Ryan Skipp, Alex Garibaldi, Alex Kuhn, and David Katz
Dr. Ngo
CSC496 Spring 2020
8 May 2020

Course Project: Deliverable #3 Technical Paper

In this project, what we are doing is setting up an Alpine Linux image, installing Docker onto that image, installing our testing programs through Docker, and then uploading and running the Alpine image to CloudLab via OpenStack in order to analyze performance in various benchmarks such as disk read/write speed, network speed and bandwidth, and more. These are all valid considerations if we wanted to host a server on CloudLab. We have documented the steps below our group has utilized to do this:

Step 1: Preparing the Alpine Linux Image.

To prepare the image, we created a standard VirtualBox install following Ngo's slides he has posted, only increasing our disk size to 20GB and our RAM size to 8GB.

Step 2: Update and Prepare Alpine Linux

This section is also largely covered by Ngo's slides, up to and including creating a non-root user (in our case, the "student" user) to safely run the benchmarks, as leaving everything running under root is a recipe for disaster once this image is running on CloudLab. Afterwards, the following steps are performed with root privileges:

`apk update` (updates list of programs from repos)
`apk upgrade` (upgrades the installed versions of these programs)
`apk add nano` (installs nano text editor, our personal preference for command line text editing)

Step 3: Installing Docker in Alpine

For our benchmarks, it is necessary to install Docker. To do this, we have to allow access to the community repository of Alpine, and then install the Docker program and ensure its services are running. The steps are as follows, and should be performed with root privileges:

`nano /etc/apk/repositories` (opens up file to edit in nano)
Add the line to the file for the repository we need to add: `http://dl-cdn.alpinelinux.org/alpine/latest-stable/community`
`apk update` (updates list of programs from repos, as we just added a repo)
`apk upgrade` (upgrades your installed versions of these programs)
`apk add docker` (installs docker)
`service docker start` (starts docker run-time service)
`rc-update add docker boot` (starts docker service at boot from now on so it no longer manually needs to be started)
`addgroup student docker` (adds your non-root user to docker group)

`docker run -it ubuntu` (starts up ubuntu in Docker)
`apt-get update` (update list of programs from repo, this time in Ubuntu however)
`apt-get install fio` (installs fio testing benchmark)
`apt-get install nuttcp` (installs netccp testing benchmark)

Then, everything else should work as a non-root user.

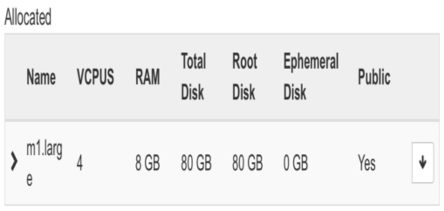
`exit` (log out of ubuntu/Docker)
`docker commit (ID) ubuntu-csc496` (this saves ubuntu image + testing programs)

Step 4: Now, booting up Alpine from scratch, you should be able to:

Login as a non root user, and then execute:
`docker run -it ubuntu-csc496` to start up Docker image
Now, Docker is running and all the benchmarks that we installed are available and ready to be used!

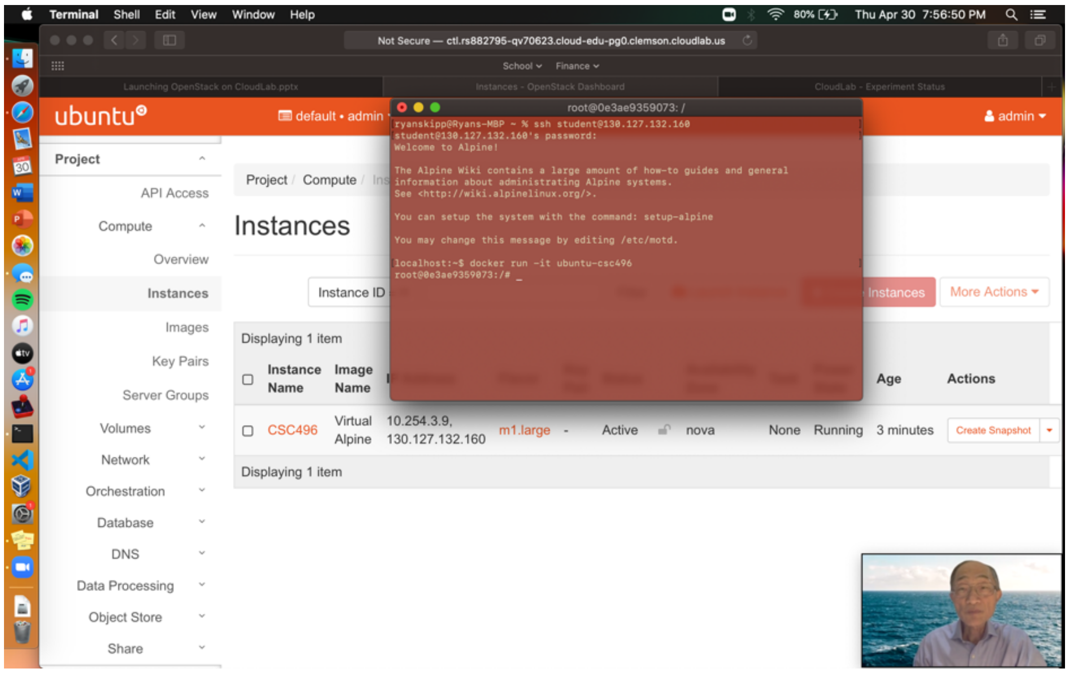
Step 5: Uploading to CloudLab and starting the server

The process for uploading and launching our Alpine server is very similar to what is shown on Ngo's slides. However, we chose the following specs for our server, which we believe to be similar to an "average" web server:



Allocated

	Name	VCPUS	RAM	Total Disk	Root Disk	Ephemeral Disk	Public	
›	m1.large	4	8 GB	80 GB	80 GB	0 GB	Yes	↓



```
ryanskipp@Ryans-MBP ~ % ssh student@130.127.132.160
student@130.127.132.160's password:
Welcome to Alpine!

The Alpine Wiki contains a large amount of how-to guides and general
information about administrating Alpine systems.
See <http://wiki.alpinelinux.org/>.

You can setup the system with the command: setup-alpine

You may change this message by editing /etc/motd.

localhost:~$ docker run -it ubuntu-csc496
root@0e3ae9359073:/# _
```

Instances

Displaying 1 item

	Instance Name	Image Name										Age	Actions
☐	CSC496	Virtual Alpine	10.254.3.9, 130.127.132.160	m1.large	-	Active		nova	None	Running		3 minutes	Create Snapshot

Displaying 1 item

Step 6: Testing using fio

This program is used to test disk writing and reading speed. To document our testing results, the plan is to compare an average of our local, native machine disk read/write speeds compared to the benchmarks running in Docker in order to get an idea of the difference in performance that the default Docker environment gives us.

For reference, the local machine used for benchmark testing is Ryan’s MacBook Pro. His laptop is running the latest macOS 10.15 and the benchmarks run natively. The specs for his machine are below, and can be considered a typical laptop:

CPU	RAM	Disk Space
2.6 GHz Dual Core Intel i5	8GB 1600MHz DDR3	256GB Flash Storage

Test #1: A measure of random write speeds. This command will write a total of 2GB of files [4 jobs x 512 MB = 2GB] through running 4 processes at a time:

```
fio --name=randwrite --ioengine=libaio --iodepth=1 --rw=randwrite
--bs=4k --direct=0 --size=512M --numjobs=4 --runtime=60 --
group_reporting
```

Three results from our CloudLab Server:

```
WRITE: bw=96.1MiB/s (101MB/s), 96.1MiB/s-96.1MiB/s (101MB/s-
101MB/s), io=2048MiB (2147MB), run=21319-21319msec
WRITE: bw=87.5MiB/s (91.8MB/s), 87.5MiB/s-87.5MiB/s (91.8MB/s-
91.8MB/s), io=2048MiB (2147MB), run=23403-23403msec
WRITE: bw=85.4MiB/s (89.5MB/s), 85.4MiB/s-85.4MiB/s (89.5MB/s-
89.5MB/s), io=2048MiB (2147MB), run=23983-23983msec
```

Three results from the personal laptop:

```
WRITE: bw=356MiB/s (374MB/s), 356MiB/s-356MiB/s (374MB/s-
374MB/s), io=2048MiB (2147MB), run=5747-5747msec
WRITE: bw=435MiB/s (456MB/s), 435MiB/s-435MiB/s (456MB/s-
456MB/s), io=2048MiB (2147MB), run=4711-4711msec
WRITE: bw=377MiB/s (396MB/s), 377MiB/s-377MiB/s (396MB/s-
396MB/s), io=2048MiB (2147MB), run=5426-5426msec
```

Test #2: A measure of random read speeds. This command will read a total of 2GB of files through running 4 processes at a time:

```
fio --name=randread --ioengine=libaio --iodepth=16 --rw=randread
--bs=4k --direct=0 --size=512M --numjobs=4 --runtime=240 --
group_reporting
```

Three results from our CloudLab Server:

```
READ: bw=1154KiB/s (1182kB/s), 1154KiB/s-1154KiB/s (1182kB/s-
1182kB/s), io=271MiB (284MB), run=240024-240024msec
READ: bw=1429KiB/s (1464kB/s), 1429KiB/s-1429KiB/s (1464kB/s-
1464kB/s), io=335MiB (351MB), run=240015-240015msec
READ: bw=1424KiB/s (1458kB/s), 1424KiB/s-1424KiB/s (1458kB/s-
1458kB/s), io=334MiB (350MB), run=240014-240014msec
```

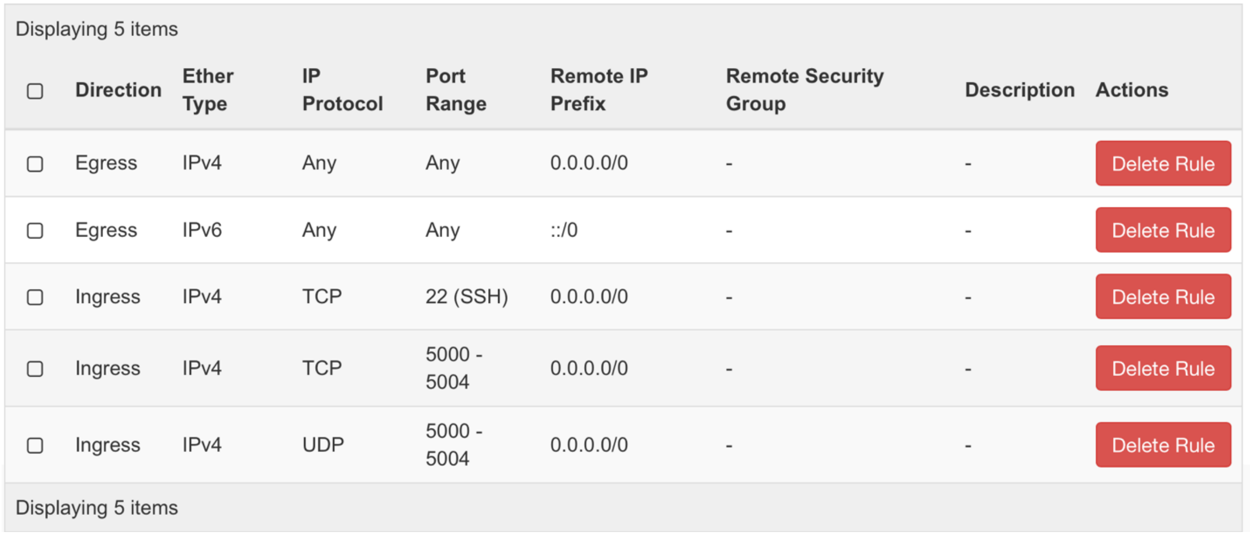
Three results from the personal laptop:

```
READ: bw=79.6MiB/s (83.5MB/s), 79.6MiB/s-79.6MiB/s (83.5MB/s-
83.5MB/s), io=2048MiB (2147MB), run=25725-25725msec
READ: bw=174MiB/s (182MB/s), 174MiB/s-174MiB/s (182MB/s-182MB/s),
io=2048MiB (2147MB), run=11798-11798msec
READ: bw=181MiB/s (190MB/s), 181MiB/s-181MiB/s (190MB/s-190MB/s),
io=2048MiB (2147MB), run=11287-11287msec
```

As can be seen through the data, there is a clear indication that the server is running at significantly slower disk/read write speeds. This is, however, not a reflection on CloudLab. *Most likely this is due to* Docker placing default restrictions on the amount of resources allocated to it compared to the machine running it natively without such restrictions. For future tests, we will try to remove these Docker restrictions.

Step 6: Testing using nuttcp

This program can be used to test network bandwidth. First, we got our server up and running on CloudLab again, ensuring we opened up the ports that nuttcp uses (5000 and 5001), as seen below:

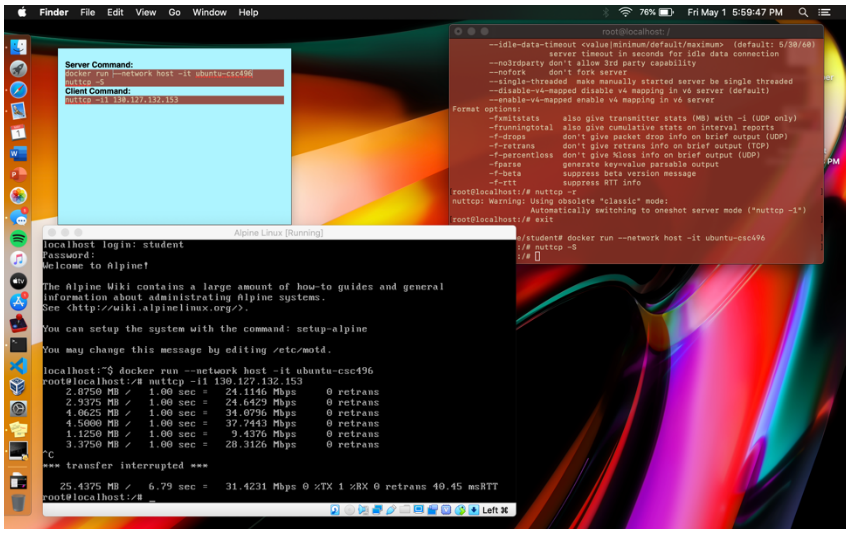


Displaying 5 items

☐	Direction	Ether Type	IP Protocol	Port Range	Remote IP Prefix	Remote Security Group	Description	Actions
☐	Egress	IPv4	Any	Any	0.0.0.0/0	-	-	Delete Rule
☐	Egress	IPv6	Any	Any	::/0	-	-	Delete Rule
☐	Ingress	IPv4	TCP	22 (SSH)	0.0.0.0/0	-	-	Delete Rule
☐	Ingress	IPv4	TCP	5000 - 5004	0.0.0.0/0	-	-	Delete Rule
☐	Ingress	IPv4	UDP	5000 - 5004	0.0.0.0/0	-	-	Delete Rule

Displaying 5 items

It was important to note that for this to work, when starting Docker, you must specify "`—-network host`" so Docker does not create its own unique IP. Then, on the server end, we run the command "`nuttcp -S`" to start listening, and then "`nuttcp -i1 <IP>`" on the client end to determine the bandwidth.



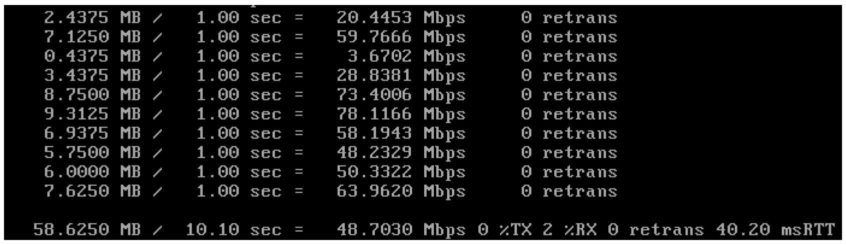
```
Server Command:
docker run --network host -it ubuntu-csc496
nuttcp -S
Client Command:
nuttcp -i1 130.127.132.153
```

```
        --idle-data-timeout <value|minimum/default/maximum>  (default: 5/30/60)
                     server timeout in seconds for idle data connection
        --no3rdparty don't allow 3rd party capability
        --nofork     don't fork server
        --single-threaded  make manually started server be single threaded
        --disable-v4-mapped disable v4 mapping in v6 server (default)
        --enable-v4-mapped enable v4 mapping in v6 server
Format options:
        -fxmitstats     also give transmitter stats (MB) with -i (UDP only)
        -frunningtotal  also give cumulative stats on interval reports
        -f-drops        don't give packet drop info on brief output (UDP)
        -f-retrans      don't give retrans info on brief output (TCP)
        -f-percentloss  don't give %loss info on brief output (UDP)
        -fparse         generate key=value parsable output
        -f-beta         suppress beta version message
        -f-rtt          suppress RTT info
root@localhost:/# nuttcp -r
nuttcp: Warning: Using obsolete "classic" mode:
                 Automatically switching to oneshot server mode ("nuttcp -1")
root@localhost:/# exit

e/student# docker run --network host -it ubuntu-csc496
:/# nuttcp -S
:/# 
```

```
localhost login: student
Password:
Welcome to Alpine!

The Alpine Wiki contains a large amount of how-to guides and general
information about administrating Alpine systems.
See <http://wiki.alpinelinux.org/>.

You can setup the system with the command: setup-alpine

You may change this message by editing /etc/motd.

localhost:~$ docker run --network host -it ubuntu-csc496
root@localhost:/# nuttcp -i1 130.127.132.153
    2.8750 MB /   1.00 sec =   24.1146 Mbps     0 retrans
    2.9375 MB /   1.00 sec =   24.6429 Mbps     0 retrans
    4.0625 MB /   1.00 sec =   34.0796 Mbps     0 retrans
    4.5000 MB /   1.00 sec =   37.7443 Mbps     0 retrans
    1.1250 MB /   1.00 sec =    9.4376 Mbps     0 retrans
    3.3750 MB /   1.00 sec =   28.3126 Mbps     0 retrans
^C
*** transfer interrupted ***

   25.4375 MB /   6.79 sec =   31.4231 Mbps 0 %TX 1 %RX 0 retrans 40.45 msRTT
root@localhost:/# _
```

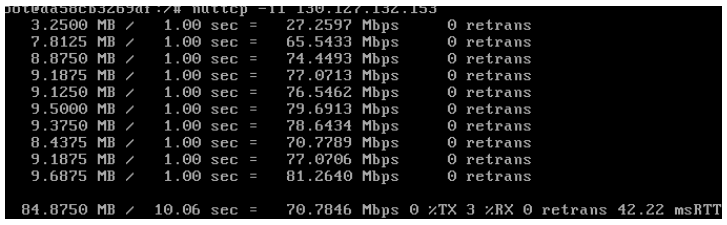
To document our testing results, we had two of our team members connect to the server to show the bandwidth they were getting. While no means entirely scientific, it shows a rough estimate of the kind of network bandwidth you can get while connecting to CloudLab:

Computer #1 Bandwidth Example:



```
    2.4375 MB /   1.00 sec =   20.4453 Mbps     0 retrans
    7.1250 MB /   1.00 sec =   59.7666 Mbps     0 retrans
    0.4375 MB /   1.00 sec =    3.6702 Mbps     0 retrans
    3.4375 MB /   1.00 sec =   28.8381 Mbps     0 retrans
    8.7500 MB /   1.00 sec =   73.4006 Mbps     0 retrans
    9.3125 MB /   1.00 sec =   78.1166 Mbps     0 retrans
    6.9375 MB /   1.00 sec =   58.1943 Mbps     0 retrans
    5.7500 MB /   1.00 sec =   48.2329 Mbps     0 retrans
    6.0000 MB /   1.00 sec =   50.3322 Mbps     0 retrans
    7.6250 MB /   1.00 sec =   63.9620 Mbps     0 retrans

   58.6250 MB /  10.10 sec =   48.7030 Mbps 0 %TX 2 %RX 0 retrans 40.20 msRTT
```

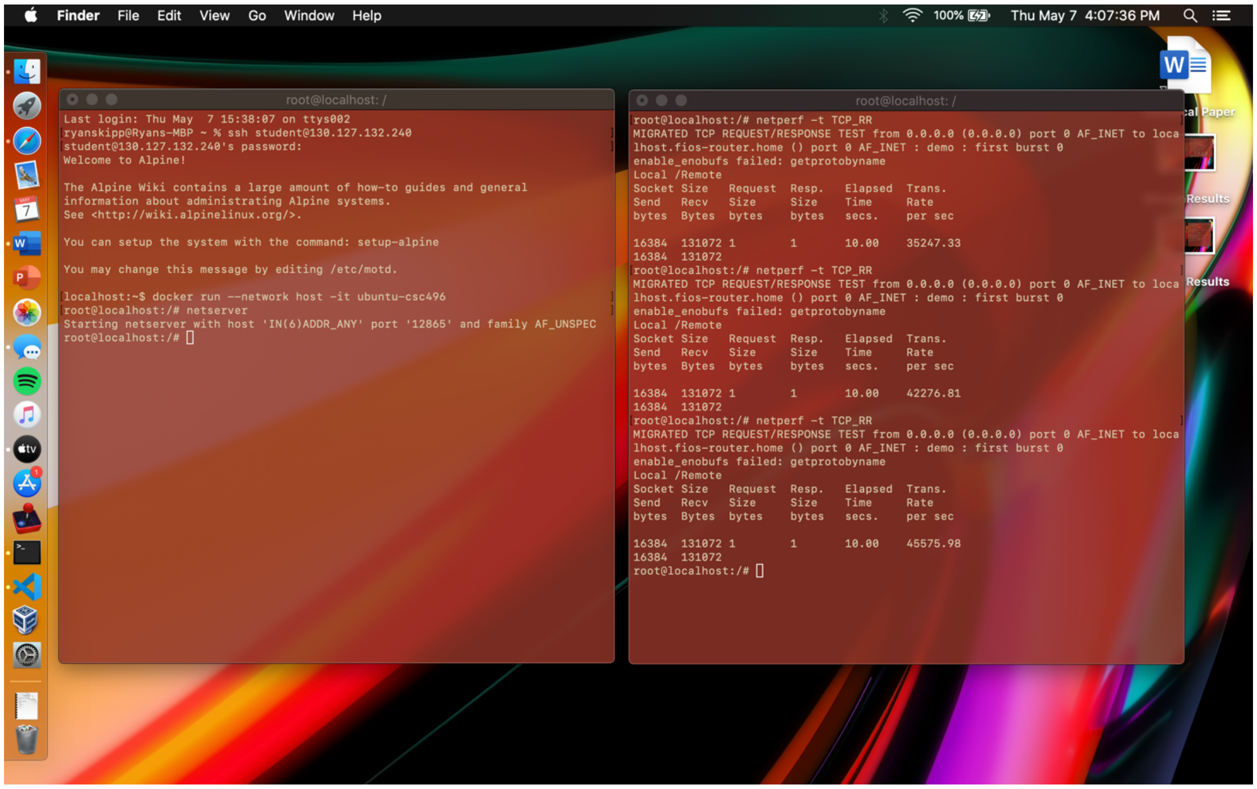
Computer #2 Bandwidth Example:



```
oot@da58cb3269af:/# nuttcp -i1 130.127.132.153
    3.2500 MB /   1.00 sec =   27.2597 Mbps     0 retrans
    7.8125 MB /   1.00 sec =   65.5433 Mbps     0 retrans
    8.8750 MB /   1.00 sec =   74.4493 Mbps     0 retrans
    9.1875 MB /   1.00 sec =   77.0713 Mbps     0 retrans
    9.1250 MB /   1.00 sec =   76.5462 Mbps     0 retrans
    9.5000 MB /   1.00 sec =   79.6913 Mbps     0 retrans
    9.3750 MB /   1.00 sec =   78.6434 Mbps     0 retrans
    8.4375 MB /   1.00 sec =   70.7789 Mbps     0 retrans
    9.1875 MB /   1.00 sec =   77.0706 Mbps     0 retrans
    9.6875 MB /   1.00 sec =   81.2640 Mbps     0 retrans

   84.8750 MB /  10.06 sec =   70.7846 Mbps 0 %TX 3 %RX 0 retrans 42.22 msRTT
```

Step 7: Testing using netperf

We used the program netperf to test the network latency. To do this, we instantiated our CloudLab server with the port 12865 opened using the same process as we did to open ports for nuttcp. Then, on one machine logged onto CloudLab, we started up the netperf server (`netserver`) and on the other we started the client program (`netperf -t TCP_RR`). The following results demonstrate the latency between the nodes:

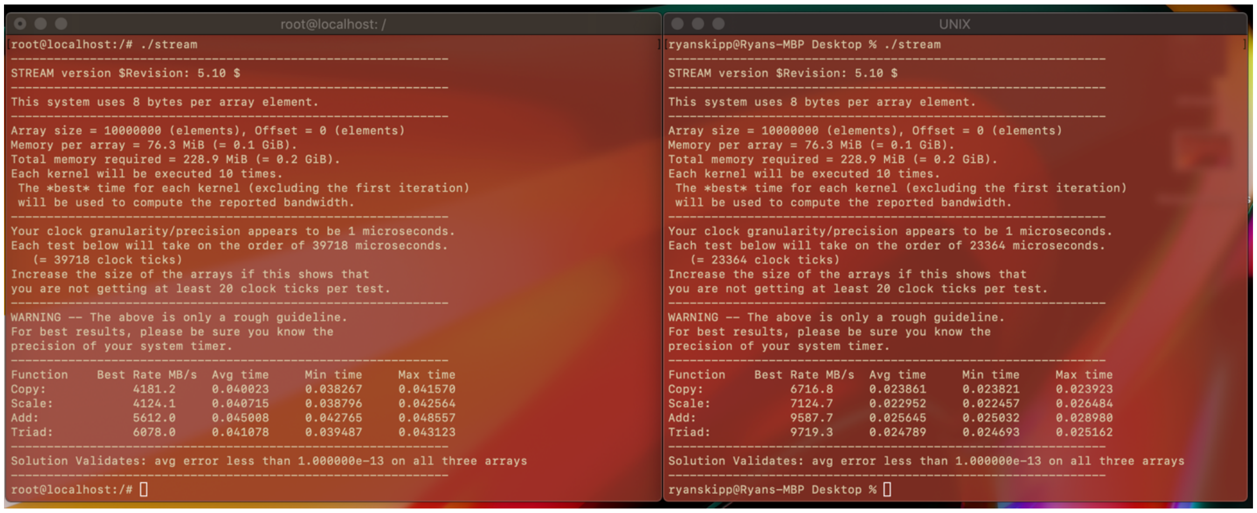


```
Last login: Thu May  7 15:38:07 on ttys002
ryanskipp@Ryans-MBP ~ % ssh student@130.127.132.240
student@130.127.132.240's password:
Welcome to Alpine!

The Alpine Wiki contains a large amount of how-to guides and general
information about administrating Alpine systems.
See <http://wiki.alpinelinux.org/>.

You can setup the system with the command: setup-alpine

You may change this message by editing /etc/motd.

localhost:~$ docker run --network host -it ubuntu-csc496
root@localhost:/# netserver
Starting netserver with host 'IN(6)ADDR_ANY' port '12865' and family AF_UNSPEC
root@localhost:/# 
```

```
root@localhost:/# netperf -t TCP_RR
MIGRATED TCP REQUEST/RESPONSE TEST from 0.0.0.0 (0.0.0.0) port 0 AF_INET to loca
lhost.fios-router.home () port 0 AF_INET : demo : first burst 0
enable_enobufs failed: getprotobyname
Local /Remote
Socket Size   Request  Resp.   Elapsed  Trans.
Send   Recv   Size     Size    Time     Rate
bytes  Bytes  bytes    bytes   secs.    per sec

16384  131072 1        1       10.00    35247.33
16384  131072
root@localhost:/# netperf -t TCP_RR
MIGRATED TCP REQUEST/RESPONSE TEST from 0.0.0.0 (0.0.0.0) port 0 AF_INET to loca
lhost.fios-router.home () port 0 AF_INET : demo : first burst 0
enable_enobufs failed: getprotobyname
Local /Remote
Socket Size   Request  Resp.   Elapsed  Trans.
Send   Recv   Size     Size    Time     Rate
bytes  Bytes  bytes    bytes   secs.    per sec

16384  131072 1        1       10.00    42276.81
16384  131072
root@localhost:/# netperf -t TCP_RR
MIGRATED TCP REQUEST/RESPONSE TEST from 0.0.0.0 (0.0.0.0) port 0 AF_INET to loca
lhost.fios-router.home () port 0 AF_INET : demo : first burst 0
enable_enobufs failed: getprotobyname
Local /Remote
Socket Size   Request  Resp.   Elapsed  Trans.
Send   Recv   Size     Size    Time     Rate
bytes  Bytes  bytes    bytes   secs.    per sec

16384  131072 1        1       10.00    45575.98
16384  131072
root@localhost:/# 
```

Step 7: Testing using stream

Stream is a program used to analyze memory bandwidth. To install the program, we had to build it from the c code found at http://www.cs.virginia.edu/stream/FTP/Code/stream.c and then compile using the command (`gcc stream.c -o stream`) which turns the code into an executable. GCC is not installed by default in Ubuntu through Docker, and neither is a way to download the code, so both wget (to download the file) and gcc (to compile the c code) were installed through the command (`sudo apt-get install gcc wget`). After installation, stream was run on both our CloudLab node as well as the local MacBook Pro machine. The results are seen below:



```
root@localhost:/# ./stream
-------------------------------------------------------------
STREAM version $Revision: 5.10 $
-------------------------------------------------------------
This system uses 8 bytes per array element.
-------------------------------------------------------------
Array size = 10000000 (elements), Offset = 0 (elements)
Memory per array = 76.3 MiB (= 0.1 GiB).
Total memory required = 228.9 MiB (= 0.2 GiB).
Each kernel will be executed 10 times.
 The *best* time for each kernel (excluding the first iteration)
 will be used to compute the reported bandwidth.
-------------------------------------------------------------
Your clock granularity/precision appears to be 1 microseconds.
Each test below will take on the order of 39718 microseconds.
   (= 39718 clock ticks)
Increase the size of the arrays if this shows that
you are not getting at least 20 clock ticks per test.
-------------------------------------------------------------
WARNING -- The above is only a rough guideline.
For best results, please be sure you know the
precision of your system timer.
-------------------------------------------------------------
Function    Best Rate MB/s  Avg time     Min time     Max time
Copy:           4181.2     0.040023     0.038267     0.041570
Scale:          4124.1     0.040715     0.038796     0.042564
Add:            5612.0     0.045008     0.042765     0.048557
Triad:          6078.0     0.041078     0.039487     0.043123
-------------------------------------------------------------
Solution Validates: avg error less than 1.000000e-13 on all three arrays
-------------------------------------------------------------
root@localhost:/# 
```

```
ryanskipp@Ryans-MBP Desktop % ./stream
-------------------------------------------------------------
STREAM version $Revision: 5.10 $
-------------------------------------------------------------
This system uses 8 bytes per array element.
-------------------------------------------------------------
Array size = 10000000 (elements), Offset = 0 (elements)
Memory per array = 76.3 MiB (= 0.1 GiB).
Total memory required = 228.9 MiB (= 0.2 GiB).
Each kernel will be executed 10 times.
 The *best* time for each kernel (excluding the first iteration)
 will be used to compute the reported bandwidth.
-------------------------------------------------------------
Your clock granularity/precision appears to be 1 microseconds.
Each test below will take on the order of 23364 microseconds.
   (= 23364 clock ticks)
Increase the size of the arrays if this shows that
you are not getting at least 20 clock ticks per test.
-------------------------------------------------------------
WARNING -- The above is only a rough guideline.
For best results, please be sure you know the
precision of your system timer.
-------------------------------------------------------------
Function    Best Rate MB/s  Avg time     Min time     Max time
Copy:           6716.8     0.023861     0.023821     0.023923
Scale:          7124.7     0.022952     0.022457     0.026484
Add:            9587.7     0.025645     0.025032     0.028980
Triad:          9719.3     0.024789     0.024693     0.025162
-------------------------------------------------------------
Solution Validates: avg error less than 1.000000e-13 on all three arrays
-------------------------------------------------------------
ryanskipp@Ryans-MBP Desktop % 
```

We expect that, once again, the results that appear on CloudLab, which show about half the performance of the local machine, are a result of the fact that Docker constricts resources by default, and this default was not modified for our testing.

Thoughts and Conclusions:

As we indicated in our Deliverable 2 Analysis, we did accomplish all four of our benchmarks as we expected. We view this as quite the accomplishment, despite our testing and results not truly comparing to the paper we used for inspiration. As you well know, the paper demonstrates the difference in performance benchmarks between native machines, KVM, and Docker. However, due to restrictions limited to time, experience, and hardware, we figured that the next best thing would be to install these benchmarks and at least demonstrate their use to the average user, as in answering questions such as “how can I measure my disk speed?” or “how can I measure my network bandwidth?”

Additionally, without time and experience restrictions I believe we could have automated the entire process, utilizing a Dockerfile to configure our image and a bash script to collect data from multiple runs of each benchmark automatically. As stated above though, we do believe our findings are of at least some merit and demonstrate the group’s ability to both install, deploy, and utilize Docker on a CloudLab instance.