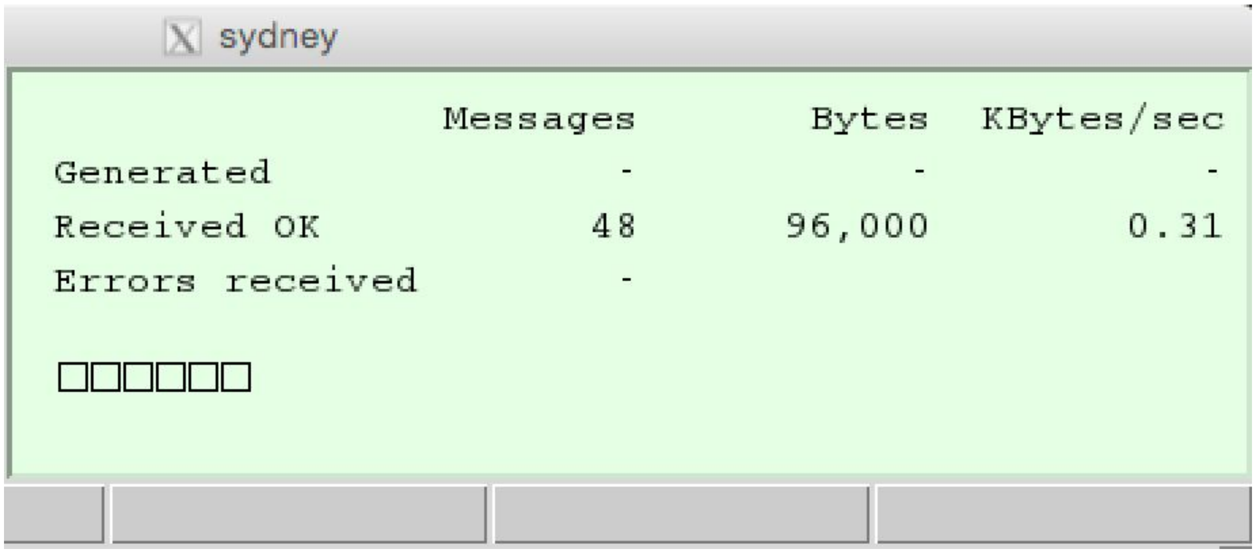
Objectives:
The objective of Part 1 was to mathematically analyze theoretical throughput of several hypothetical stop and wait networks and compare the mathematical analysis to the simulated results from cnet. This was a very valuable exercise, it had us think deeply about how to model such networks. Since the simulated results did not match the theoretical results exactly we had to consider several extraneous factors that may affect the cnet simulation but that weren't considered in our mathematical model. This required some intricate thinking on the nature of individual samples and some factors included in cnet but not in our formula.

In Part 2 we constructed another hypothetical network where each router node between hosts ran a stop and wait protocol. This protocol had to manage acking data packets and waiting for acks of forwarded data packets before accepting further data packets. We also had to make this protocol robust to frame corruption and loss. This was a very valuable exercise in that it helped more deeply understand the mechanics of stop and wait and also forced us to go outside the standard stop and wait protocol by having to deal with corrupted and lost packets which we have not covered in class. This encouraged further understanding of the mechanics of stop and wait.

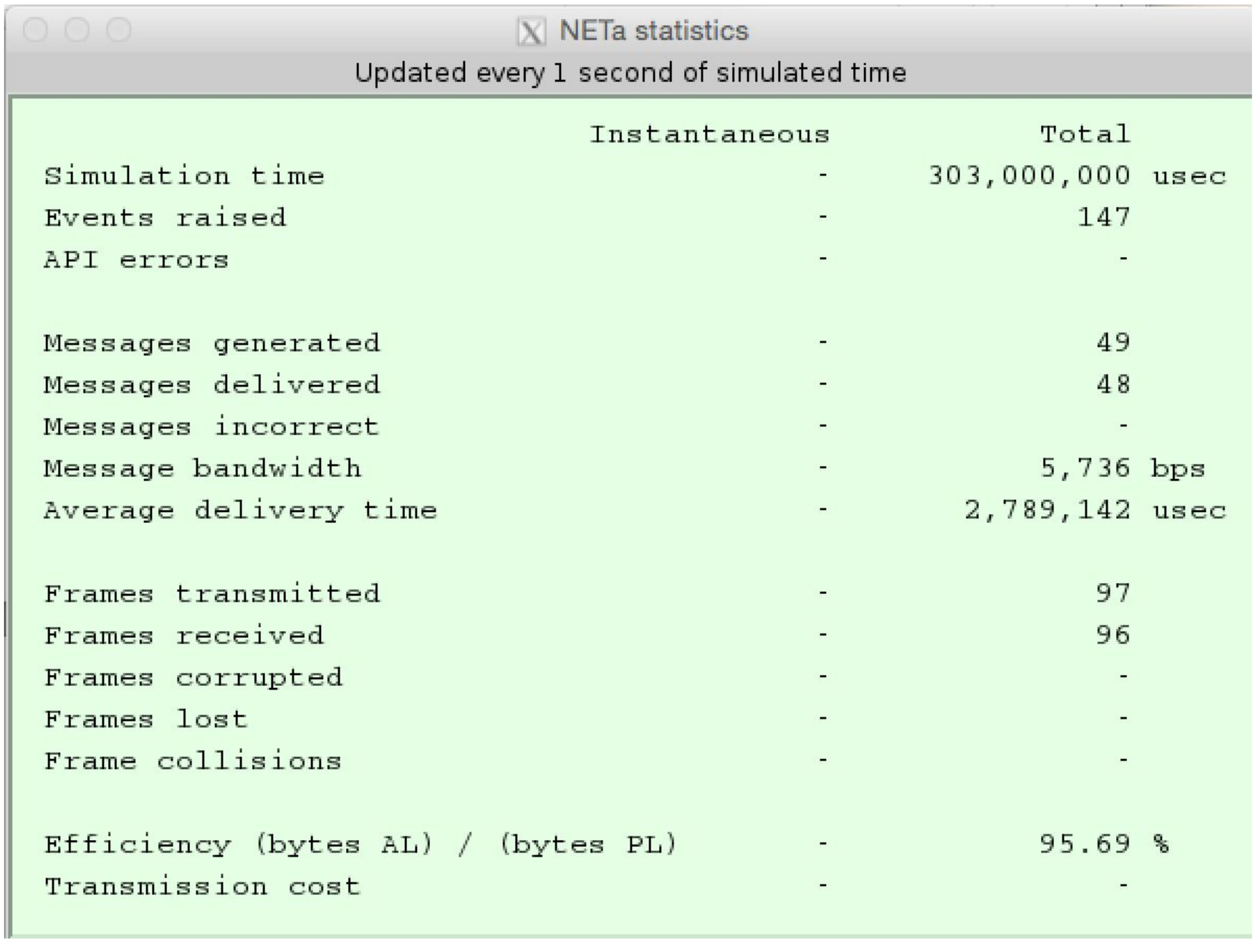
PART 1:
1.
NETa



```
                  Messages        Bytes  KBytes/sec
Generated               49       98,000        0.32
Received OK              -            -           -
Errors received          -

□□□□□□
```

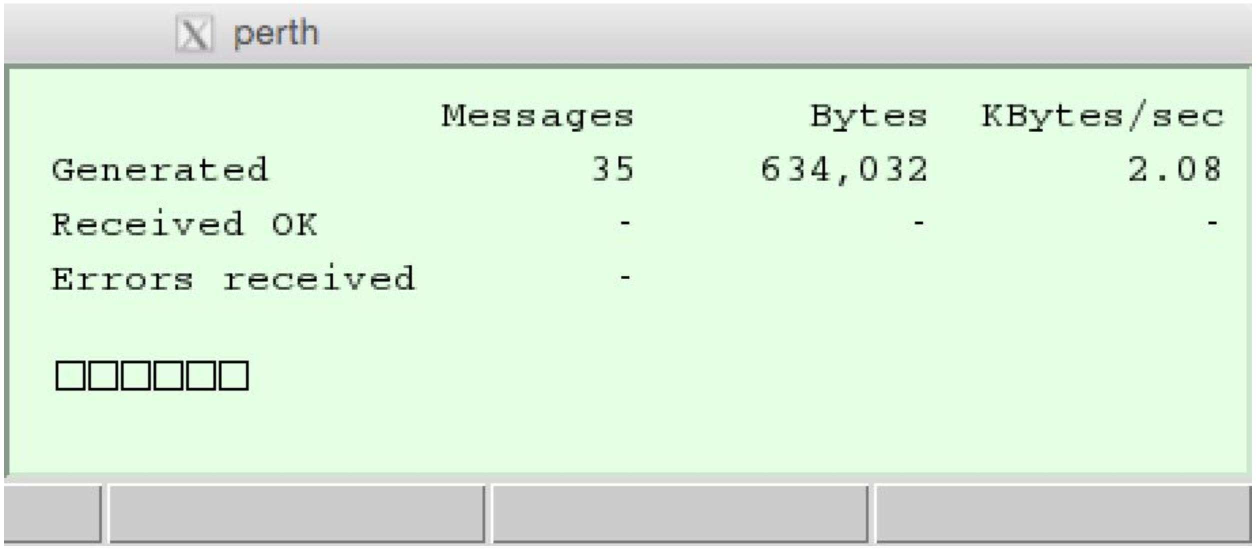


```
                  Messages        Bytes  KBytes/sec
Generated                -            -           -
Received OK             48       96,000        0.31
Errors received          -

□□□□□□
```

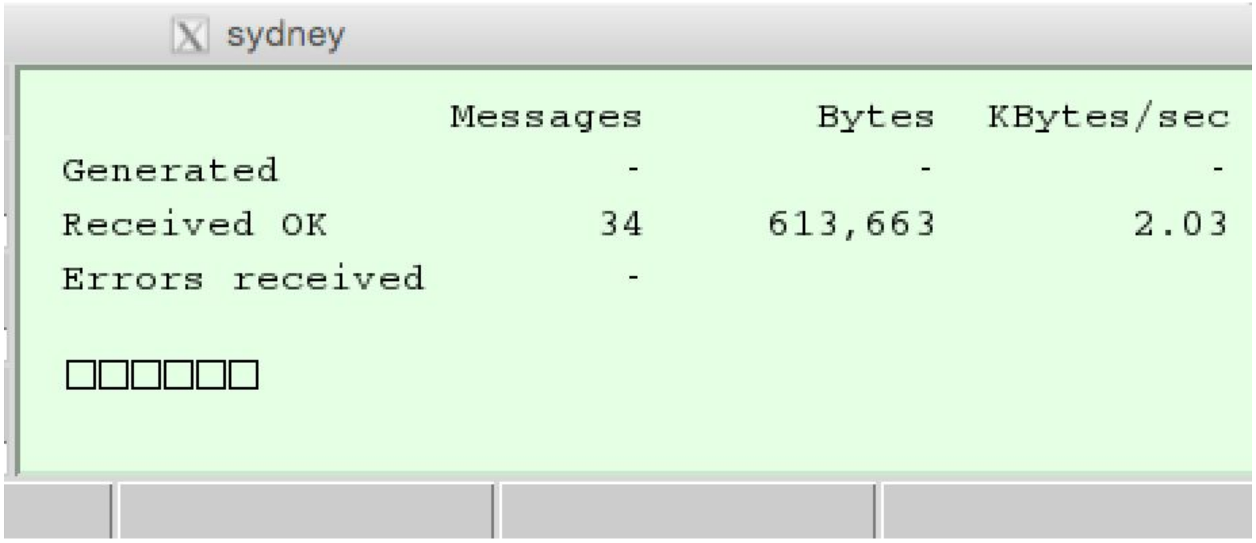


```
NETa statistics
Updated every 1 second of simulated time

                                        Instantaneous             Total
Simulation time                                     -       303,000,000 usec
Events raised                                       -               147
API errors                                          -                 -

Messages generated                                  -                49
Messages delivered                                  -                48
Messages incorrect                                  -                 -
Message bandwidth                                   -             5,736 bps
Average delivery time                               -         2,789,142 usec

Frames transmitted                                  -                97
Frames received                                     -                96
Frames corrupted                                    -                 -
Frames lost                                         -                 -
Frame collisions                                    -                 -

Efficiency (bytes AL) / (bytes PL)                  -             95.69 %
Transmission cost                                   -                 -
```

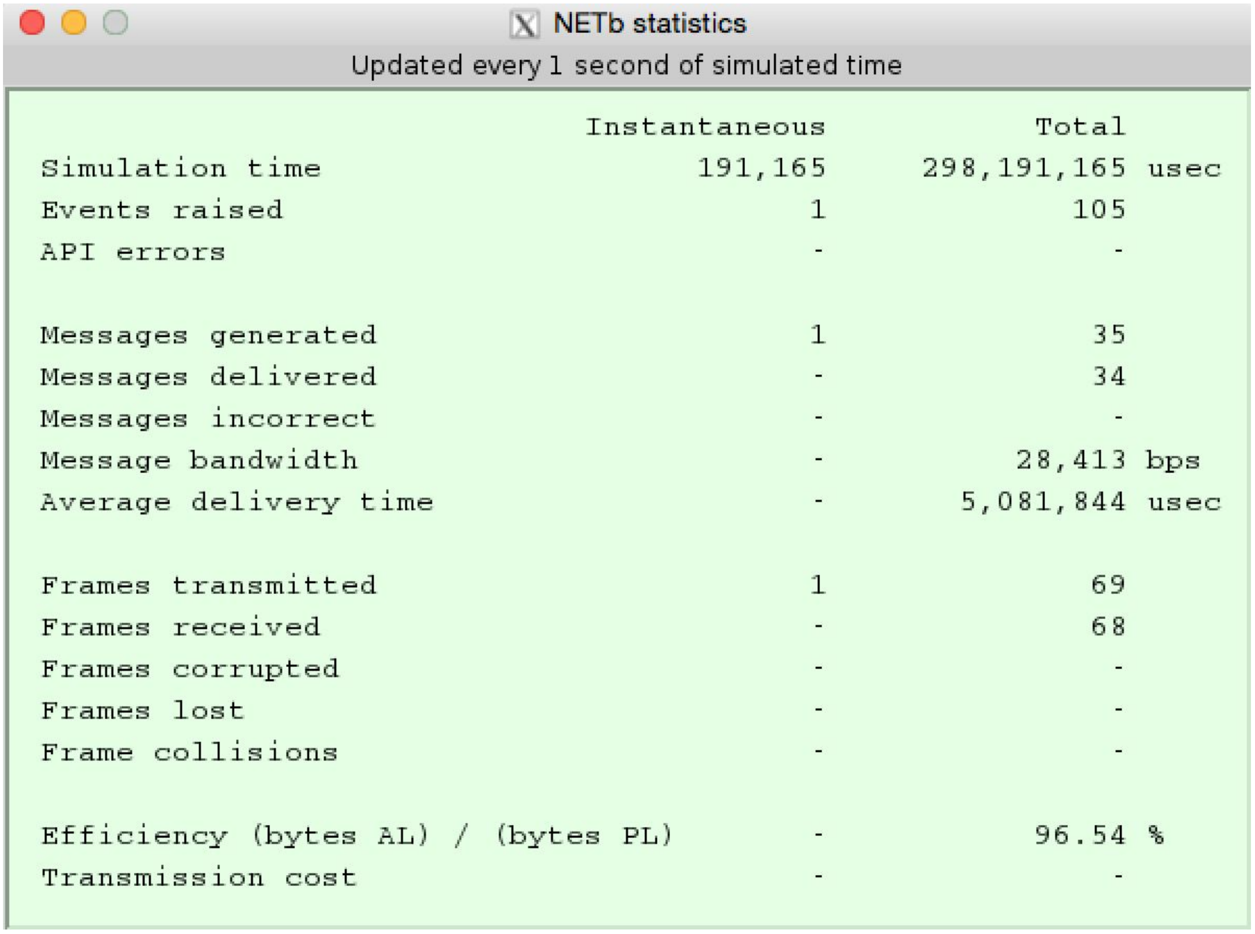
NETb



```
                 Messages        Bytes  KBytes/sec
Generated              35      634,032        2.08
Received OK             -            -           -
Errors received         -

□□□□□□
```



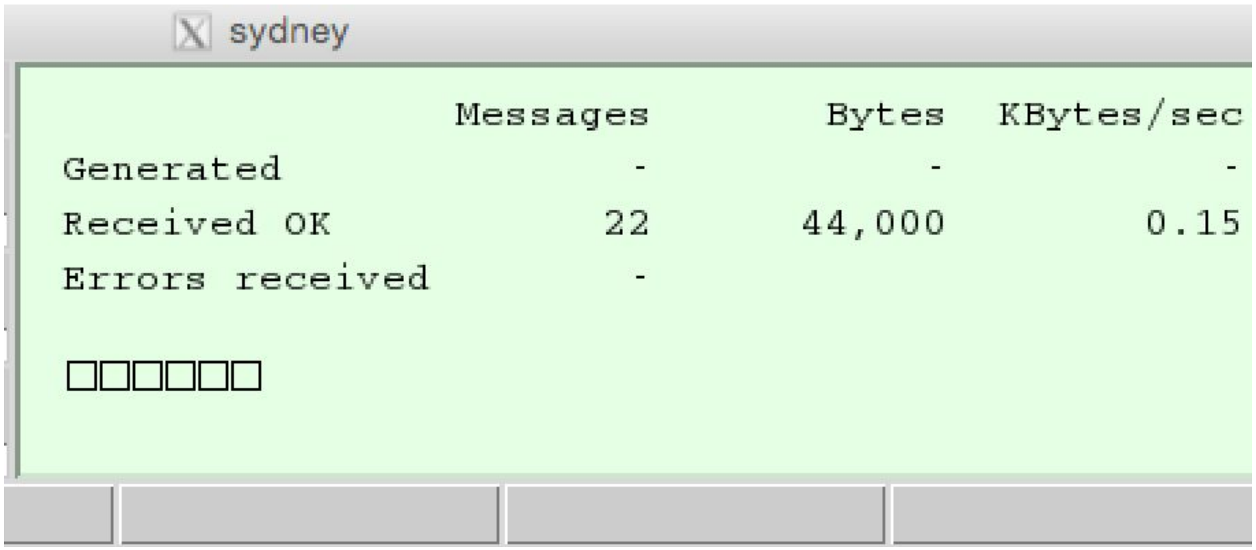
```
                 Messages        Bytes  KBytes/sec
Generated               -            -           -
Received OK            34      613,663        2.03
Errors received         -

□□□□□□
```


NETb statistics

Updated every 1 second of simulated time

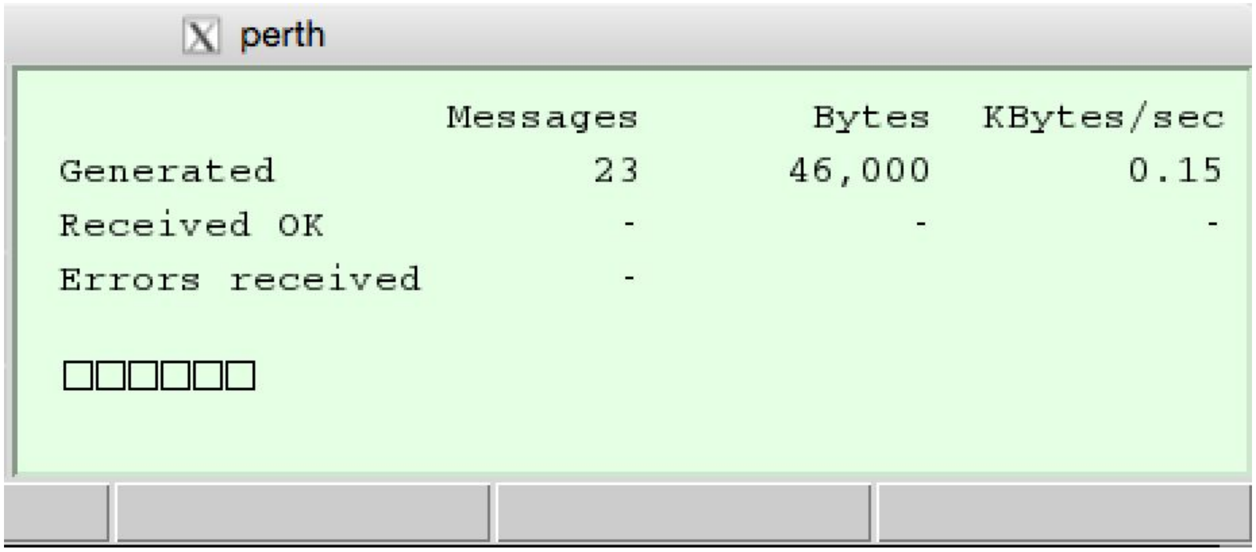
```
                              Instantaneous          Total
Simulation time                     191,165    298,191,165 usec
Events raised                             1            105
API errors                                -              -

Messages generated                        1             35
Messages delivered                        -             34
Messages incorrect                        -              -
Message bandwidth                         -         28,413 bps
Average delivery time                     -      5,081,844 usec

Frames transmitted                        1             69
Frames received                           -             68
Frames corrupted                          -              -
Frames lost                               -              -
Frame collisions                          -              -

Efficiency (bytes AL) / (bytes PL)        -          96.54 %
Transmission cost                         -              -
```

NETc



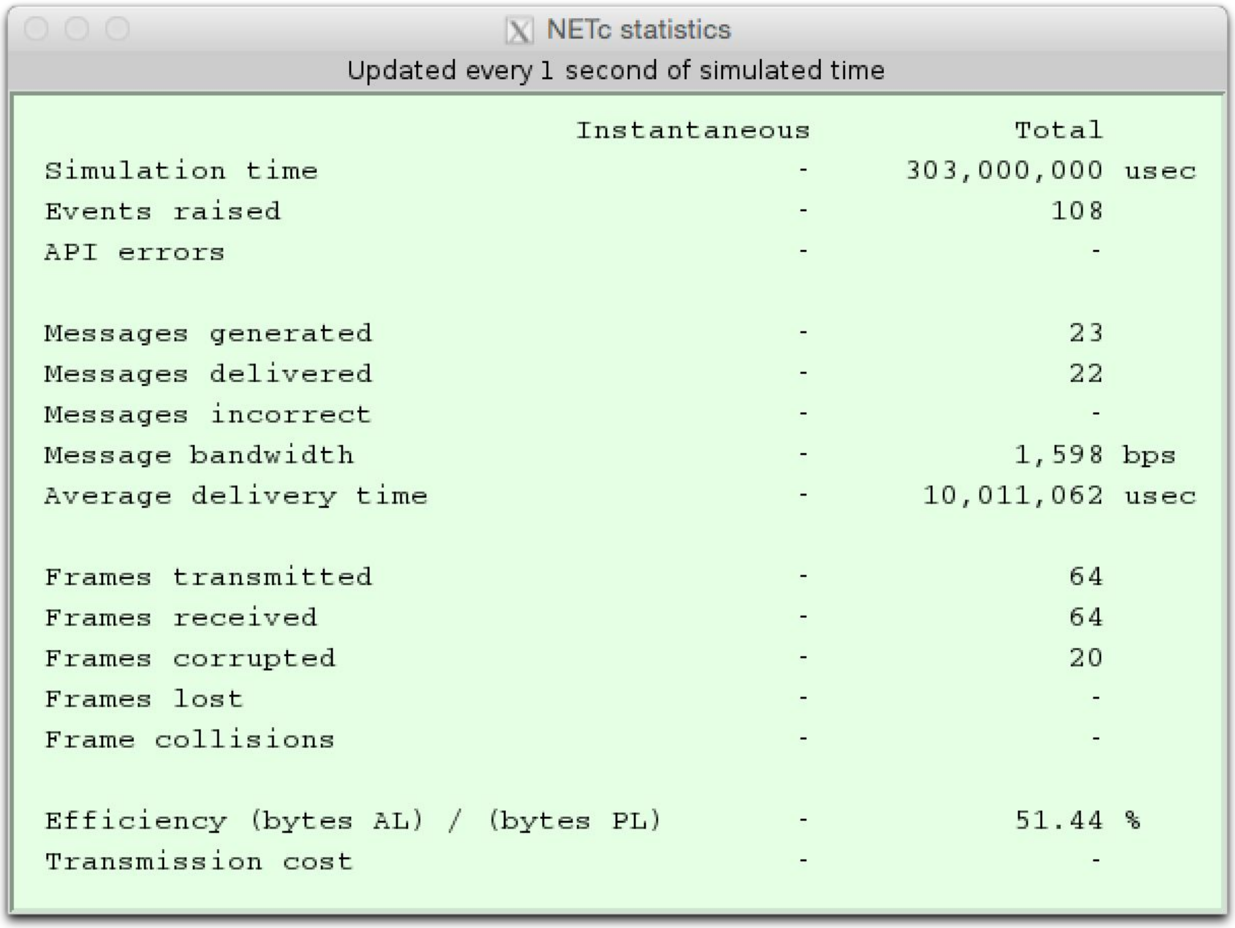
sydney

```
                      Messages        Bytes   KBytes/sec
Generated                    -            -            -
Received OK                 22       44,000         0.15
Errors received              -

□□□□□□
```



perth

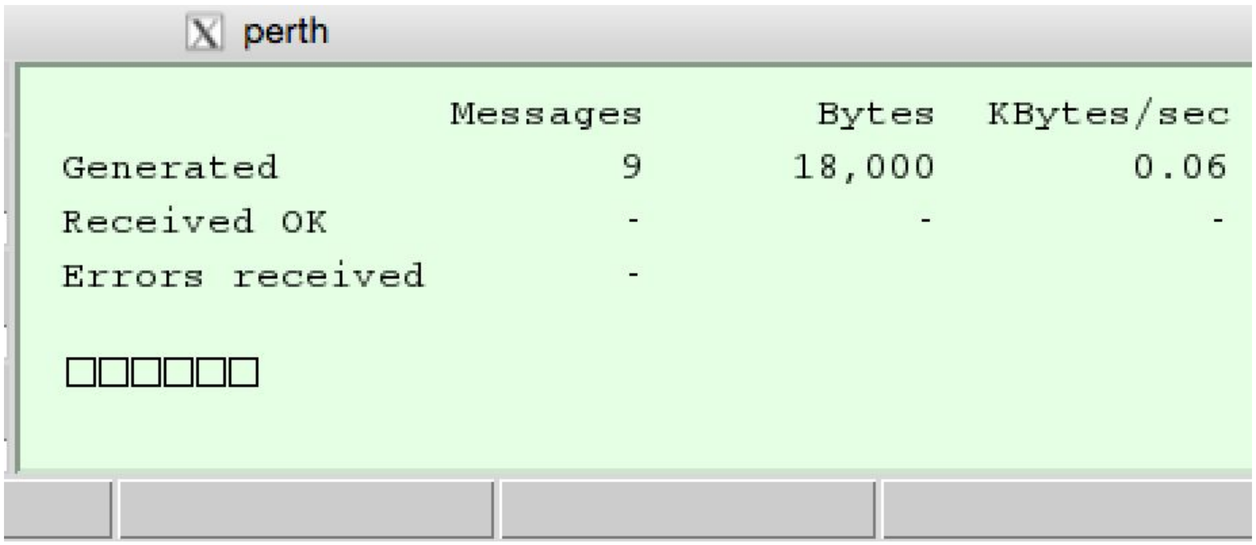
```
                      Messages        Bytes   KBytes/sec
Generated                   23       46,000         0.15
Received OK                  -            -            -
Errors received              -

□□□□□□
```


NETc statistics

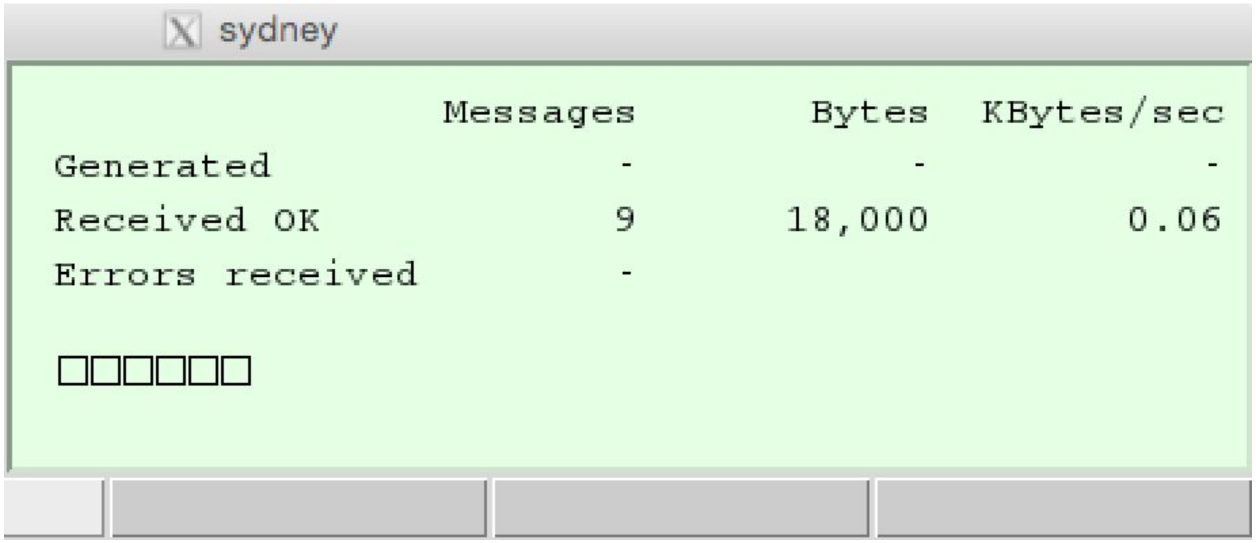
Updated every 1 second of simulated time

```
                                   Instantaneous           Total
Simulation time                                -     303,000,000 usec
Events raised                                  -             108
API errors                                     -               -

Messages generated                             -              23
Messages delivered                             -              22
Messages incorrect                             -               -
Message bandwidth                              -           1,598 bps
Average delivery time                          -      10,011,062 usec

Frames transmitted                             -              64
Frames received                                -              64
Frames corrupted                               -              20
Frames lost                                    -               -
Frame collisions                               -               -

Efficiency (bytes AL) / (bytes PL)             -           51.44 %
Transmission cost                              -               -
```

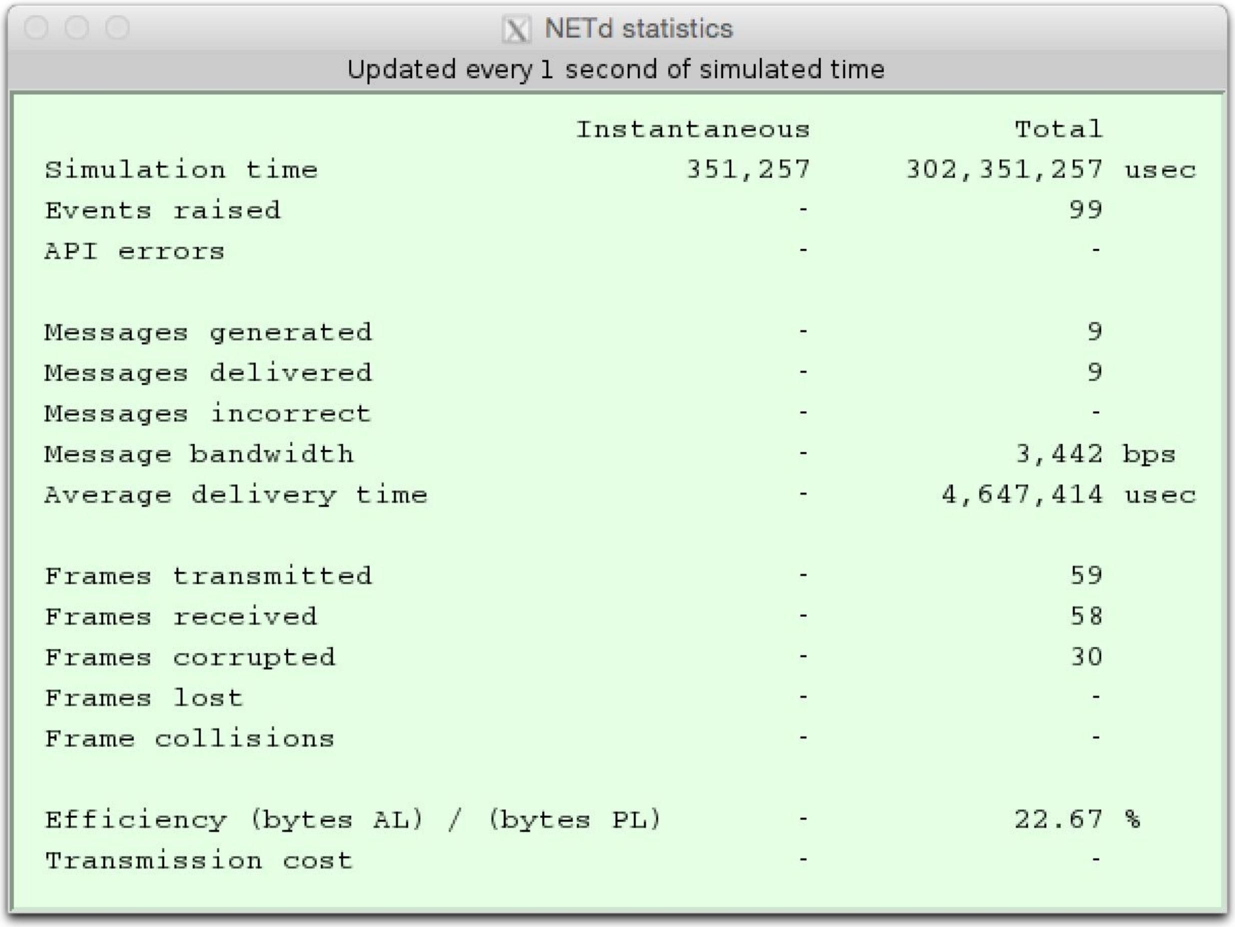
NETd



```
                         Messages          Bytes   KBytes/sec
Generated                       9         18,000         0.06
Received OK                     -              -            -
Errors received                 -

□□□□□□
```


sydney

```
                      Messages        Bytes   KBytes/sec
Generated                    -            -            -
Received OK                  9       18,000         0.06
Errors received              -

□□□□□□
```


NETd statistics

Updated every 1 second of simulated time

```
                                      Instantaneous          Total
Simulation time                             351,257    302,351,257 usec
Events raised                                     -             99
API errors                                        -              -

Messages generated                                -              9
Messages delivered                                -              9
Messages incorrect                                -              -
Message bandwidth                                 -          3,442 bps
Average delivery time                             -      4,647,414 usec

Frames transmitted                                -             59
Frames received                                   -             58
Frames corrupted                                  -             30
Frames lost                                       -              -
Frame collisions                                  -              -

Efficiency (bytes AL) / (bytes PL)                -          22.67 %
Transmission cost                                 -              -
```

Part 1 Question 2
We start with the throughput formula from the notes.
Since we can have a range of packets sizes we use the formula for expected value of a uniform random variable to get an expected packet size. We then take into consideration the probability of corruption/loss by dividing the result by 1/1-probability of loss/corruption.This gives us the first formula below for NETa, NETb, and NETc

Average throughput =

$$\frac{1}{\left[\dfrac{\left(\dfrac{\frac{\textit{message size lower limit} + \textit{message size upper limit}}{2}}{\textit{link speed}}\right) + RTT}{\dfrac{1}{1 - \textit{probability of corruption data frames}}}\right]} \quad \text{pkts/sec}$$

The mathematical model for NETd is similar we can simply square the denominator since the probability of corrupt data frames is the same for ack packets.

$$\frac{1}{\left[\dfrac{\left(\dfrac{\frac{\textit{message size lower limit} + \textit{message size upper limit}}{2}}{\textit{link speed}}\right) + RTT}{\left(\dfrac{1}{1 - \textit{probability of corruption data frames}}\right)^2}\right]} \quad \text{pkts/sec}$$

To obtain the KB/sec we simply take the results from the above formula and multiply by the average message size.

Part 1 Question 3
NETa
RTT = 2.5 seconds (cnet quantity) * 2 = 5 seconds
TPKT = 2kB(cnet quantity)/7kB/sec(cnet quantity)

Throughput =

$$\frac{1}{\left[\frac{\left(\frac{\frac{2kB+2kB}{2kB}}{7kB/sec}\right)+5\ seconds}{\frac{1}{1-0}}\right]}$$

= .189 pkts/second or .3784 kB/second

NETb

Message size lower limit = 2kB (Cnet quantity)
Message size upper limit = 32.768kB (Cnet quantity)

All other values are the same as above equation
Throughput =

$$\frac{1}{\left[\frac{\left(\frac{\frac{2kB+32.768kB}{2kB}}{7kB/sec}\right)+5\ seconds}{\frac{1}{1-0}}\right]}$$

= .1336 pkts/second or 2.3225 kB/second

NETc

Probability of corruption data frames = .5 (used in snet probframecorrupt = 1)

$$\frac{1}{\left[\frac{\left(\frac{\frac{2kB+2kB}{2kB}}{7kB/sec}\right)+5\ seconds}{\frac{1}{1-.5}}\right]}$$

= .0945 pkts/second or .189 kB/second

NETd

Probability of corruption for both ack and data frames is .5 so we square our denominator

$$\frac{1}{\left[\frac{\left(\frac{\frac{2kB + 2kB}{2kB}}{7kB/sec}\right) + 5\ seconds}{\left(\frac{1}{1-.5}\right)^2}\right]}$$

= .04725 pkts/second or .0945 kB/second

Part 1 Question 4
NETa mathematical model predicts .189 pkts/second or .3784 kB/second
The simulation provided .32Kb/second
This is quite close

NETb mathematical model predicts .1336 pkts/second or 2.3225 kB/second
The simulation provided 2.08kB/second
Again these results are quite close

NETc mathematical model predicts .0945 pkts/second or .189 kB/second
The simulation provided .15Kb/second
Again these results are quite close

NETd mathematical model predicts .04725 pkts/second or .0945 kB/second
The simulation provided .06kB/second
Again these results are quite close.

There are various reasons why the simulation results do not meet the predicted results exactly.

1. Each simulation trial has its own unique characteristics. Since it is a sample size of 1 for each trial we are much more likely to get results in the tails of the normal distribution. We could come close to our predicted results if we ran several trials of each simulation and averaged the results.
2. In the theoretical model new messages are sent out at Tpkt + RTT however in the simulation we have a message rate of 100ms. This message rate is not accounted for in our formula.
3. Cnet uses 1024 Bytes in a kB whereas our formula uses 1000 Bytes in a kB so we experience some rounding error biasing our results downward.
4. There are other reasons but these seem to be the largest contributors in my opinion.

Part 2 (Design Overview)
Highlights:

- 2 protocols run on each router node.
- The only shared state between these protocols is a buffer variable.
- Upon receipt of a data packet, the data protocol checks for corruption
- If the packet is not corrupt an ack is sent.
- If the packet has the appropriate sequence number the data packet is forwarded to the next router, a timeout is set to resend upon expiration, and the packet is also saved in a buffer.
- This buffer is shared with the ack side of the protocol.
- If any data messages are received while the buffer is full they are ignored.
- When a router receives an ack, it ensures it has the appropriate sequence number, and then clears the buffer so that the left side of the protocol can begin accepting data messages again.

High Level Description
The requirements asked us to implement 2 stop and wait protocols at each node.
I implemented this by using the basic stop and wait protocol we were provided, adding 3 routers between the hosts, and adding a 2 protocols to each router. One manages data packets originating from the left. On receipt of a data packet, after checking for the appropriate sequence number, the router sends an ack to the data packet sender (could be router or host) and forwards the data packet to the next (router/host). After forwarding it the packet is stored in a buffer. A timeout is set and the packet is resent from the buffer on each timeout until an ack is received. If the buffer is full the protocol will ignore any further incoming data packets. The other protocol managing ack packets from the right simply waits for ack packets to arrive, checks the sequence number and if correct clears the buffer freeing up the left side protocol to receive data packets once again.

Project Status:
The requirements and analysis required in part 1 are contained in this report.

The program for part 2 functions as requested in the assignment. That includes with no frame loss or corruption, with one of the two, and with both corruption and loss. You can run and examine the application by typing cnet SAW in directory containing the code.

Part 1 of the project was straightforward, my only issue was not dividing by the appropriate probability originally. I fixes this by squaring the denominator in my equation.

In part 2 the largest problem I had was I originally removed the unconditional ack of all packets in the example stop and wait code. This created deadlocks when errors in the channel were introduced. For example if router 1 received data packet sequence 0 and acked it, but the ack was corrupted or lost the sender would continue to send the packet with sequence number 0, but the router would expect sequence 1. This created a deadlock situation where once the first lost/corrupted ack occurred no further progress could be made. After including this back in original stop and wait code and also including it in my stop and wait data packet code the program ran as expected.

Another issue I had was understanding that the stop and wait protocol was independent at each node. I originally assumed that the ack required by the sender was originating from the receiver (perth) and passed by the routers back to the receiver. This caused some confusion in my original design but the TA helped me understand that each protocol was independent and had no shared state with other nodes.

Acknowledgements:
All information used was from the Cnet documentation and helpful TAs.