EPA CA2

X00130180 – Jordan Williams

Script:

```
#!/bin/bash

# Store filenames in variables for reusability/ensures consitency.
results=./results.dat
cleanup=./cleanup.sh
stats=./stats.txt
loadtest=./loadtest
synthetic=./synthetic.dat
csv=./results.csv

# If a previous run was made it will clean up the previously made files
# for the current run, instead of appending to those files.
# Does this by checking if ./results.dat file exists. If not, ignore.
if [ -e "$results" ]; then
    echo "Cleaning up from previous run..."
        # Calls cleanup script that just removes the previously made files.
        # This was useful in the testing of my application as I didn't have
        # to constantly clean up from previous runs as the task was automated.
        rm $results $stats $synthetic
fi

# Just some output to give some understanding to the user, so as they have
# an idea of what is happening under the hood.
# Felt like it made my script a little bit prettier :)
echo "Will now begin to run ${loadtest##*/}, please wait..."
echo "#########################################################"
# ${results##*/} returns only the filename from the results variable declared
# at the beginning of my script e.g "Output from results.dat".
echo "#            Output from ${results##*/}            #"
echo "#########################################################"

# Just adds the headers C0, N, and idle to my results file.
printf "C0\tN\tidle\n" >> $results
cat $results
# For loop as instructed in the document to loop through script from 1 to 50.
for i in {1..50}
do
        # timeout n so the script will only run for n seconds before it times out.
        # Runs the script ./loadtest stored in the loadtest variable with param
        # specified by the index of the for loop, i, e.g. ./loadtest 1 then 2, 3 etc.
        # Runs the mpstat command and stores the results from the command in the
        # stats file (./stats.txt)
        timeout 2 $loadtest $i | mpstat 1 1 >> $stats

        # Creates variable c0 that stores the current value of that column during
        # this run. This value is the number of lines contained in the synthetic.dat
        # file, i.e. number of transactions completed.
```

```
        c0=`cat $synthetic | wc -l`
        # Uses printf so I can tab etc.
        # Puts variable held in c0 and i into results.dat in format "24   3         "
        printf "$c0\t$i\t" >> $results

        # Using awk I can retrieve the value of the 12th header (idle).
        # But because multiple results of mpstat are held in the stats file I need
        # to only take the last results, which would be the latest result of mpstat
        # that was appended into the stats file.
        awk '{print $12}' $stats | tail -n1 >> $results | tr -d '\n'
        # Just to give continious output to the user so they know the script is
        # indeed running correctly.
        cat $results | tail -n1
done

tr -s '\t' < $results | tr '\t' ',' > $csv
```

Specs For The VM

I wrote my script on my Linux machine and ran it on the Fedora VM's supplied to us in the Enterprise Performance Application Lab. I then limited the VM to 1 CPU through VirtualBox, as per the CA specification.

Virtual Machine Specs

Name:
- Fedora 28

Type:
- Linux

Version:
- Fedora (64-bit)

Memory:
- 2505 MB

CPU's:
- 1

Exec Cap:
- 100%

Storage

Type:
- Normal (VMDK)

Virtual Size:
- 50.00 GB

Actual Size:
- 42.32 GB

Details:
- Dynamically Allocated Storage

Data Visualisation

After I had gathered all the necessary data it was then time to begin the visualisation outlined in the document.

Ui vs. N

I went about calculating this was via the following method:

$$\frac{Bi}{100-[Idle]}$$

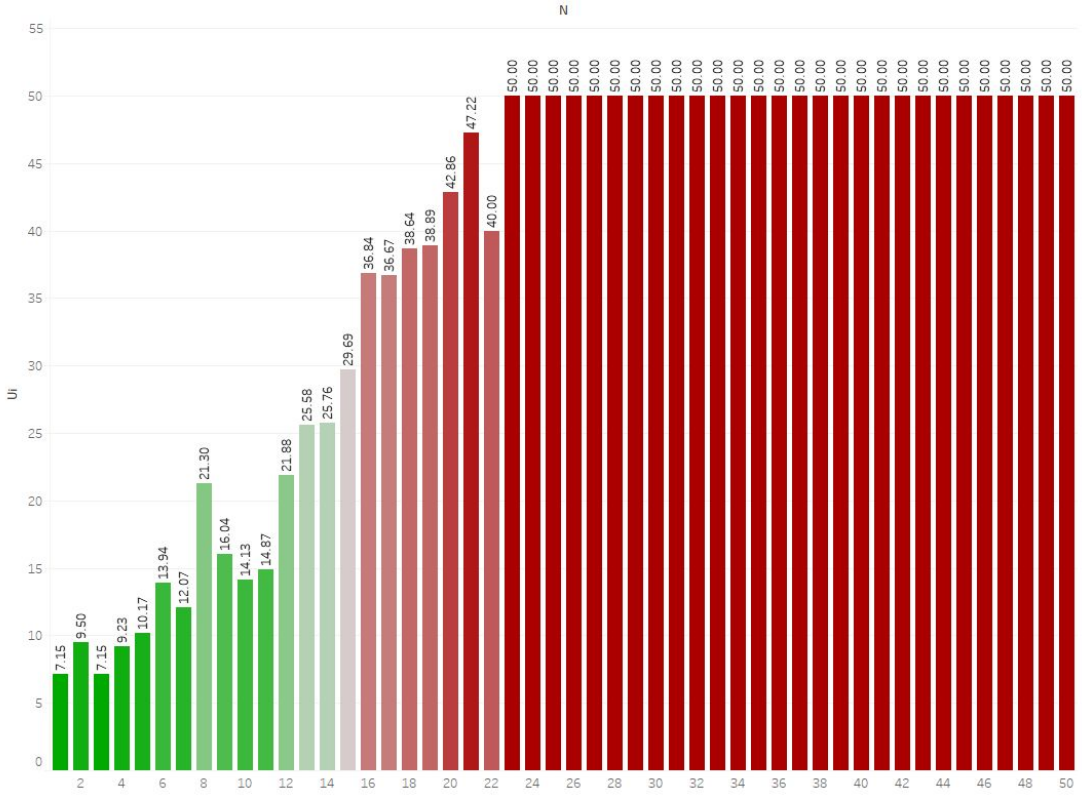
$$\frac{T}{2}$$

$$Ui$$

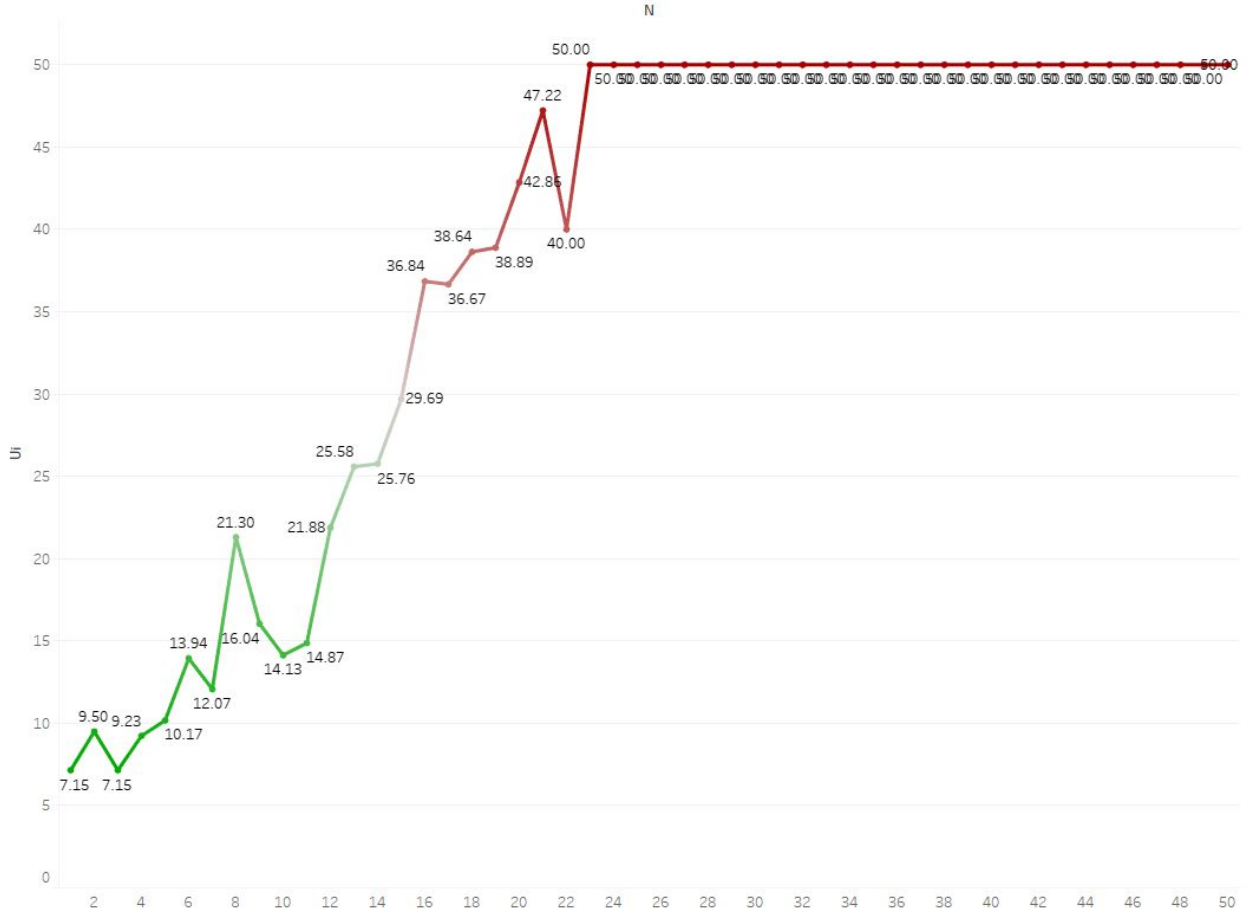
That left me with the following result of:

$$[Bi]/[T]$$

Ui vs. N



Ui vs. N Line



As you can see the the utilisation of resources is consistently rising the longer it goes, until eventually it gets maxed out and no longer has anymore resources to devote to any tasks. This explains why from N=24 Ui results in 50, the max. The results weren't surprising as I was only running my machine with limited resources, such as 1 cpu. With such limited resources, it's easy to see why the machine quickly utilised all its resources. When I was running my script with a lot more resources, it didn't approach anywhere near 50. The results will differ depending on which system you are running it on.

The range on the graph was from 7-50. That is a stark contrast in values. This shows how the program running is gradually needing more and more resources to function. With this number of resources currently allocated to the Fedora VM, I can run 23 iterations before all my resources are being utilised by the loadtest.

Di vs. N

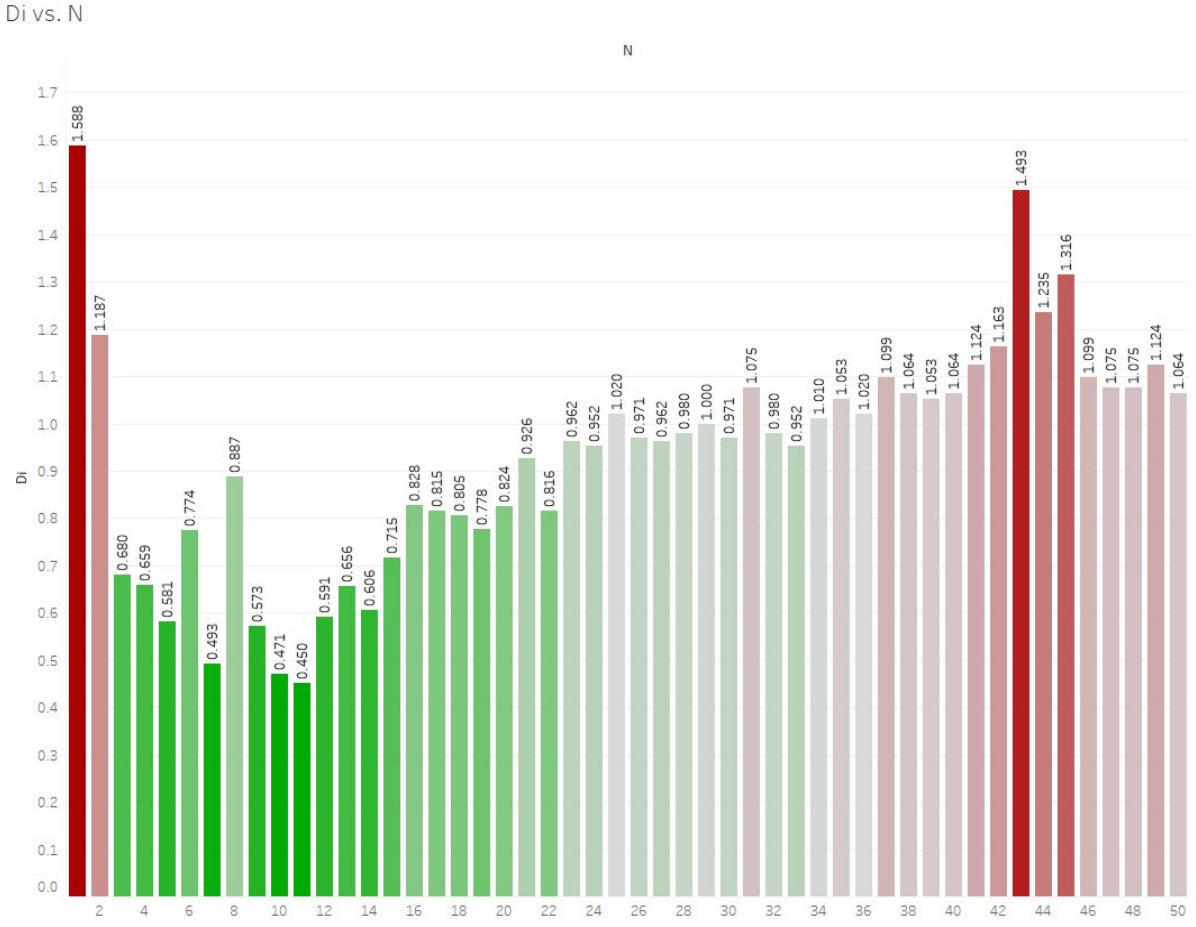
I went about calculating this was via the following method:

Di

```
([Ui]*[T])/[C0]
```

This formula for Di is used to calculate the service demands via resource utilizations and system throughput. We are able to retrieve the total time in which a resource is busy by multiplying Ui by T. We can then divide this by the total number of completed requests and get the actual time each resource was busy serving each request.

This yielded the following results:



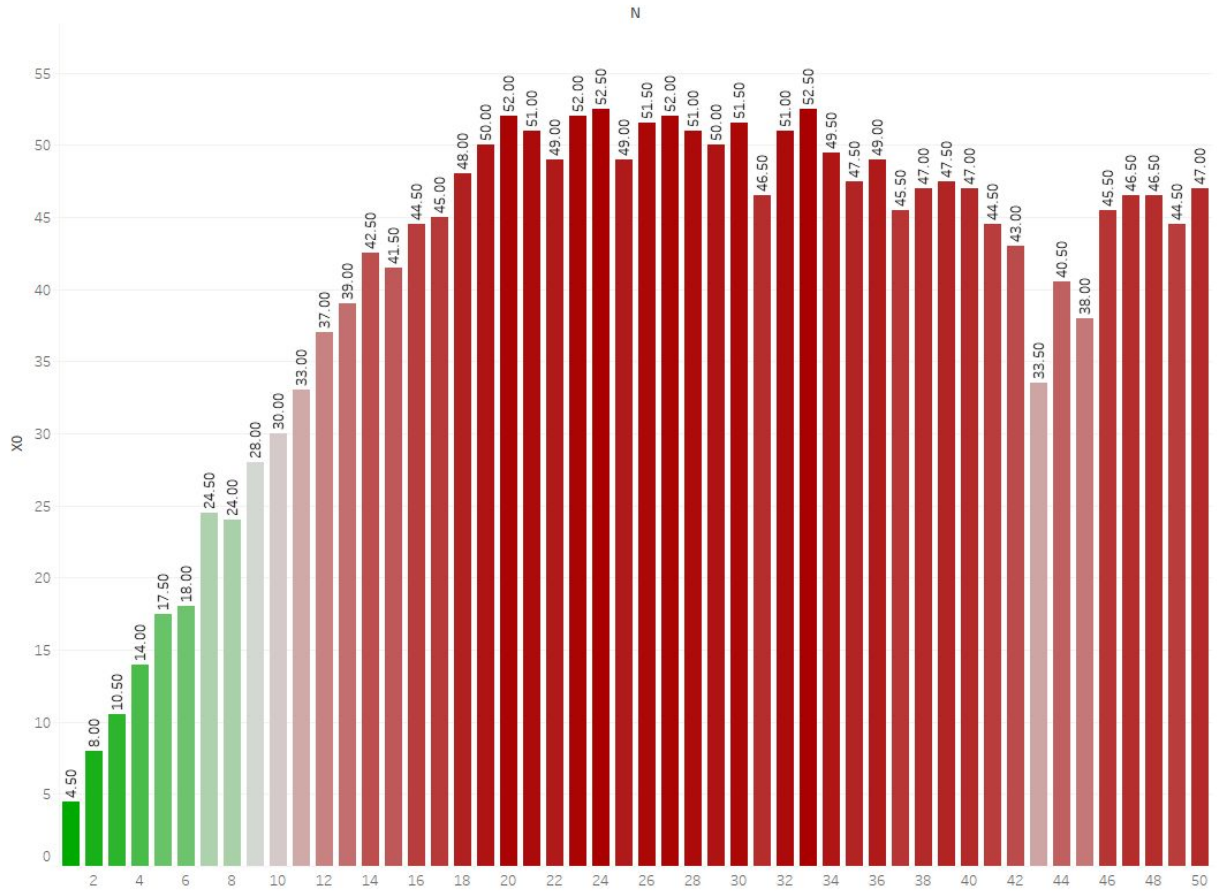
N=1 was the most busy, I guess this was because of the overhead of initially running the tests as this declines over the next few N's before rising again. Because my script timed out at 2 seconds, none of the values go to 2 or over. The values become higher again towards the end as more resources are being utilized. This is what I would have expected when I visualized my chart. To see an initial influx in resource utilization, followed by a slowing down, before finally beginning to rise again.

X0 vs. N

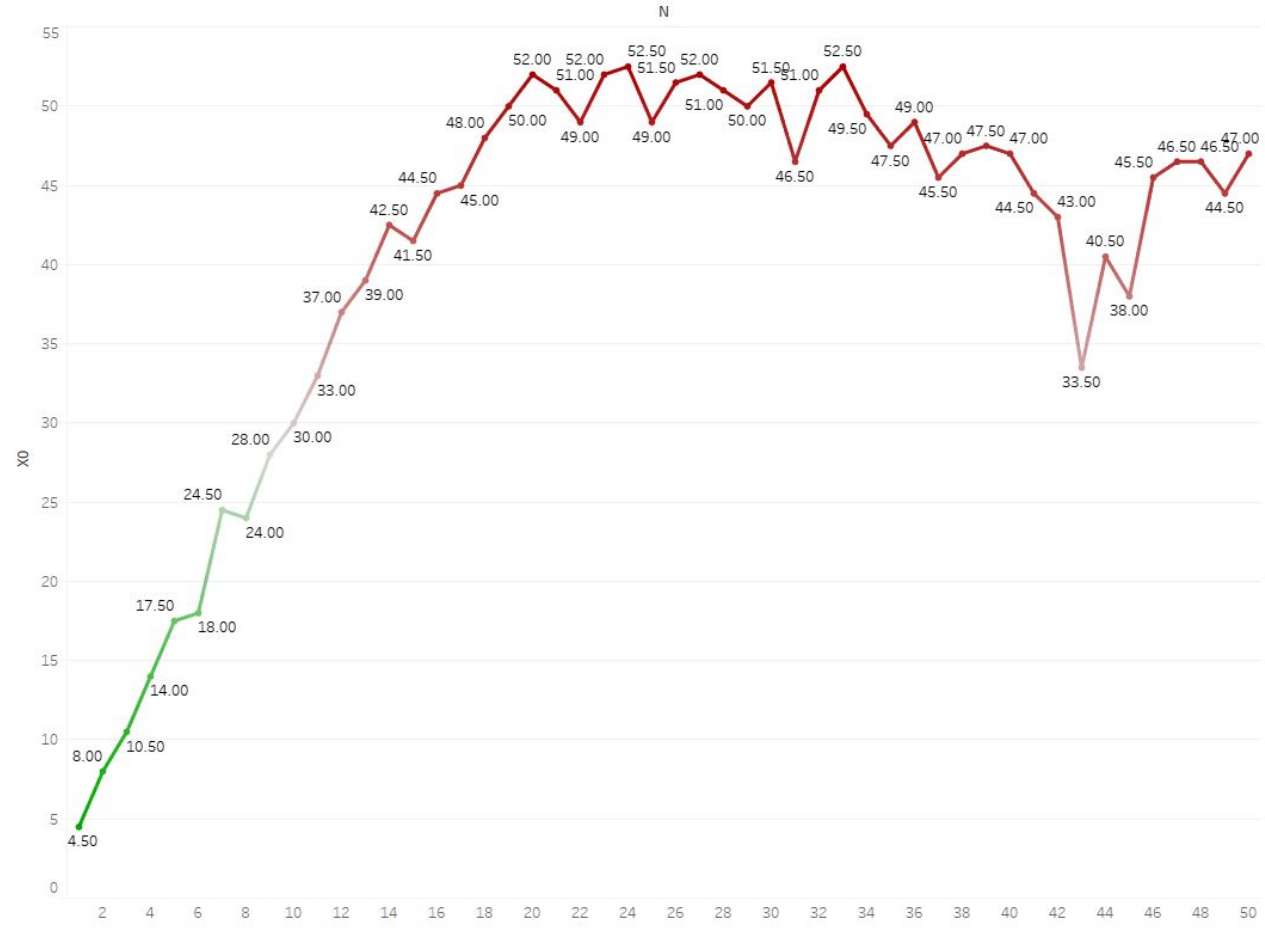
The formula I used for this was:

X0 is used to calculate throughput. In this case we are attempting to calculate the throughput for N.

X0 vs. N



X0 vs. N Line



Throughput measures just how many units of information a system can process during a given amount of time.

Due to the limited resources on the machine, this will severely affect throughput, causing it to rise rapidly, until finally it can no longer rise. We get kind of a bell curve here with this graph, this is possibly due to a freeing up of some of the resources towards the end of my run.

As we can see from the graph, it is clearly taking a lot longer the bigger N gets. This is due to the higher amount of throughput being produced the longer the program is run.

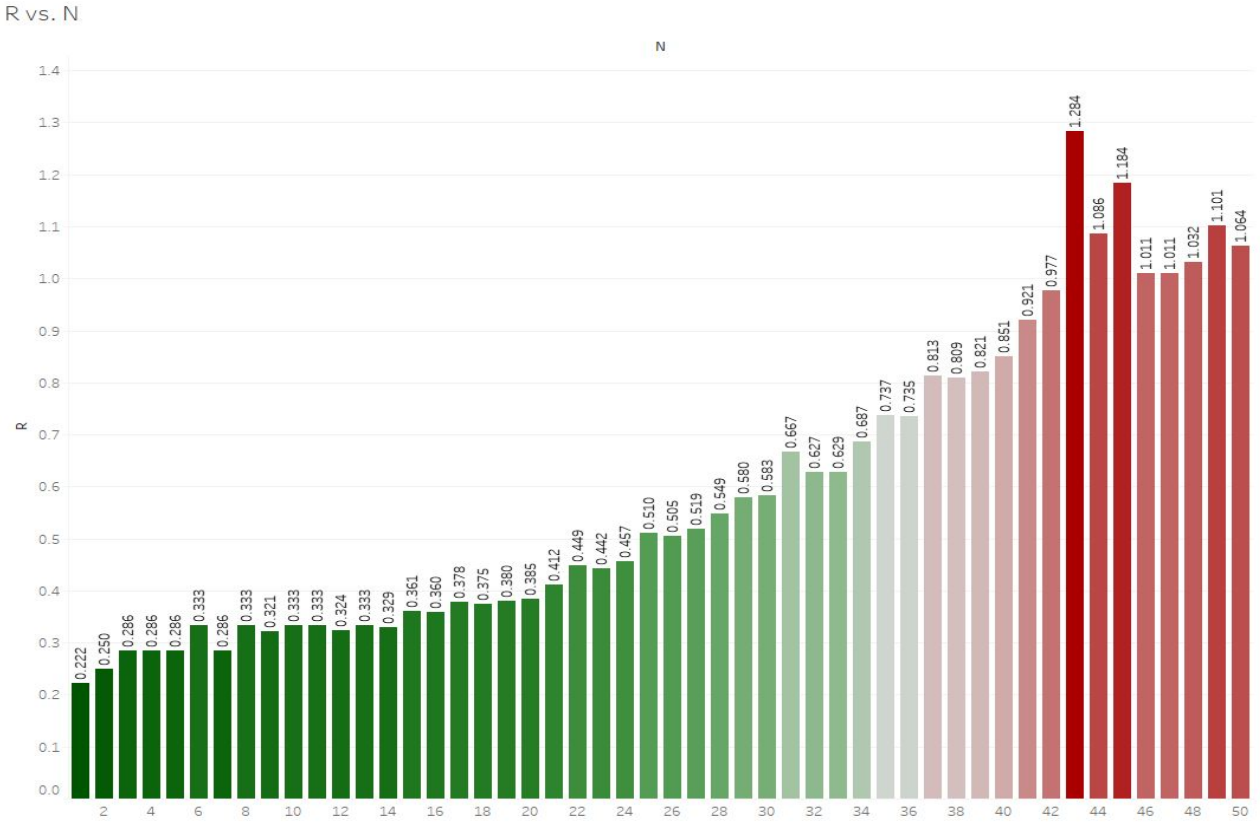
This is likely to have been caused by limiting the VM to a single CPU core. Performance & Capacity management required a network and the nodes to have insufficient processing capability at all times.

The throughput ranges from 4-47, again this is a very large difference from being not utilised almost at all to almost being fully utilised by the loadtest.

R vs. N

The formula I used was R=N/X0

This gave me the following results:



This graph visualizes the response time of database transactions during the specific measurement time. As we can see from the graph, the graph rises with N. This is due to less resources being available and various other factors such as that process being less of a priority.

We can see a gradual rise in the response time of db transactions as the loadtest runs, this is to be expected as the system is being allocated less resources for a larger workload.

Github Repo:

All my code can be found at the following git repo: EPACA2