Google Cloud Platform

ID2210 Project Report

Group 1

Xin Ren, Bernardo Gonzalez Riede, Adisu Wagaw Terefe

31/05/2018

https://github.com/nathanrenxin/ID2210-project

Content

Introduction 2

1 Obligatory Tasks 2

Task 1 - External Storage support 2

Available storage options 2

Google Cloud Storage API 3

Benchmark 4

Task 2 - Setting up a VM 6

Creation of VM through (web) console 6

Remote Creation API 7

Configuration Changes for TCP/UDP 8

Task 3 - Internal storage support 9

Accessing Google Cloud Storage from VM 9

Cloud Storage benchmarks 10

Cloud Storage 12

Local SSD 17

Task 4 - Simple Network 19

2 Bonus Tasks 25

Task 5 - Custom 25

Standard Parameter benchmarks 26

Multi stream LEDBAT 28

Modifying parameters 30

Ledbat Kompics implementation 32

How to Run Our Code 33

References 34

Introduction

These days, cloud computing is part of any IT company, be it giants like Google, Facebook, Spotify and even start-ups.Two of the main benefits of the cloud is that it is managed and you can request resources on demand. As such many companies prefer to invest all their resources on their specific software products and not on maintaining the infrastructure for the product.

It is thus important to get contact and understand the different technologies behind cloud infrastructure. This project, part of the ID2210 - Distributed Computing, Peer-to-Peer and GRIDS, focuses on components of the Google Cloud Platform(GCP) such as compute, storage and network. We examined these components, deployed the necessary architecture and benchmarked their performance.

1 Obligatory Tasks

For the Google Cloud project, task 1 to 4 were obligatory.

Task 1 - External Storage support

Available storage options

The available storage options for Google Cloud are shown in table 1.

Storage & Database	Limitations	Characteristics
Google Cloud Storage	• Does not provide Mobile SDKs • Unstructured data	• Single API across storage classes • Scalable to exabytes of data • Designed for 99.999999999% durability • Very high availability across all storage classes • Time to first byte in milliseconds • Strongly consistent listing • Zero carbon emissions
Cloud SQL	• Only for rational structured data • Not for analytical workload	• Scalability • High Performance • Integrated • Simple & Fully Managed

	• Does not support horizontal scalability	• Reliability & Security
Cloud Bigtable	• No consistency guarantees for multi-row updates or cross-table updates • Does not support SQL queries or joins • Not a good solution for small amounts of data (< 1 TB)	• High Performance • Security & Permissions • Low Latency • Fully Managed • Redundant Autoscaling • Seamless Cluster Resizing • HBase Compatible • Global Availability • High Durability
Cloud Spanner	• Expensive • Higher latencies • Not ideal for high write-throughput use-cases • Write API is not strictly SQL • Vendor lock-in	• Global Scale • Fully Managed • Relational Semantics • Multi-Language Support • Transactional Consistency • Enterprise Grade Security • Highly Available
Cloud Datastore	• Expensive • Slower writing data due to synchronous replication	• Rich Admin Dashboard • Multiple Access Methods • Fully Managed • Diverse Data Types • ACID Transactions • Fast & Highly Scalable • Easy to Use Query Language • Simple & Integrated

Table 1: Google Cloud storage options

Google Cloud Storage API

Google Cloud Storage can be accessed using gsutil tool, Cloud Storage Client Libraries, XML API, JSON API and Storage Transfer Service.

The gsutil is a Python application that lets user access Cloud Storage from the command line, and it supports a wide range of bucket and object management tasks, including Creating and deleting buckets, Uploading, downloading and deleting objects, listing buckets and objects, moving, copying and renaming objects, editing object and bucket ACLs.

Cloud Storage Client Libraries provide interface to access the Storage via C#, GO, JAVA, NODE.JS, PHP, PYTHON and RUBY program.

XML API is a RESTful interface that lets user manage Cloud Storage data in a programmatic way. Access to Cloud Storage through the XML API is useful when users are using tools and libraries that must work across different storage providers, or when users are migrating from another storage provider to Cloud Storage. In the latter case, users only need to make a few simple changes to their existing tools and libraries to begin sending requests to Cloud Storage.

JSON API is a simple, JSON-backed interface for accessing and manipulating Cloud Storage projects in a programmatic way. It is fully compatible with the Cloud Storage Client Libraries and intended for software developers familiar with web programming and be comfortable creating applications that consume web services through HTTP requests.

Storage Transfer Service can be used to quickly import online data into Cloud Storage and transfer data within Cloud Storage, from one bucket to another. The service can be used on console, with Google API Client Library in JAVA and PYTHON, and with Storage Transfer Service API. It is more recommended than gsutil when transferring data from another cloud storage provider.

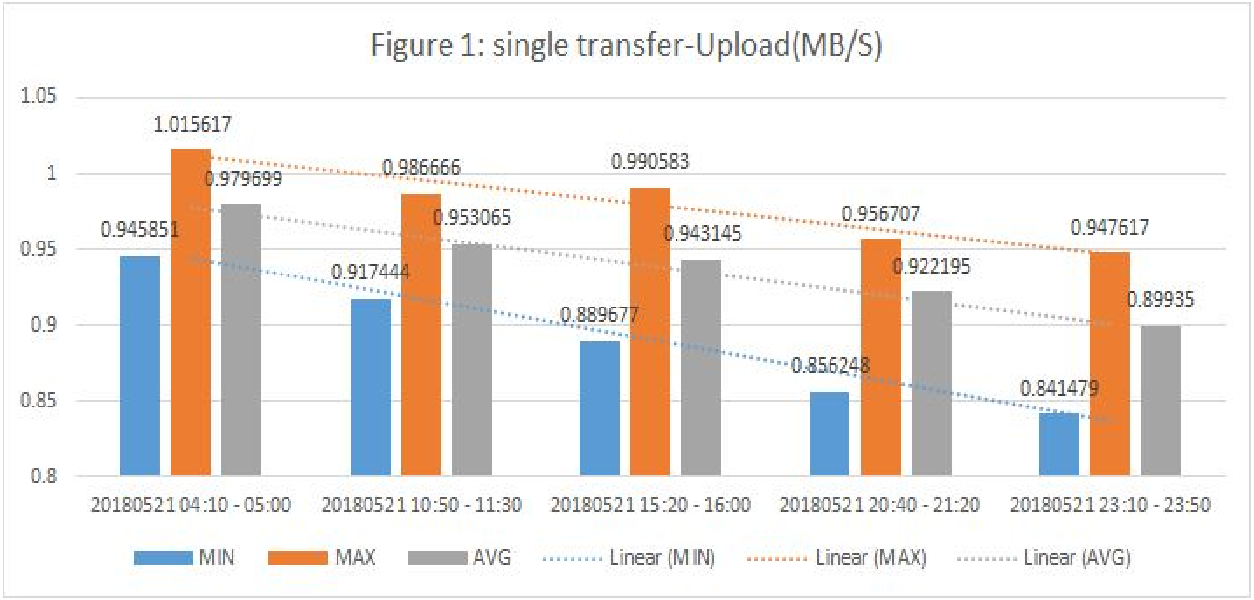
All APIs requires authentication before accessing the Storage for most of the operations. Cloud Storage uses OAuth 2.0 for API authentication and authorization.

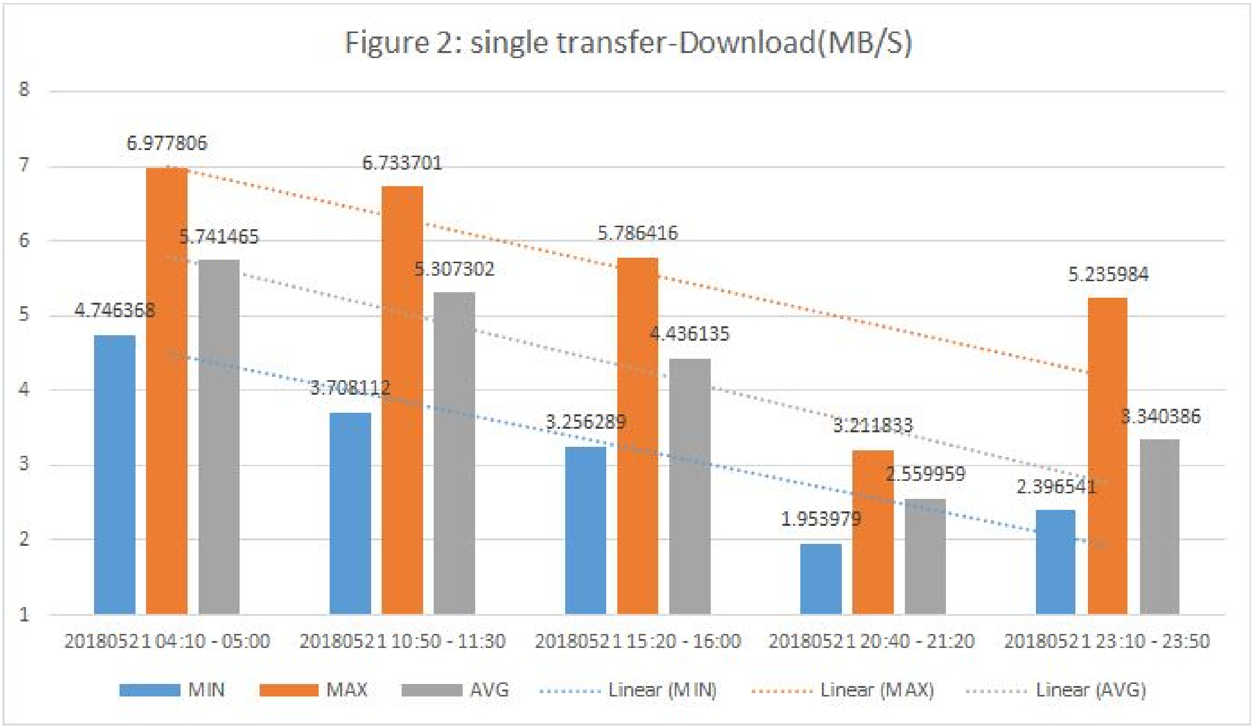
What's more, Cloud Storage can be integrated with Google Cloud Platform Services and Tools, for example, mounted as a file system on a virtual machine instance in Compute Engine.

Benchmark

We developed Java code to create bucket, upload file, download file, remove file, and remove bucket with Cloud Storage Client Libraries. It supports single transfer and parallel transfer (both no interval and 5s interval) base on the arguments users give.

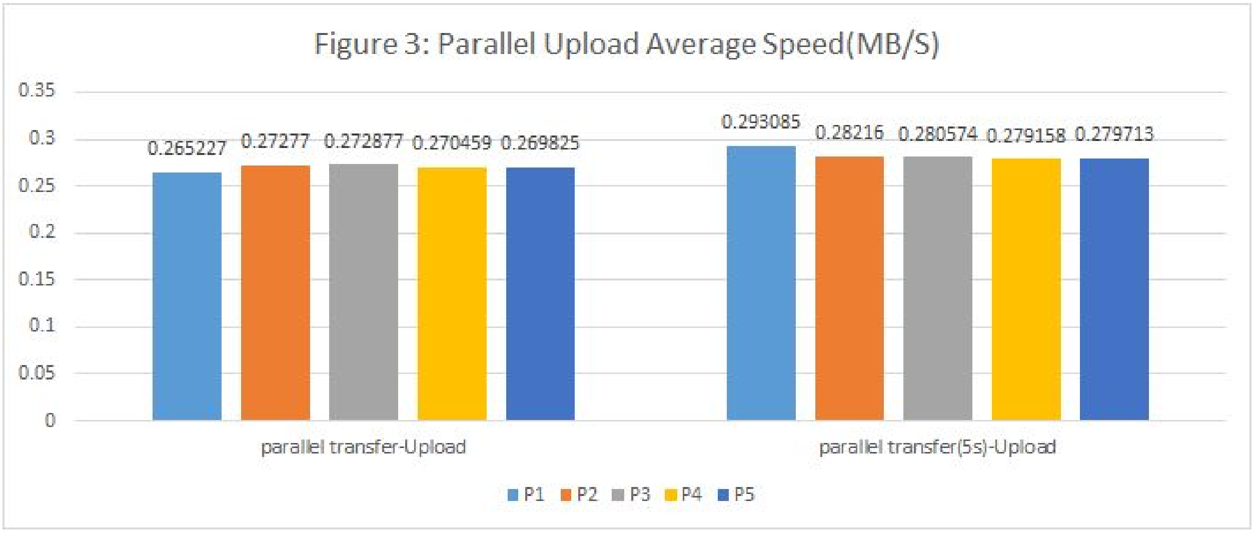
We did single transfer benchmark test at 5 different time periods of a day. And for each period, 10 repeats were done. Here is our results.

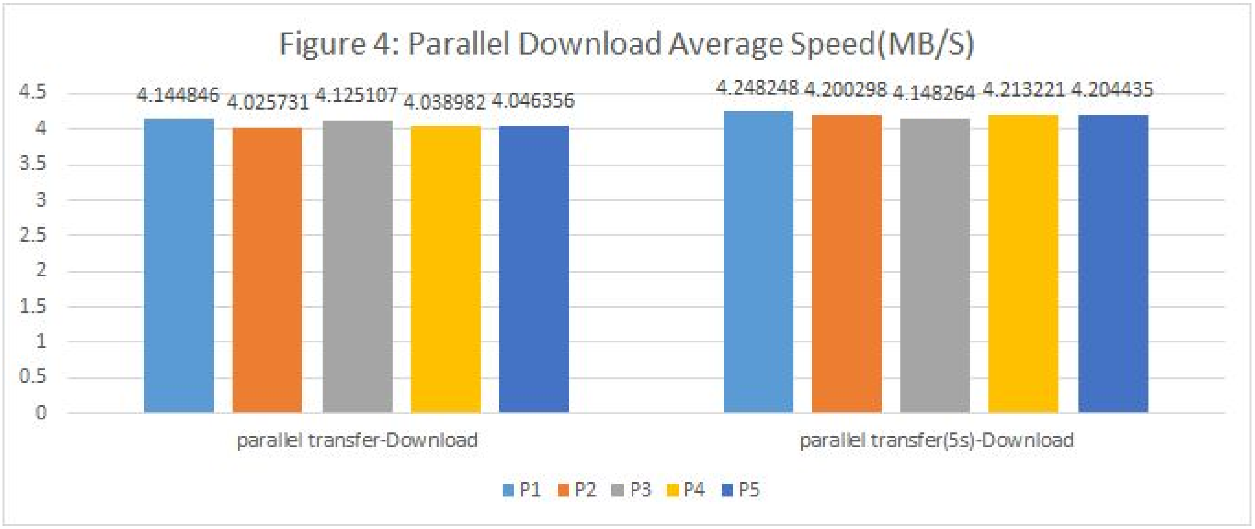




As we can see that the speed goes down from morning to evening and the rush hours are in the evening.

For parallel transfers, we did 20 repeats in which all 5 processes started uploading/downloading at the same time and another 20 repeats in which there was 5s interval between processes. And here is the average speeds of each process.





We can see that the sum of speeds of parallel transfer processes are higher than the speed of single transfer, which means transferring multiple files at a time increases throughput. We did not notice any newcomer advantage or disadvantage.

Task 2 - Setting up a VM

Creation of VM through (web) console

First step was we logged in to our Google cloud console and create a new project with any name we want. (As a matter of chance we were able to create two different projects).

To create an instance from a public image we have three different options.

1. Create using Google Cloud Console.
2. Create using gcloud command line tool.
3. Create using Compute Engine APIs (REST and client APIs)

But in this subtask, we will only focus on the first option. There are a few steps to create a VM using google cloud console. (we listed the steps as an instruction that will be used by others).

1. In the GCP Console, go to the VM Instances page.
2. Select your project and click **Continue**.
3. Click the **Create instance** button.
4. Specify a **Name** for your instance.
5. Optionally, change the **Zone** for this instance.
6. Select a **Machine type** for your instance.
7. In the **Boot disk** section, click **Change** to configure your boot disk. Create a boot disk no larger than 2TB to account for the limitations of MBR partitions.
8. In the **OS images** tab, choose an image.
9. Click **Select**.
10. To add secondary non-root disks to your VM instance:
    a. Click on the **Management, disks, networking, SSH keys**.
    b. Select the **Disks** tab.
    c. Under **Additional disks** click **Add item**.
    d. Specify a disk **Name, Mode**, and set the **When deleting instance** option.
    e. Add additional disks as needed.
11. Click the **Create** button to create and start the instance.

Remote Creation API

There are several ways to create an instance remotely.
But, to summarize the available options we can classify them as follows.

1. Using REST API

   In this API, we will construct a POST request to the instances URI with the same request body. We can add up to 15 secondary non-boot disks at the time we create a VM instance by using the **initializeParam** property for each additional disk. Create additional disks with a public or a private image. To add blank disks, we do not need to specify an image source. Optionally, we can include the **diskSizeGb** and **diskType** properties.

2. Using client libraries.

   Client libraries provide better language integration, improved security, and support for making calls that require user authorization.
   The available client libraries are categorized as

   1) community libraries
      a) Libcloud
      b) Jclouds
      c) Fog.io
   2) Google cloud client library
      a) Google Cloud Node.js Client Library
   3) Google API client libraries
      a) Google APIs Java Client Library
      b) Google APIs JavaScript Client Library
      c) Google APIs Ruby Client Library
      d) Google APIs Node.js Client Library
      e) Google API Objective C Client Library
      f) Google API PHP Client Library
      g) Google API Python Client Library

We picked the java client library and tested it in creating a new instance and listing the available instances.

Configuration Changes for TCP/UDP

In this section, we will explain how we were able to access the VM from our own laptop. The first step was to define the firewall rule that will be used by TCP/UDP connections. This can be done in several ways, but we did it by using gcloud tool. For TCP

```
gcloud compute firewall-rules create default-allow-tcp-8080\

    --network default \

    --action allow \

    --direction ingress \

    --rules tcp:8080 \

    --source-ranges 0.0.0.0/0 \

    --priority 1000 \
```

And for UDP

```
gcloud compute firewall-rules create allow-udp\

    --network default \

    --action allow \

    --direction ingress \

    --rules udp:8181 \

    --source-ranges 0.0.0.0/0 \

    --priority 1000 \
```

After defining these rules, we created 4 different java classes, 2 for TCP(Client and Server) and 2 for UDP(Client and Server) communications. Then we run the server codes on the VM after installing the jdk on the specific instances. From our laptop we run the client codes by providing the external IP of the VM on the specific ports. (8080 for tcp client and 8181 for the udp).

Task 3 - Internal storage support

Accessing Google Cloud Storage from VM

VM can access Cloud Storage by using all the APIs mentioned in task 1. And by using

Cloud Storage URI, VM can export an image to Cloud Storage and use a startup script stored in Cloud Storage. What's more, VM can mount a bucket on Cloud Storage as a file system with Google Cloud Storage FUSE tool. The mounted bucket behaves similarly to a persistent disk even though Cloud Storage buckets are object storage.

The FUSE has been tested successfully with Linux (minimum kernel version 3.10) and OS X (minimum version 10.10.2). Cloud Storage FUSE works by translating object storage names into a file and directory system, interpreting the "/" character in object names as a directory separator so that objects with the same common prefix are treated as files in the same directory. Applications can interact with the mounted bucket like a simple file system, providing virtually limitless file storage running in the cloud. While Cloud Storage FUSE has a file system interface, it is not like an NFS or CIFS file system on the backend.

Cloud Storage FUSE retains the same fundamental characteristics of Cloud Storage, preserving the scalability of Cloud Storage in terms of size and aggregate performance while maintaining the same latency and single object performance. As with the other access methods, Cloud Storage does not support concurrency and locking. For example, if multiple Cloud Storage FUSE clients are writing to the same file, the last flush wins.

Cloud Storage FUSE has much higher latency than a local file system. As such, throughput may be reduced when reading or writing one small file at a time. Using larger files and/or transferring multiple files at a time will help to increase throughput. Individual I/O streams run approximately as fast as gsutil. Small random reads are slow due to latency to first byte. Random writes are done by reading in the whole blob, editing it locally, and writing the whole modified blob back to Cloud Storage. Small writes to large files work as expected, but are slow and expensive.

After installing and setting up credentials for Cloud Storage FUSE, users can mount a bucket to a VM simply by "gcsfuse example-bucket /path/to/mount".

Cloud Storage benchmarks

We ran following benchmark test cases for external storage and local SSD.

ID	External Storage	Local SSD
1	write_bandwidth_test	write_bandwidth_test
2	write_iops_test	write_iops_test
3	write_latency_test	write_latency_test
4	read_bandwidth_test	read_bandwidth_test
5	read_iops_test	read_iops_test
6	read_latency_test	read_latency_test
7	write_bandwidth_test_seq	write_bandwidth_test_seq
8	write_iops_test_seq	write_iops_test_seq
9	write_latency_test_seq	write_latency_test_seq
10	read_bandwidth_test_seq	read_bandwidth_test_seq
11	read_iops_test_seq	read_iops_test_seq
12	read_latency_test_seq	read_latency_test_seq
13	write_bandwidth_test_pt_seq	write_bandwidth_test_pt
14	write_iops_test_pt_seq	write_iops_test_pt
15	write_latency_test_pt_seq	write_latency_test_pt
16	read_bandwidth_test_pt_seq	read_bandwidth_test_pt
17	read_iops_test_pt_seq	read_iops_test_pt
18	read_latency_test_pt_seq	read_latency_test_pt

Table 2: Test Cases ("pt" means parallel transfer with 4 processes, otherwise it is single transfer. "seq" means it is sequential read/write, otherwise it is random read/write)

For External Storage test cases, we used Ubuntu 16.04 LTS with 4 vCPU, 15 GB memory and 10G SSD and executed the test cases in 3 regions.

VM Location	Storage Location	Different Time Periods of a day	Repetitions per period
europe-west2-a	EUROPE-WEST2	3	9
asia-northeast1-a	ASIA-NORTHEAST1	3	9
us-central1-a	US-CENTRAL1	3	9

Table 3: External Storage Benchmark Configurations

For Local SSD test cases, we executed them with following different configurations of Ubuntu 16.04 LTS.

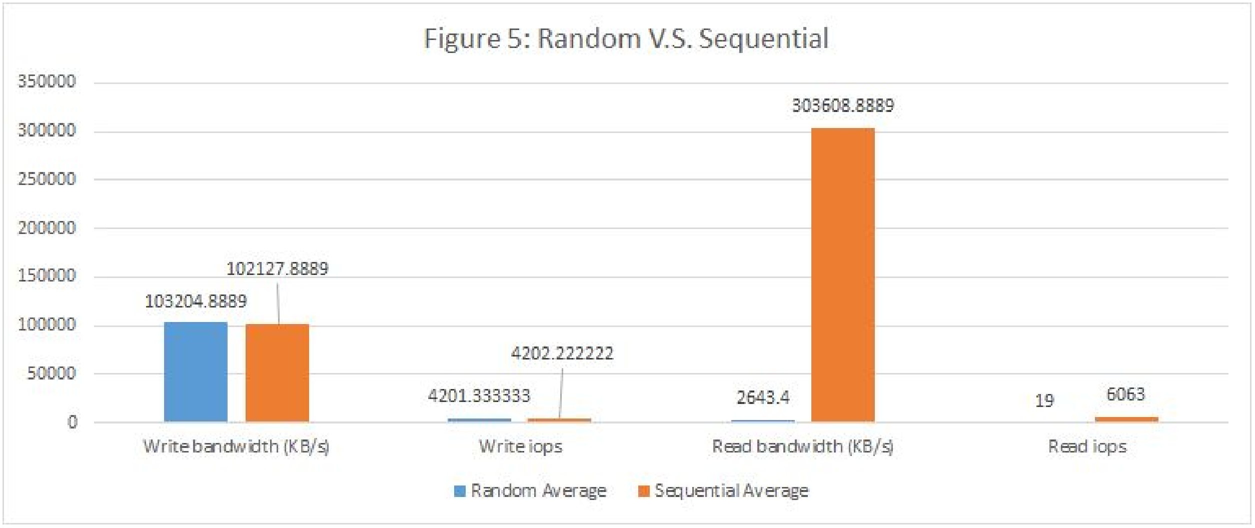
SSD Size	Number of vCPU	Memory	Different Time Periods of a day	Repetitions per period
500G	16	60G	3	3
834G	32	120G	3	3
2048G	64	240G	3	3
2048G	96	360G	3	3

Table 4: Local SSD Benchmark Configurations

In total, 2106 cases were executed.

Cloud Storage

1. The performance of random write is similar to that of sequential write, but the performance of sequential read is much higher than that of random read.



Data from EU at 2018.0525.0940

2. No obvious difference on performance observed at different times of a day. We suspect that since both VM and storage are in the same region, the connections were established in the internal network. (Data from single sequential transfer in EU region at different times of a day)

Figure 6: Sequential Write Bandwidth(KB/S)

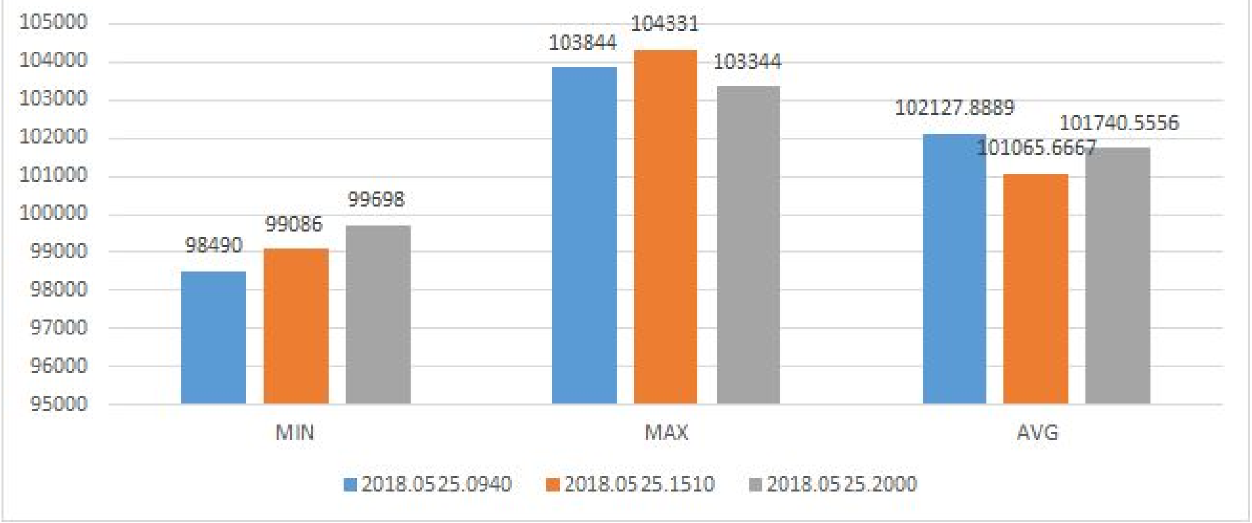
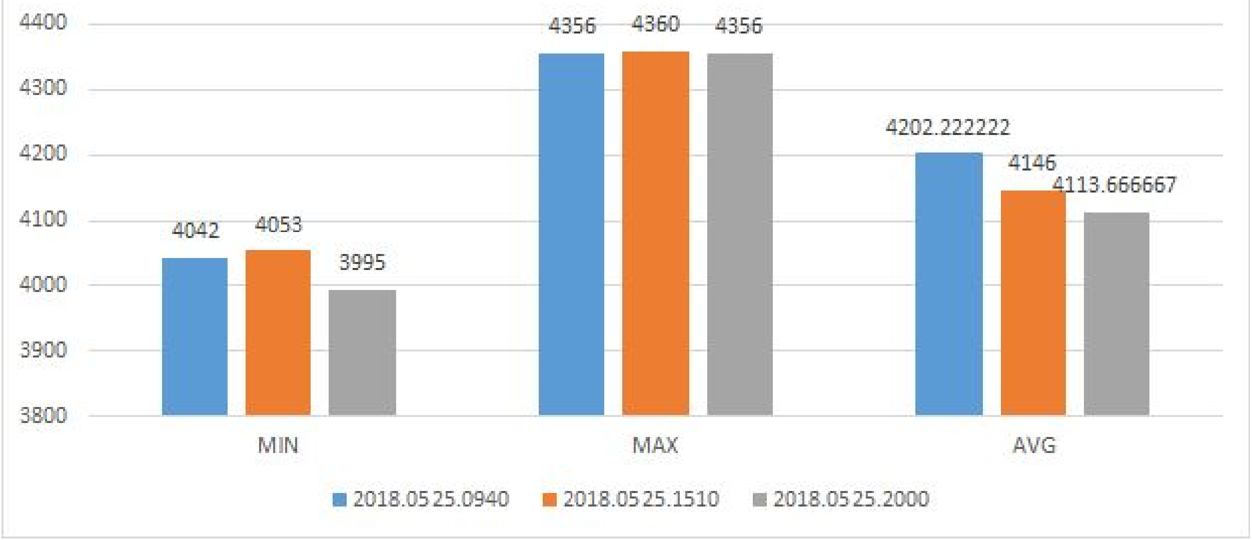
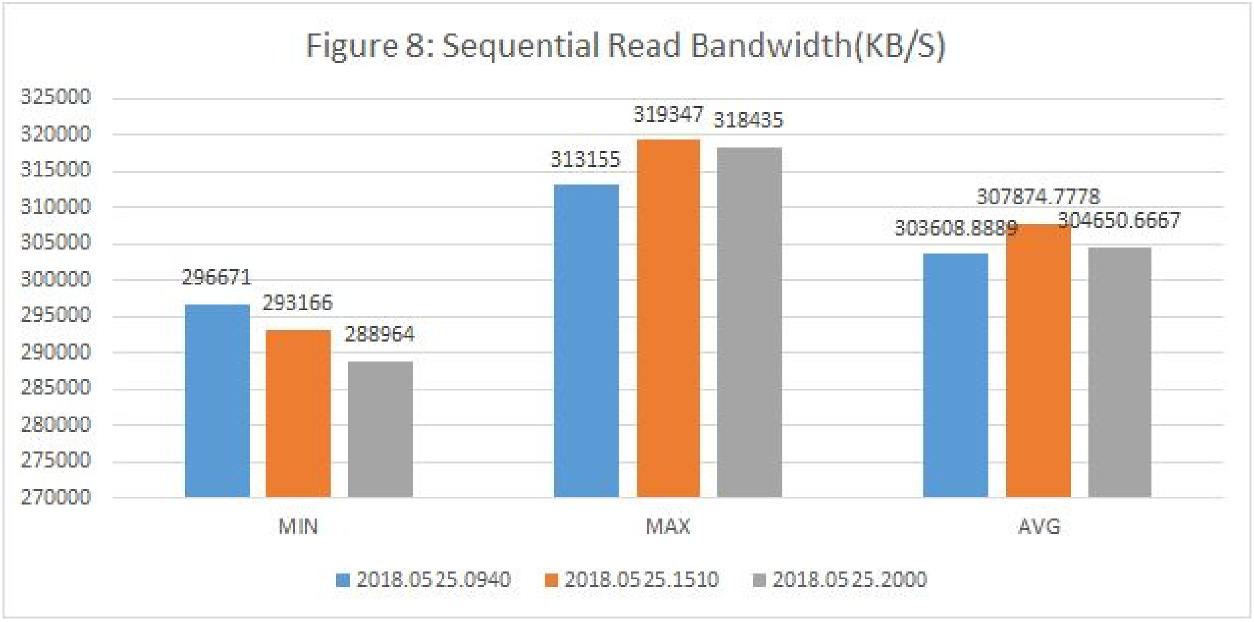
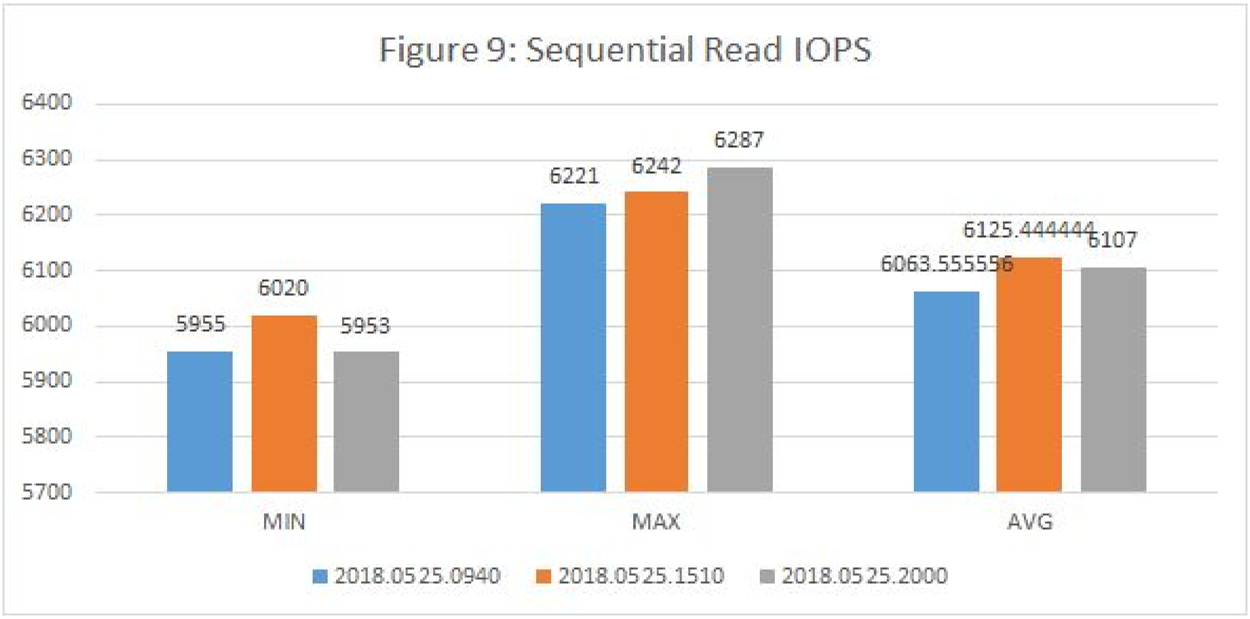


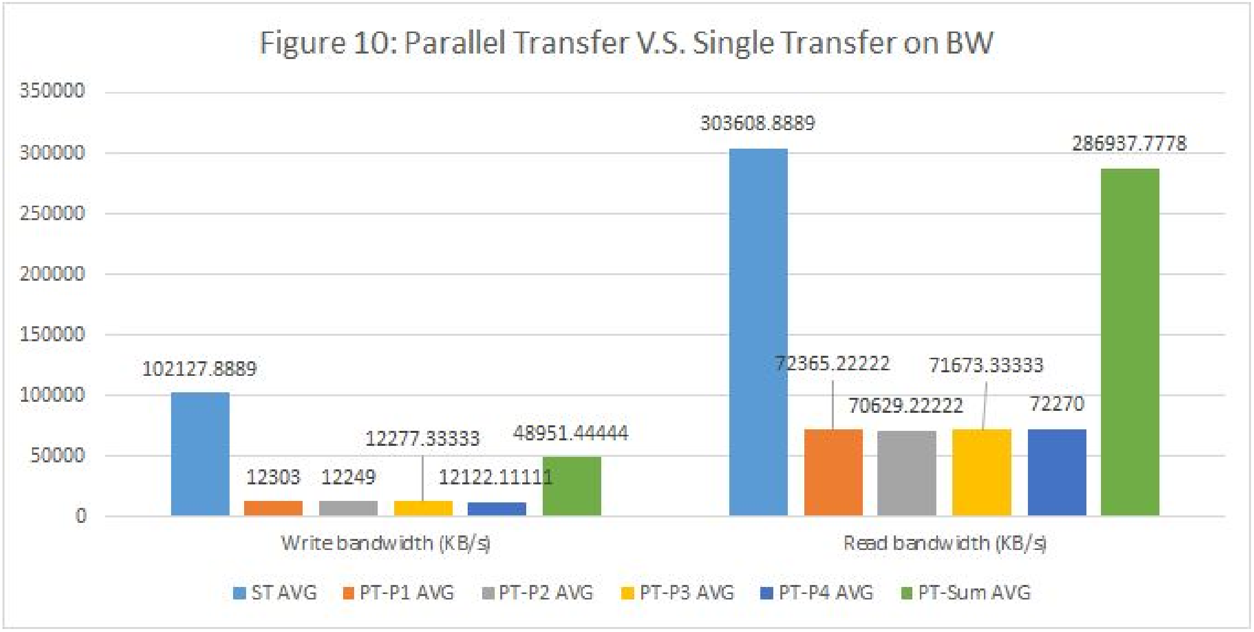
Figure 7: Sequential Write IOPS

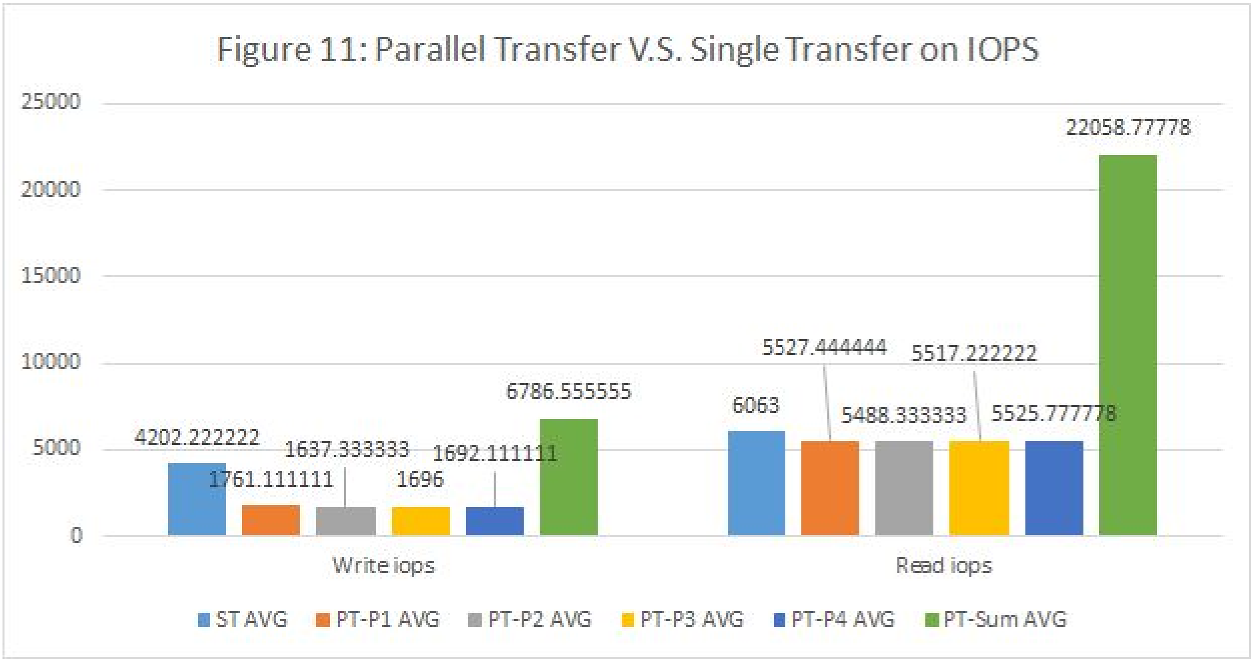






3. Parallel transfer processes share similar performance and transferring multiple files at a time helps to increase overall IOPS. (Data from EU at 2018.0525.0940)





4. EU has the best performance and US has the worst/most unstable performance (Data from all 3 different time periods).

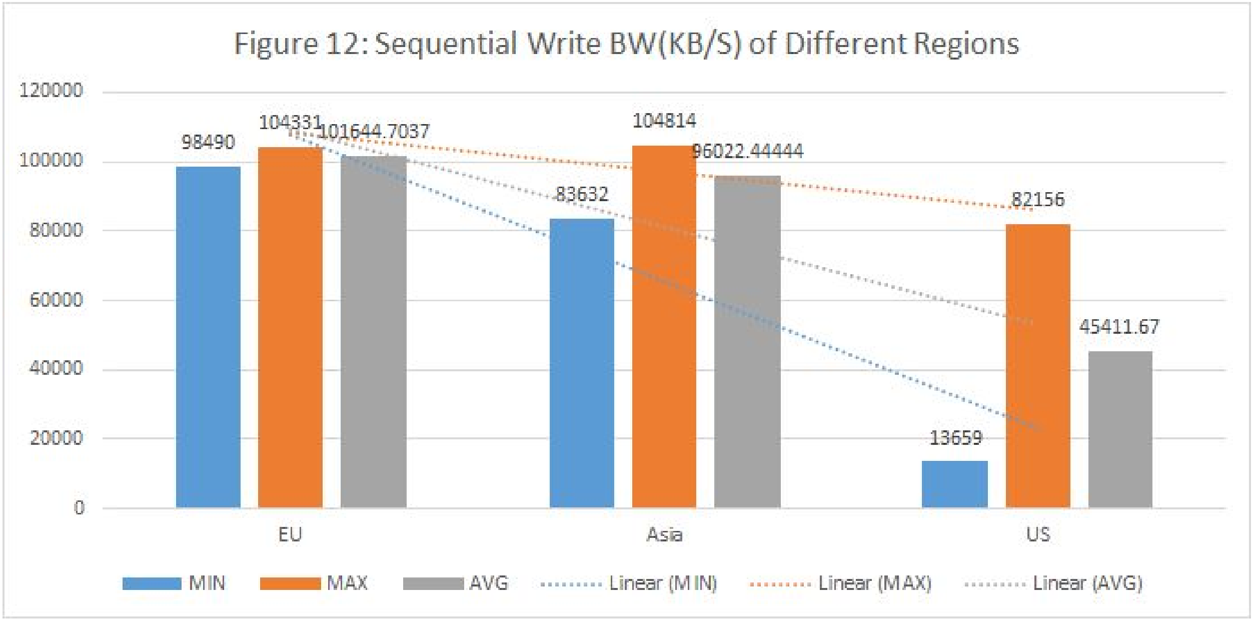


Figure 12: Sequential Write BW(KB/S) of Different Regions
120000
100000
80000
60000
40000
20000
0
98490
104331
101644.7037
83632
104814
96022.44444
13659
82156
45411.67
EU
Asia
US
MIN
MAX
AVG
Linear (MIN)
Linear (MAX)
Linear (AVG)

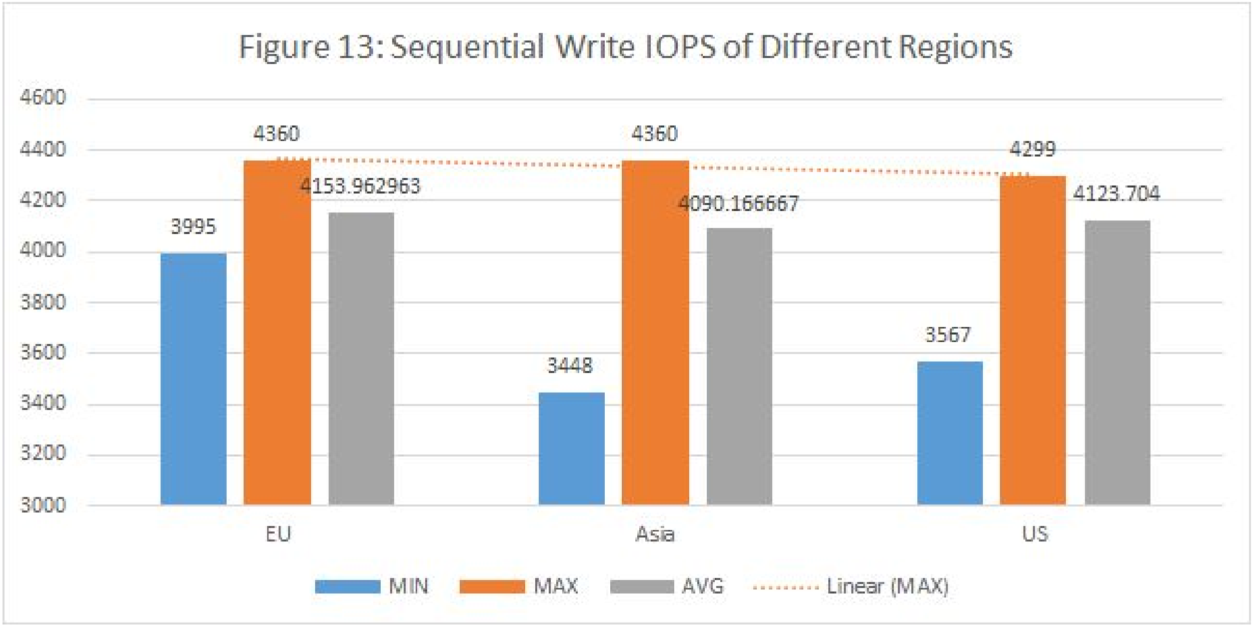
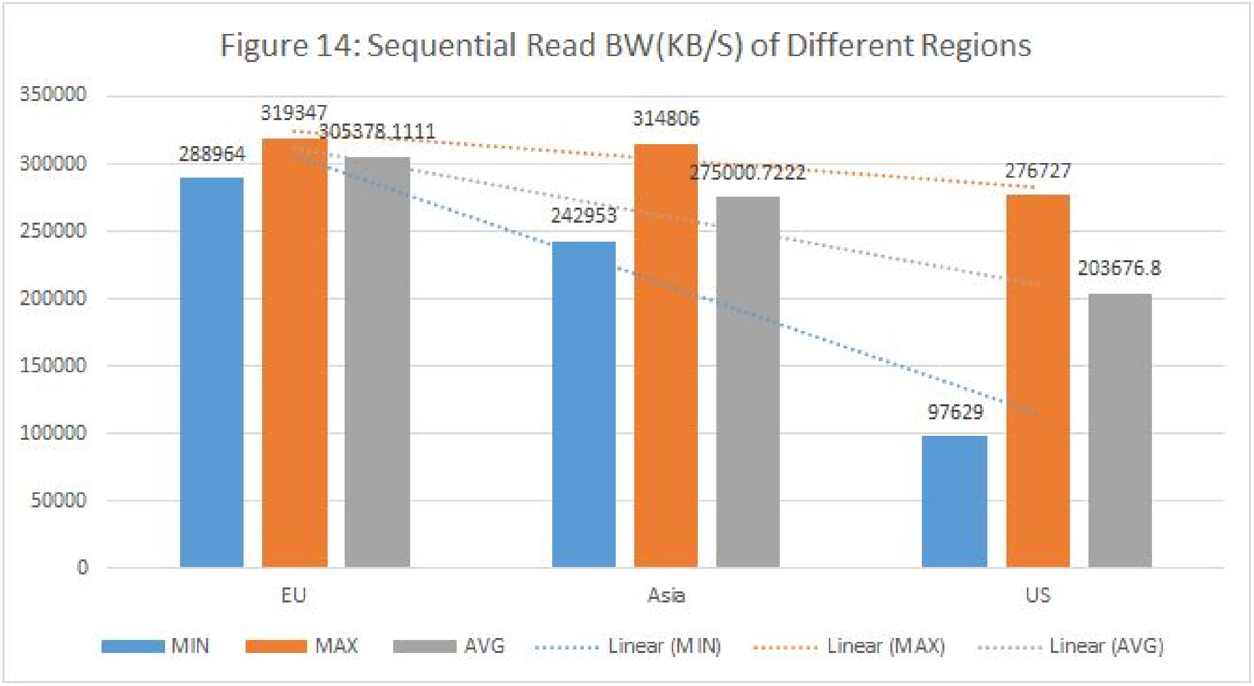
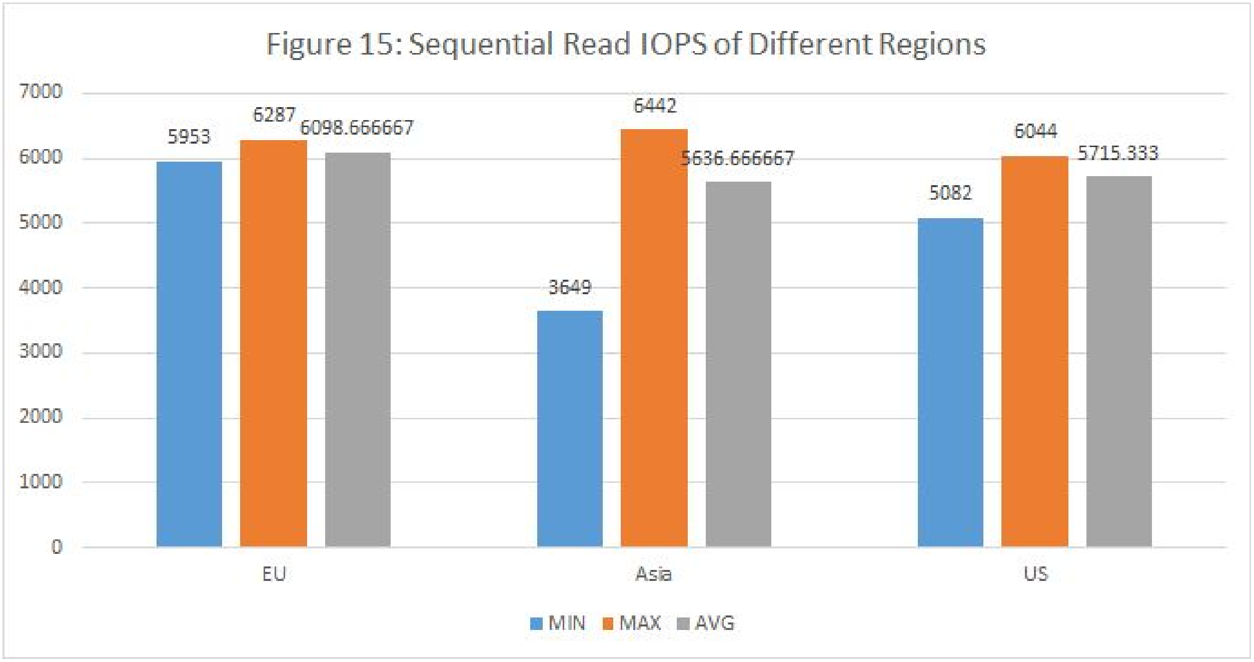


Figure 13: Sequential Write IOPS of Different Regions
4600
4400
4200
4000
3800
3600
3400
3200
3000
3995
4360
4153.962963
3448
4360
4090.166667
3567
4299
4123.704
EU
Asia
US
MIN
MAX
AVG
Linear (MAX)





Local SSD

Following is our results from local SSD test.

	500G(16 vCPUs, 60 GB memory)	834G(32 vCPUs, 120 GB memory)	2048G(64 vCPUs, 240 GB memory)	2048G(96 vCPUs, 360 GB memory)
Random Write IOPS	15000	25000	30000	30000
Random Read IOPS	15000	11000	13000	19000
Seq Write IOPS	15000	25000	30000	30000
Seq Read IOPS	15000	25000	40000	40000
Random Write BW (MB/S)	245	410	410	410
Random Read BW (MB/S)	245	410	820	820
Seq Write BW (MB/S)	245	410	410	410
Seq Read BW (MB/S)	245	410	820	820
Expected Random Write IOPS	15000	25000	30000	30000
Expected Random Read IOPS	15000	25000	40000	40000
Expected Random Write BW (MB/S)	240	400	400	400
Expected Random Read BW (MB/S)	240	400	800	800

Table 5: Local SSD Benchmark

Google provided SLA on random IOPS and bandwidth for SSD in following page.
https://cloud.google.com/compute/docs/disks/performance

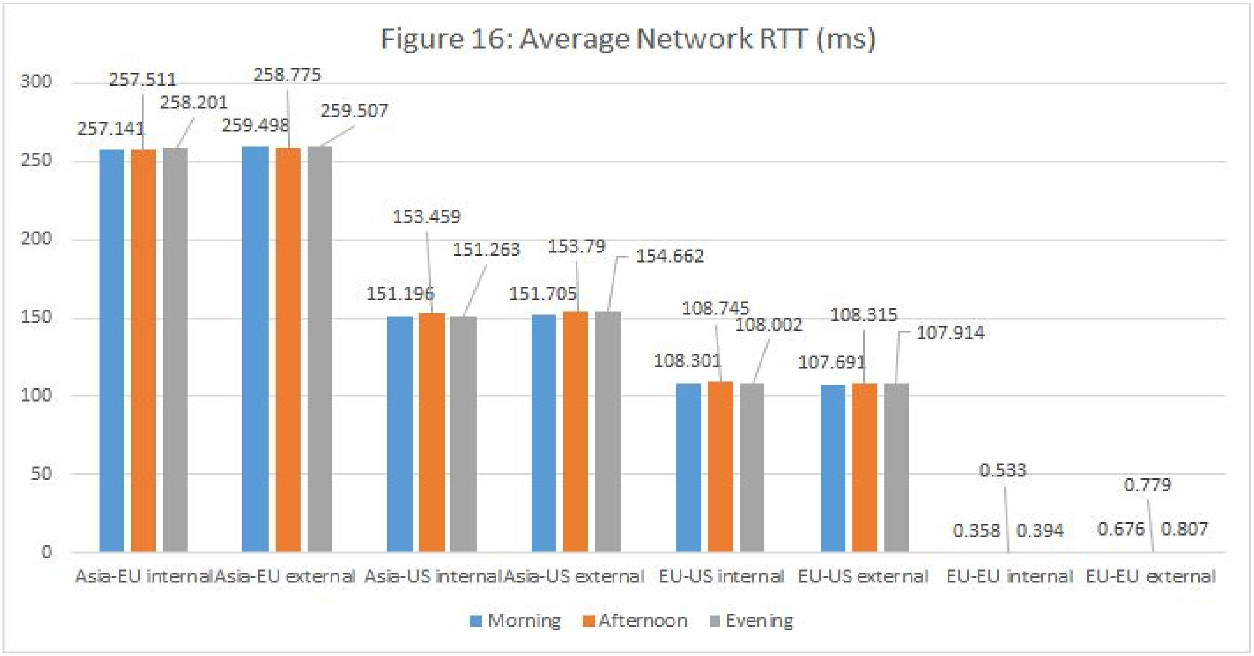
Our result shows that sequential read/write performance is similar to random read/write performance, and matches with Google SLA, except the random read iops of 834G and 2048G size SSD, which is always below the expected IOPS. From the result, it looks like the higher number of vCPU and more memory, the better the performance, so we are not sure if the unexpected low performance was caused by insufficient resources as we can max assign 96 vCPU to a VM.

As mentioned earlier, we also ran tests on latency for both external storage and local SSD, however, due to time limitation and latency is less interesting for this task, we do not provide analysis on the latency test result. But if you are interested, you can find the result in our original result files that we can provide on request.

Task 4 - Simple Network

In this section we are going to explain the results of the benchmark that we did across regions and in the same zones using iperf tool. The VM machines we used in this task are Ubuntu 16.04 LTS with 4 vCPU, 15 GB memory, until it is specified with 96vCPU.

First, we did the network latency test using ping command across regions and across zones in the same European region with their internal and external IP addresses at different times of a day. The result can be viewed as follows.



As we can see in the above figure, the RTT of region to region is higher than the RTT of within the same region. The possible assumptions for this variation are, in the same availability region, VMs are supposed to be in the same data center or in relatively closer geographical locations. Internal IP shows advantage in same region test.

The other part of task 4 is the measurement of the tcp/udp communications. In this section we tested 3 instances across regions with their internal and external IPs and 3 instances in the same availability region. In addition, we also examined using different cores.

For UDP benchmark, following table shows the packet loss percentage for different bandwidth we chose.

	10M	20M	30M	50M	100M	150M
Asia-US External	0	0.0059	0.0078	0.02	0.073	0.051
Asia-EU external	0	0.0059	0.0065	0.013	0.01	0.02
EU-US external	0	0.002	0.0052	0.012	0.0066	0.016

Asia-US Internal	0	0	0	0.00078	0.00078	0.0094
Asia-EU Internal	0	0	0	0	0.00078	0.001
EU-US Internal	0	0	0	0	0	0.00026
EUA-EUB external	0	0	0	0.012	0.0082	0.088
EUA-EUC external	0	0	0.0039	0.0071	0.0094	0.014
EUB-EUC external	0	0	0	0.0071	0.025	0.013
EUA-EUB internal	0	0	0	0	0.0012	0.0055
EUA-EUC internal	0	0	0	0	0.007	0.0039
EUB-EUC internal	0	0	0	0	0.015	0.0086
EUA-EUB External 96vCPU	0	0	0	0	0.0035	0.0062
EUA-EUB Internal 96vCPU	0	0	0	0	0.0035	0.0023

Table 6: UDP Packet loss percentage (%)

As we can see in above table, with external IP, when bandwidth was 20M, it started packet loss. Internal IP shows advantages. And below is the related jitter for your reference.

	10M	20M	30M	50M	100M	150M
Asia-US External	0.053	0.065	0.058	0.043	0.039	0.036

Asia-EU external	0.059	0.046	0.054	0.128	0.06	0.049
EU-US external	0.074	0.038	0.042	0.065	0.041	0.03
Asia-US Internal	0.041	0.071	0.039	0.023	0.019	0.018
Asia-EU Internal	0.048	0.059	0.041	0.034	0.066	0.032
EU-US Internal	0.034	0.027	0.032	0.027	0.022	0.023
EUA-EUB external	0.056	0.060	0.058	0.031	0.031	0.017
EUA-EUC external	0.028	0.039	0.027	0.024	0.035	0.044
EUB-EUC external	0.053	0.062	0.040	0.069	0.032	0.044
EUA-EUB internal	0.022	0.025	0.019	0.021	0.022	0.026
EUA-EUC internal	0.023	0.025	0.028	0.024	0.022	0.031
EUB-EUC internal	0.096	0.036	0.029	0.024	0.027	0.013
EUA-EUB External 96vCPU	0.032	0.042	0.039	0.016	0.013	0.014
EUA-EUB Internal 96vCPU	0.009	0.022	0.008	0.008	0.002	0.001

Table 7: UDP Jitter (ms)

For TCP benchmark, below table shows the bandwidth we got with iperf default window size.

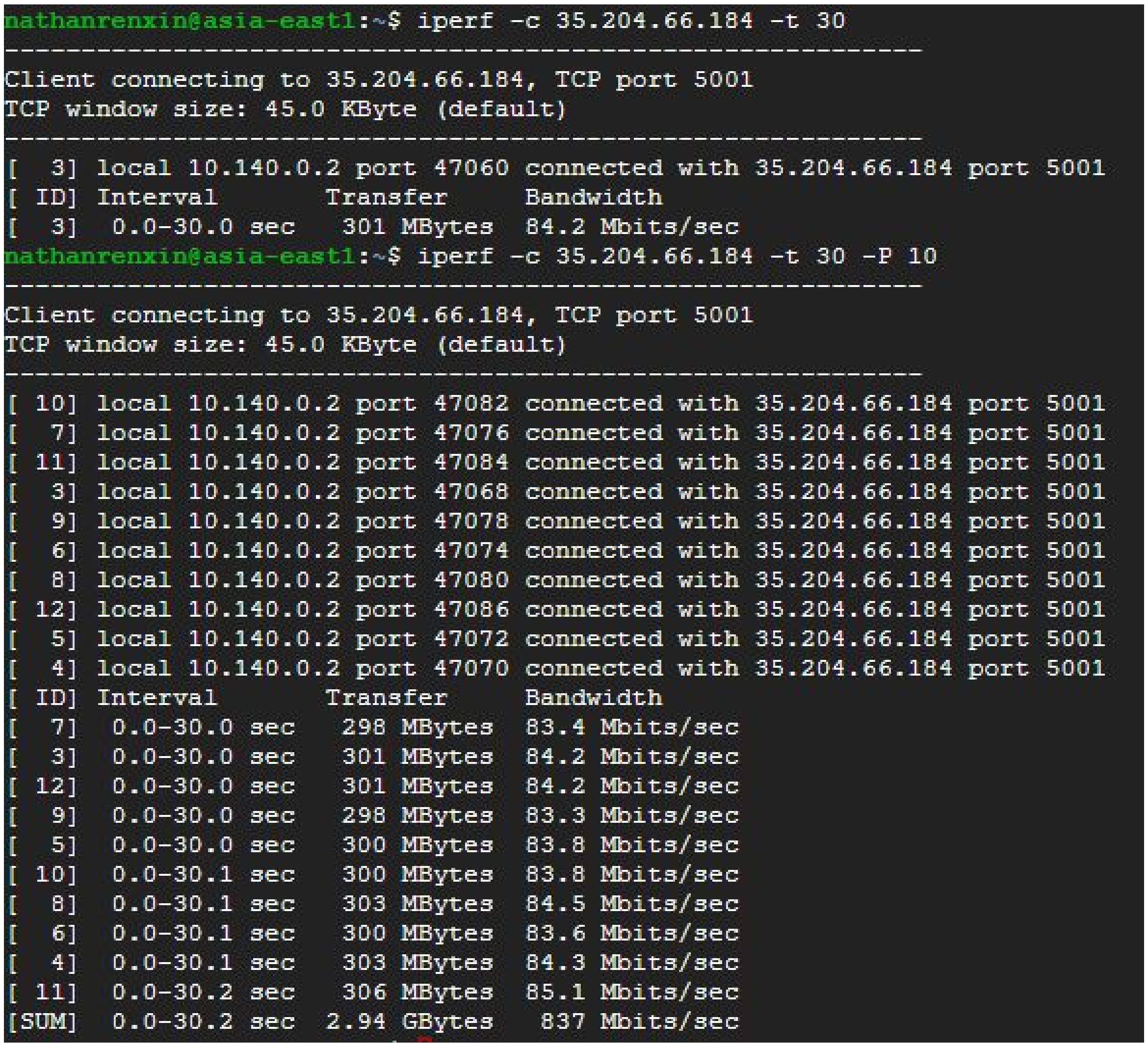
Asia-US External	144 Mb/s
Asia-US internal	157 Mb/s
Asia-EU external	83.5 Mb/s
Asia-EU internal	88.9 Mb/s
EU-US external	208 Mb/s
EU-US internal	225 Mb/s
EUA-EUB external	1.51 Gb/s
EUA-EUB internal	7.86 Gb/s
EUB-EUC external	1.88 Gb/s
EUB-EUC internal	7.85 Gb/s
EUA-EUC external	1.66 Gb/s
EUA-EUC internal	7.87 Gb/s
EUA-EUB External 96vCPU	1.86 Gb/s
EUA-EUB Internal 96vCPU	14.2 Gb/s

Table 8: TCP bandwidth with default WS

It shows that Europe and US have the best connection, while Asia and EU have the worst. Using internal IP or increase VM resources can help improve the throughput.

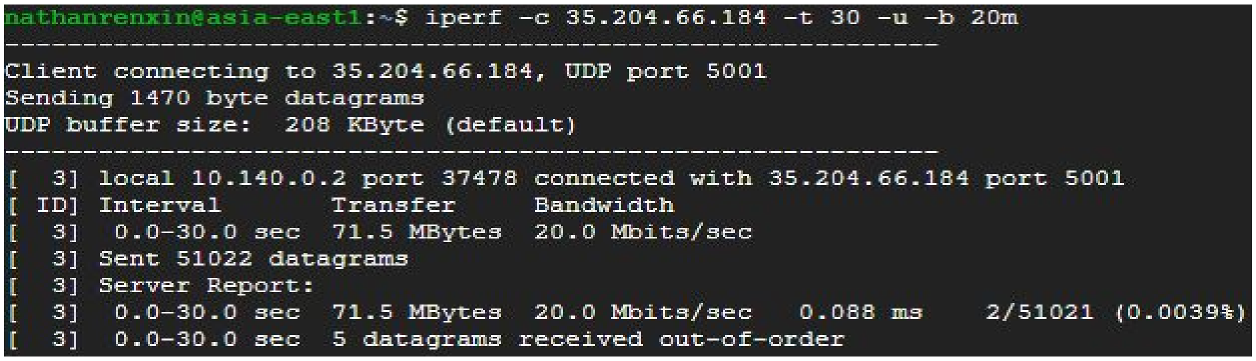
We examined parallel transfer for both TCP and UDP, which showed that parallel transfer can help improve throughput and all the processes shares similar performance. As you can see from our example data from Asia and Europe connection with external

IP addresses.



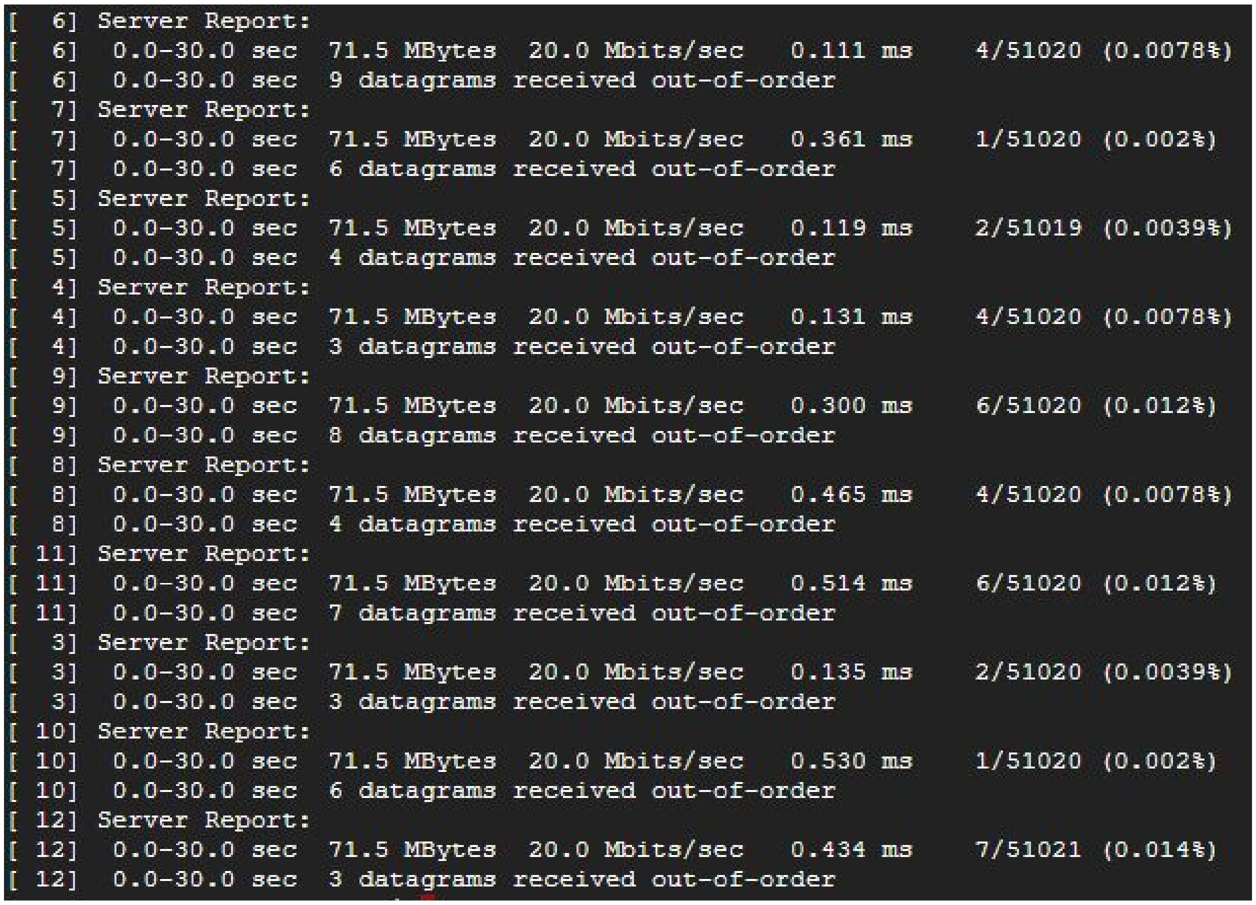
```
nathanrenxin@asia-east1:~$ iperf -c 35.204.66.184 -t 30
------------------------------------------------------------
Client connecting to 35.204.66.184, TCP port 5001
TCP window size: 45.0 KByte (default)
------------------------------------------------------------
[  3] local 10.140.0.2 port 47060 connected with 35.204.66.184 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-30.0 sec   301 MBytes  84.2 Mbits/sec
nathanrenxin@asia-east1:~$ iperf -c 35.204.66.184 -t 30 -P 10
------------------------------------------------------------
Client connecting to 35.204.66.184, TCP port 5001
TCP window size: 45.0 KByte (default)
------------------------------------------------------------
[ 10] local 10.140.0.2 port 47082 connected with 35.204.66.184 port 5001
[  7] local 10.140.0.2 port 47076 connected with 35.204.66.184 port 5001
[ 11] local 10.140.0.2 port 47084 connected with 35.204.66.184 port 5001
[  3] local 10.140.0.2 port 47068 connected with 35.204.66.184 port 5001
[  9] local 10.140.0.2 port 47078 connected with 35.204.66.184 port 5001
[  6] local 10.140.0.2 port 47074 connected with 35.204.66.184 port 5001
[  8] local 10.140.0.2 port 47080 connected with 35.204.66.184 port 5001
[ 12] local 10.140.0.2 port 47086 connected with 35.204.66.184 port 5001
[  5] local 10.140.0.2 port 47072 connected with 35.204.66.184 port 5001
[  4] local 10.140.0.2 port 47070 connected with 35.204.66.184 port 5001
[ ID] Interval       Transfer     Bandwidth
[  7]  0.0-30.0 sec   298 MBytes  83.4 Mbits/sec
[  3]  0.0-30.0 sec   301 MBytes  84.2 Mbits/sec
[ 12]  0.0-30.0 sec   301 MBytes  84.2 Mbits/sec
[  9]  0.0-30.0 sec   298 MBytes  83.3 Mbits/sec
[  5]  0.0-30.0 sec   300 MBytes  83.8 Mbits/sec
[ 10]  0.0-30.1 sec   300 MBytes  83.8 Mbits/sec
[  8]  0.0-30.1 sec   303 MBytes  84.5 Mbits/sec
[  6]  0.0-30.1 sec   300 MBytes  83.6 Mbits/sec
[  4]  0.0-30.1 sec   303 MBytes  84.3 Mbits/sec
[ 11]  0.0-30.2 sec   306 MBytes  85.1 Mbits/sec
[SUM]  0.0-30.2 sec  2.94 GBytes   837 Mbits/sec
```

Figure 17: TCP single transfer V.S. Parallel transfer



```
nathanrenxin@asia-east1:~$ iperf -c 35.204.66.184 -t 30 -u -b 20m
------------------------------------------------------------
Client connecting to 35.204.66.184, UDP port 5001
Sending 1470 byte datagrams
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
[  3] local 10.140.0.2 port 37478 connected with 35.204.66.184 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-30.0 sec  71.5 MBytes  20.0 Mbits/sec
[  3] Sent 51022 datagrams
[  3] Server Report:
[  3]  0.0-30.0 sec  71.5 MBytes  20.0 Mbits/sec   0.088 ms    2/51021 (0.0039%)
[  3]  0.0-30.0 sec  5 datagrams received out-of-order
```

Figure 18: UDP single transfer



```
[  6] Server Report:
[  6]  0.0-30.0 sec  71.5 MBytes  20.0 Mbits/sec   0.111 ms    4/51020 (0.0078%)
[  6]  0.0-30.0 sec  9 datagrams received out-of-order
[  7] Server Report:
[  7]  0.0-30.0 sec  71.5 MBytes  20.0 Mbits/sec   0.361 ms    1/51020 (0.002%)
[  7]  0.0-30.0 sec  6 datagrams received out-of-order
[  5] Server Report:
[  5]  0.0-30.0 sec  71.5 MBytes  20.0 Mbits/sec   0.119 ms    2/51019 (0.0039%)
[  5]  0.0-30.0 sec  4 datagrams received out-of-order
[  4] Server Report:
[  4]  0.0-30.0 sec  71.5 MBytes  20.0 Mbits/sec   0.131 ms    4/51020 (0.0078%)
[  4]  0.0-30.0 sec  3 datagrams received out-of-order
[  9] Server Report:
[  9]  0.0-30.0 sec  71.5 MBytes  20.0 Mbits/sec   0.300 ms    6/51020 (0.012%)
[  9]  0.0-30.0 sec  8 datagrams received out-of-order
[  8] Server Report:
[  8]  0.0-30.0 sec  71.5 MBytes  20.0 Mbits/sec   0.465 ms    4/51020 (0.0078%)
[  8]  0.0-30.0 sec  4 datagrams received out-of-order
[ 11] Server Report:
[ 11]  0.0-30.0 sec  71.5 MBytes  20.0 Mbits/sec   0.514 ms    6/51020 (0.012%)
[ 11]  0.0-30.0 sec  7 datagrams received out-of-order
[  3] Server Report:
[  3]  0.0-30.0 sec  71.5 MBytes  20.0 Mbits/sec   0.135 ms    2/51020 (0.0039%)
[  3]  0.0-30.0 sec  3 datagrams received out-of-order
[ 10] Server Report:
[ 10]  0.0-30.0 sec  71.5 MBytes  20.0 Mbits/sec   0.530 ms    1/51020 (0.002%)
[ 10]  0.0-30.0 sec  6 datagrams received out-of-order
[ 12] Server Report:
[ 12]  0.0-30.0 sec  71.5 MBytes  20.0 Mbits/sec   0.434 ms    7/51021 (0.014%)
[ 12]  0.0-30.0 sec  3 datagrams received out-of-order
```

Figure 19: UDP parallel transfer

2 Bonus Tasks

Task 5 - Custom Network

The custom network task concerns the testing and implementation of the LEDBAT protocol.

Then benchmarks were executed in Google’s data center in Frankfurkt, both machines belonging to the same region. Both, the server and the client machine were the same as in previous tasks, i.e. 4-core 15 GB Ubuntu 16.04 vms.

The main question to answer was if the used LEDBAT implementation correctly backs of when a TCP stream is encountered. For further details of LEDBAT the user is refered to [1]. The testing was done with an open source implementation [3] of LEDBAT in python 3 which doesn’t incorporate the DTL improvement from [2].

The server is started with:

```
python3 testapp.py
```

While the client runs with:

```
python3 testapp.py --role=client --remote=<IP of the first host>
```

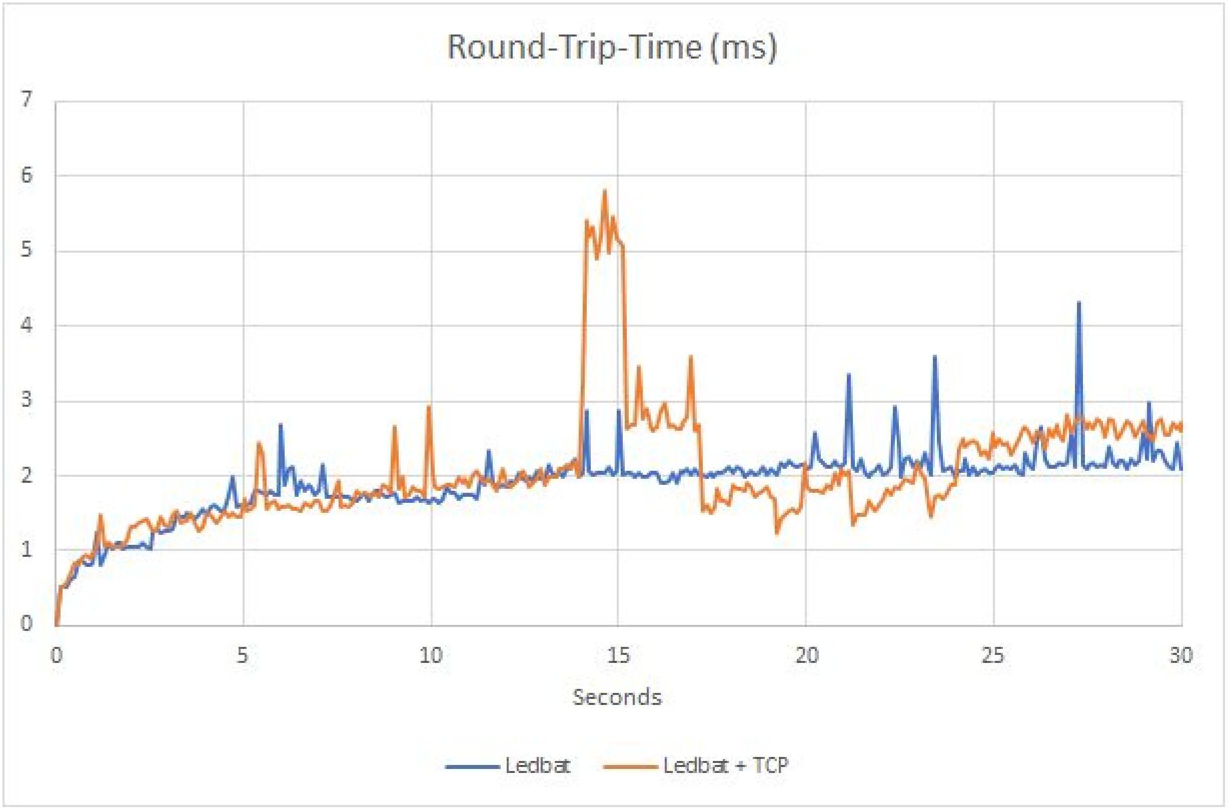
Using *-h* opens the help to see the parameters available, more can be found in [3]. The ones used were:

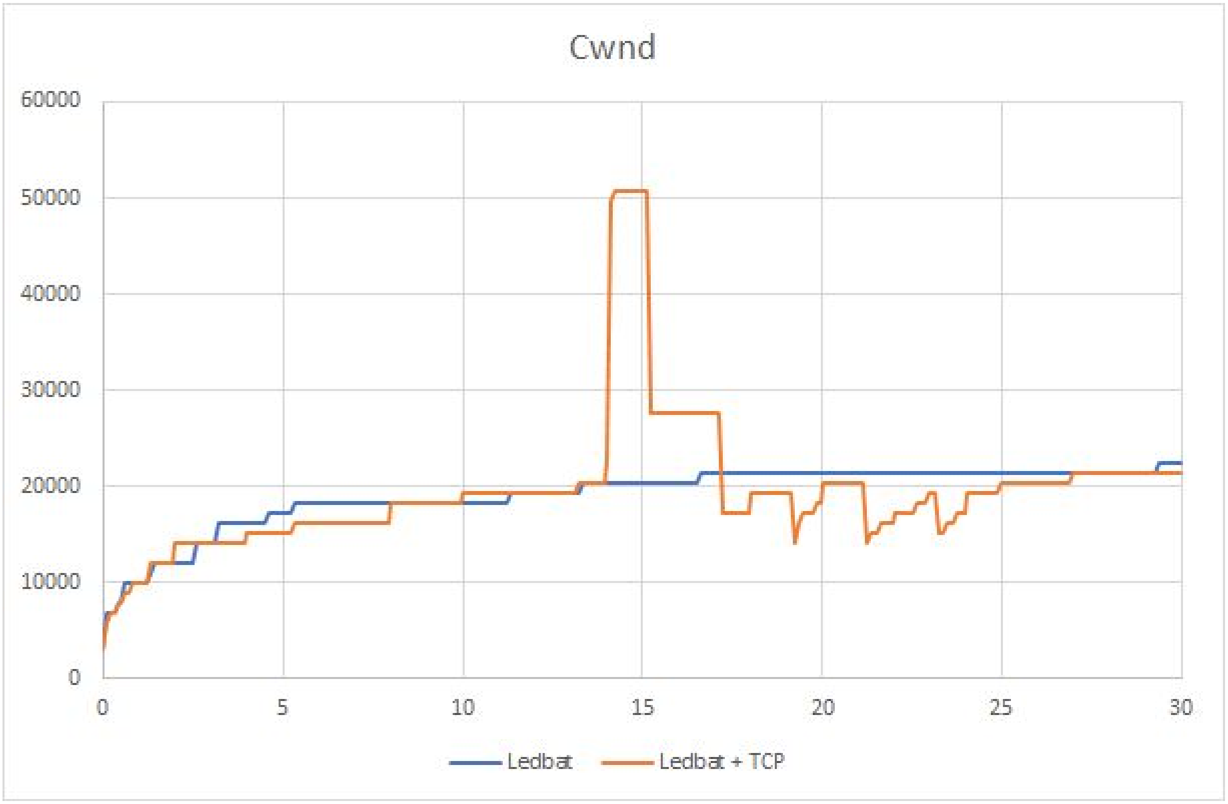
- `--role {client|server}` Run client in the given mode. Server ignores all other options
- `--remote <IP Address>` IP Address of the LEDBAT test application running in the server mode
- `--makelog` Save various application runtime values into CSV file
- `--log-name <Name>` Name of the log file. By default it is UnixTime-RemoteIP-RemotePort.csv
- `--log-dir <Name>` Path to the directory where the log file should be saved
- `--time <NSec>` Run client for indicated number of seconds before exiting
- `--parallel <N>` Run indicated number of parallel data transfers
- `--ledbat-set-target <ms>` Set the LEDBAT target delay to the indicated value (ms)

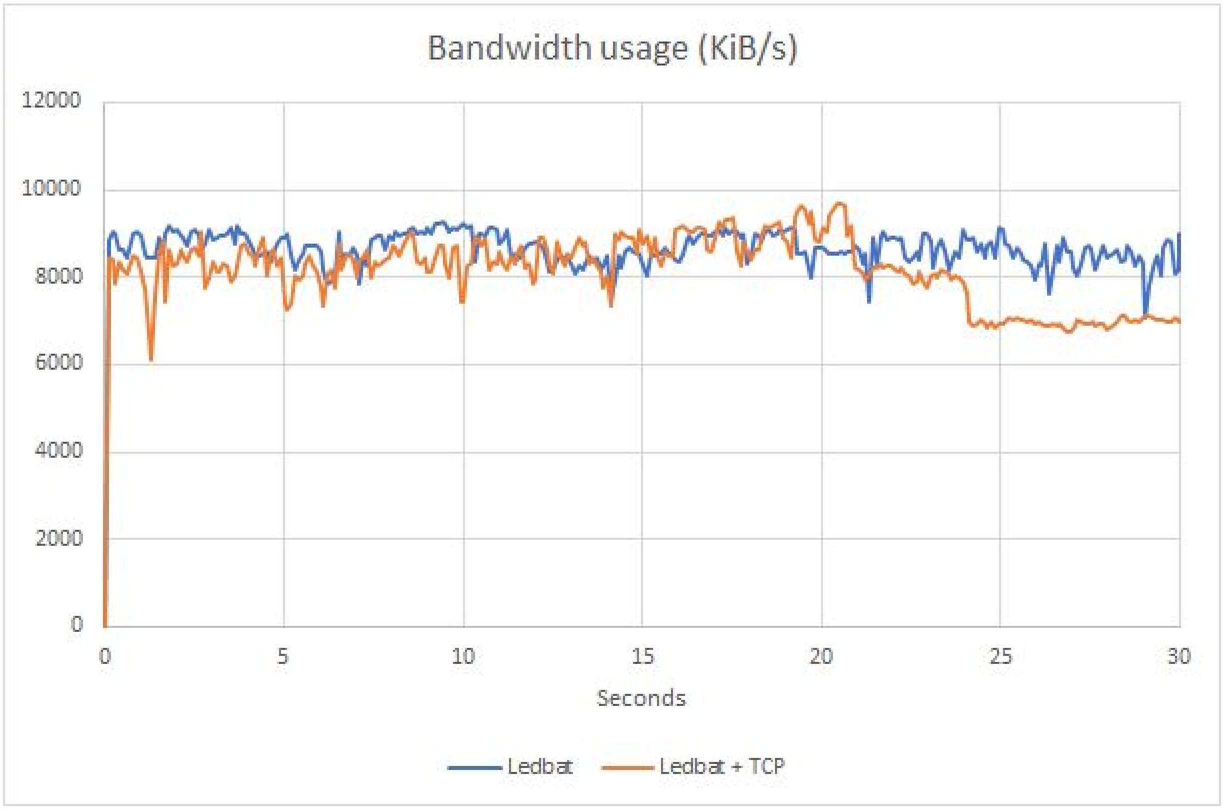
Standard Parameter benchmarks

The cloned implementation has a target delay of 100ms and sends a number of 1KB messages to a given server. We will first present 3 graphs comparing the throughput, Round-Trip-Time and CWND of a typical single Ledbat run to a run which had a TCP connection running from second 14 to 24 using the IPERF tool as in Task 4.

```
python3 testapp.py --role=client --remote<...> --time 30
--makelog
```


Round-Trip-Time (ms)
7
6
5
4
3
2
1
0
0
5
10
15
20
25
30
Seconds
Ledbat
Ledbat + TCP


Cwnd
60000
50000
40000
30000
20000
10000
0
0
5
10
15
20
25
30
Ledbat
Ledbat + TCP

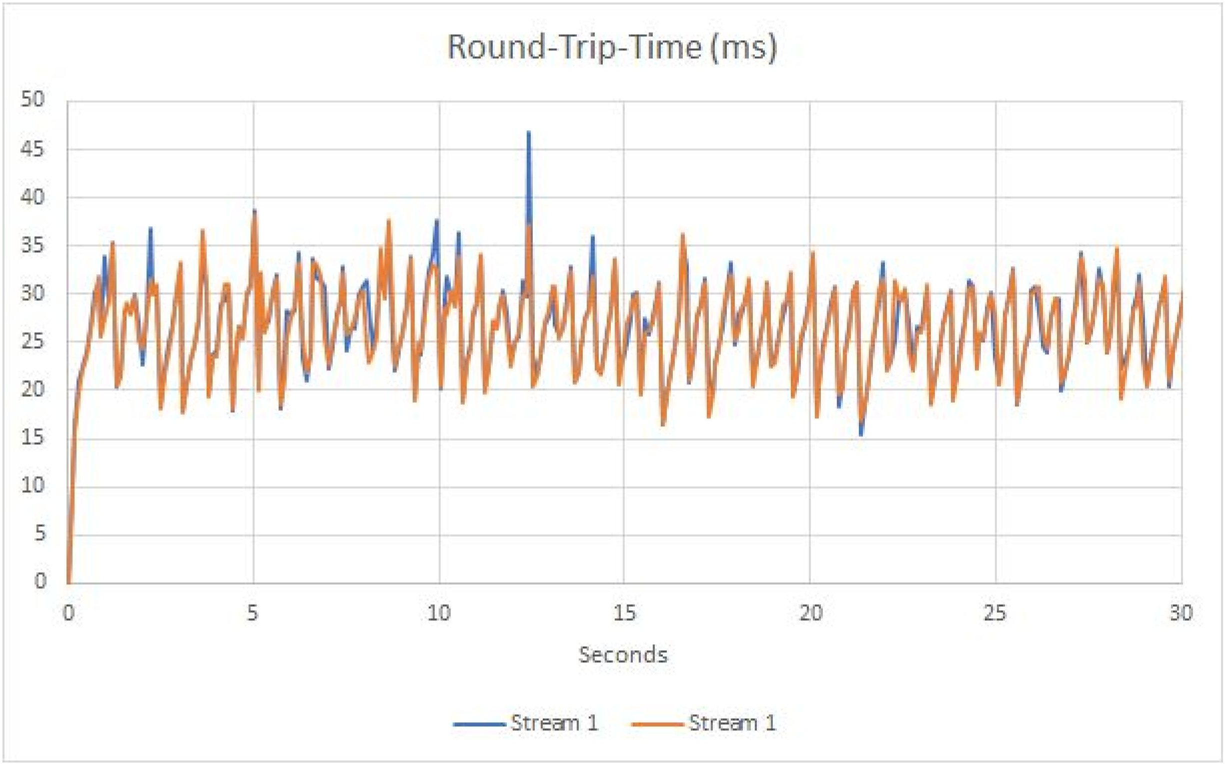


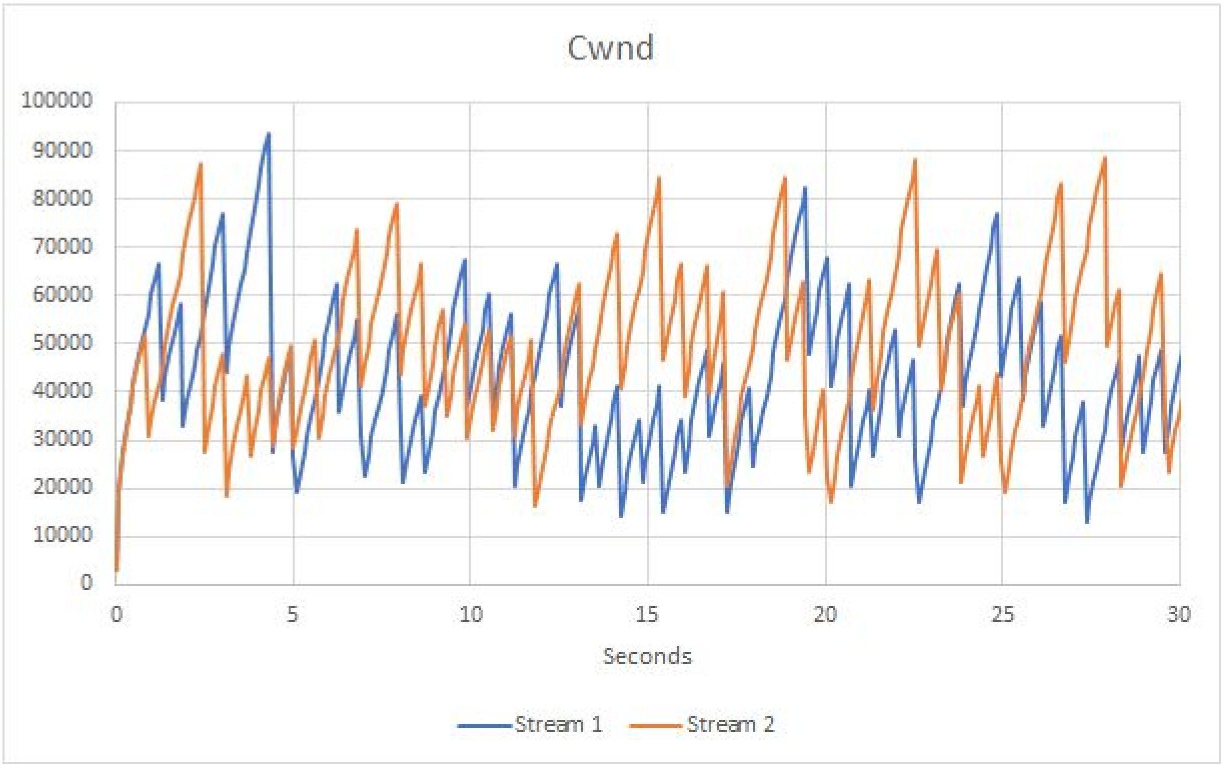
In task 4, high bandwidth was observed, making it most likely the culprit behind the unaffected bandwidth usage of the Ledbat stream while iperf was running. Nonetheless, the CWND and RTT graphs give a different impression. Both indicate an event happening around second 14, the same moment IPERF started transmitting. Moreover, both remain unstable until IPERF finishes at second 25, before becoming stable again.

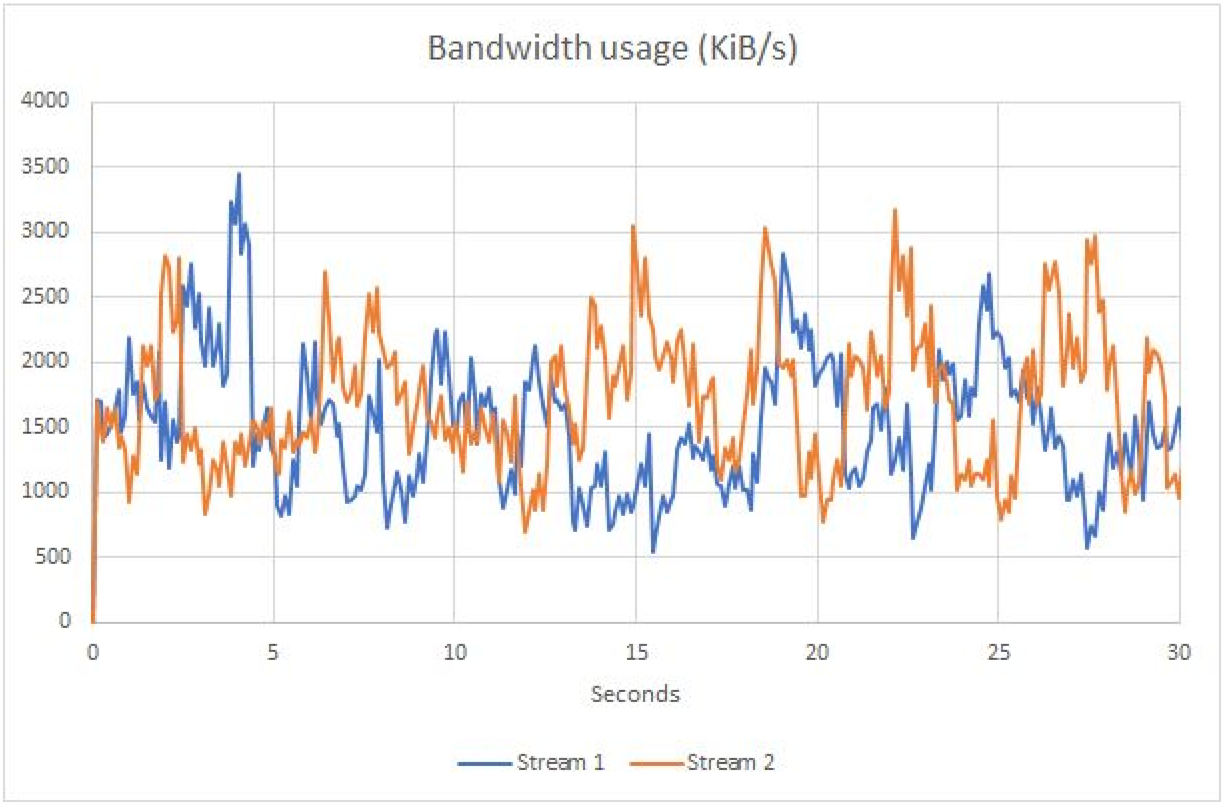
Multi stream LEDBAT

The multi stream test were executed with the following parameters:

```
python3 testapp.py --role=client --remote=<...> --time 30
--makelog --parallel 2
```


Round-Trip-Time (ms)
50
45
40
35
30
25
20
15
10
5
0
0
5
10
15
20
25
30
Seconds
Stream 1
Stream 1


Cwnd
100000
90000
80000
70000
60000
50000
40000
30000
20000
10000
0
0
5
10
15
20
25
30
Seconds
Stream 1
Stream 2



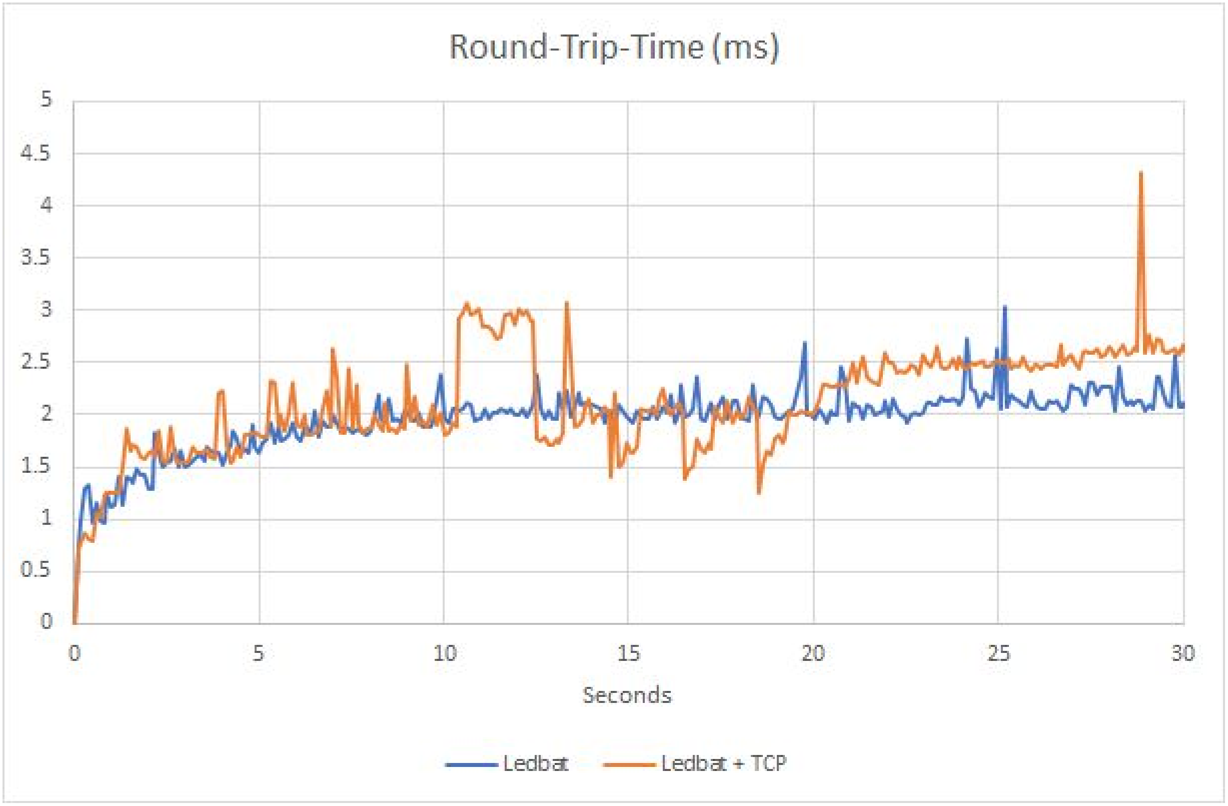
The python implementation provides functionality for having multiple streams open at once. Apparently the streams compete for local resources as can bee seen in the graphs. This leads to the conclusion that they run in the same thread.

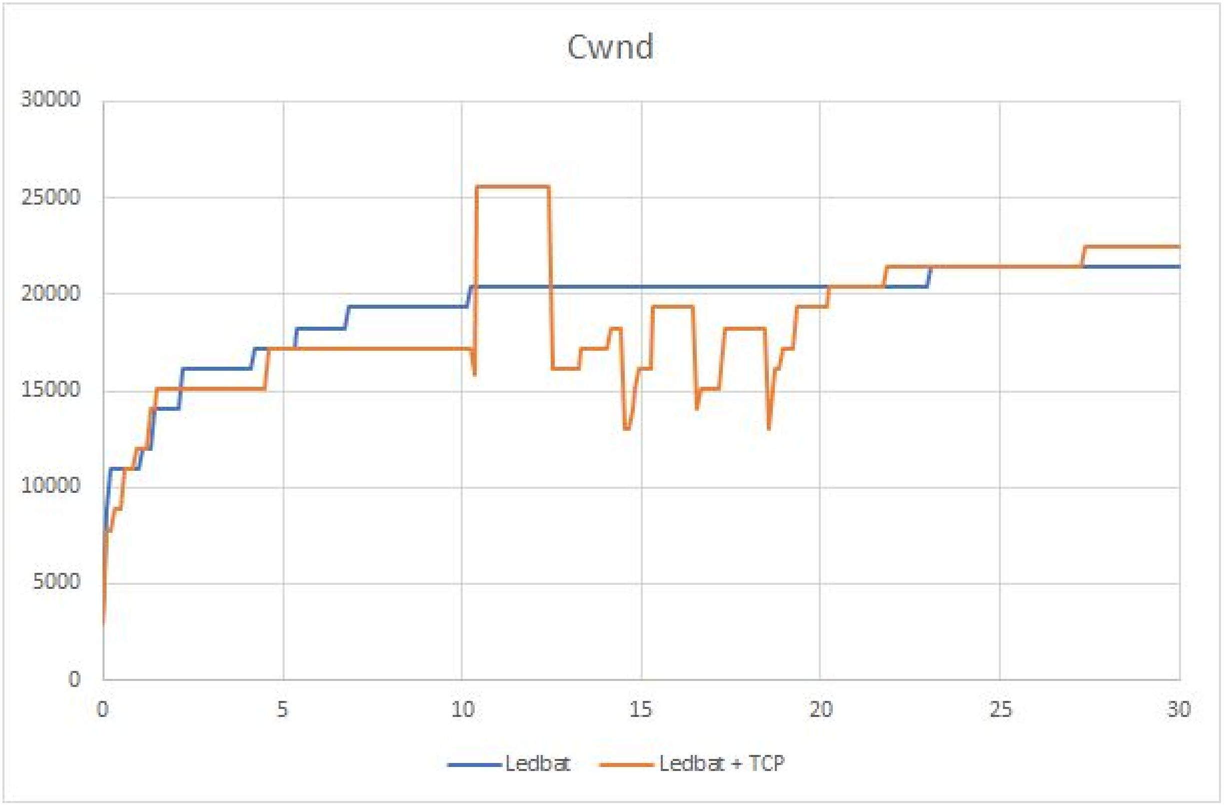
Modifying parameters

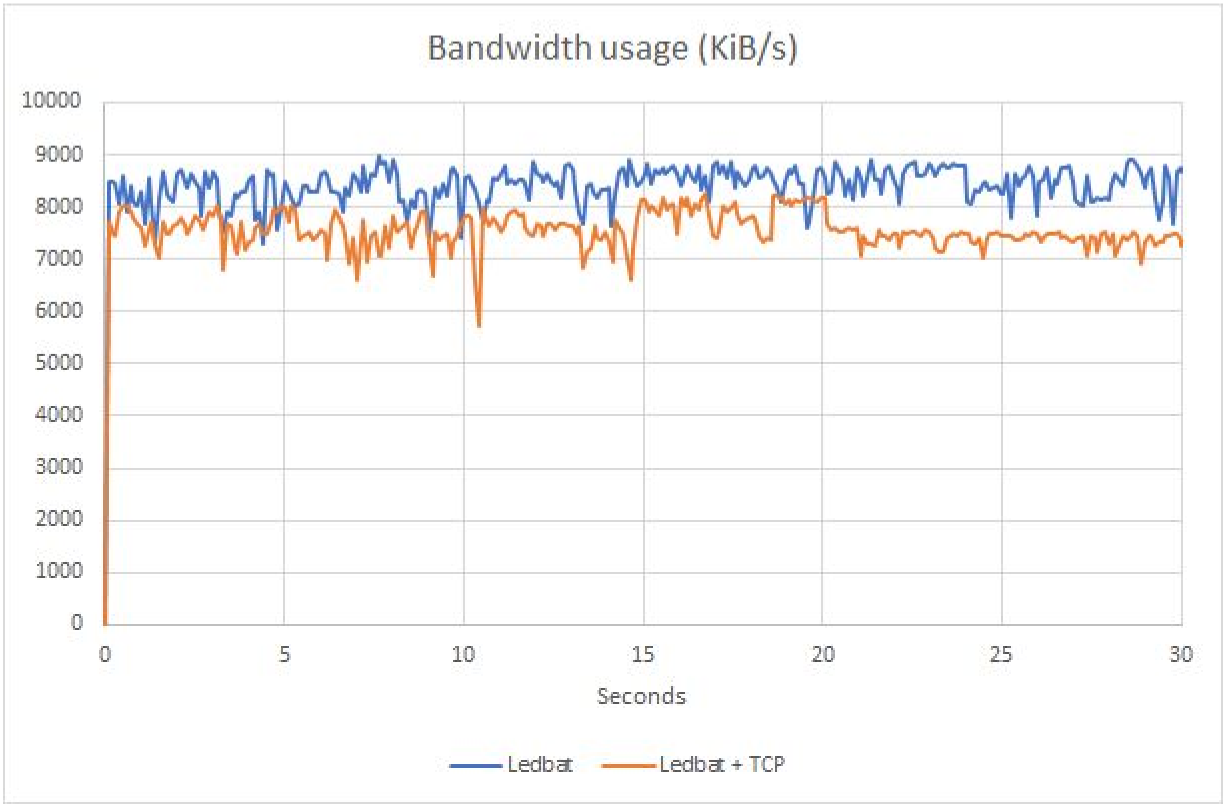
Repeated runs with the same configuration as in the standard benchmark, except using a target delay of 10ms shows a very similar picture. In the following depictions IPERF was started at second 10.

Executed with:

```
python3 testapp.py --role=client --remote<...> --time 30 --makelog --ledbat-set-target 10
```


Round-Trip-Time (ms)
5
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
0
5
10
15
20
25
30
Seconds
Ledbat
Ledbat + TCP


Cwnd
30000
25000
20000
15000
10000
5000
0
0
5
10
15
20
25
30
Ledbat
Ledbat + TCP



As expected, there is no noticeable difference to the 100 ms target. This is due to 2 facts:

1. The very high bandwidth inside a region.
2. The very low and steady RTT, even in presence of a TCP stream (except the TCP start).

Ledbat Kompics implementation

The Ledbat code was finished with one exception; only the first ACK is delivered to the higher layer while the rest of the messages time out and get resend although it seems that the client receives the messages. This can be seen in the deserialization logging. In its current form the logging is reduced to INFO through the use of a *logback.xml* file under *ledbat/src/main/resources*.

While not being able to find the error, we are confident that it is the only piece missing for a complete implementation. The serializer is a custom implementation heavily based on [4].

The server is started with:

```
se.application.Main <its public IP> <port to listen>
Example-> se.application.Main localhost 55555
```

```
The client is started with:
```

```
se.application.Main <its public IP> <port to listen> <server’s
IP> <server’s port> <seconds to run>
Example-> se.application.Main localhost 55555 localhost 54321 10
```

Inside *SendingApplication.java* a hardcoded variable with the message buffer size can be found. The client will fill the pending list of messages to be send with said amount and fill it up every second with the amount of Acks received as to keep the buffer full. The messages transfered are 10 KB big.

How to Run Our Code

For task 1 and 2, we have a Java Maven project named project_id2210 as we use Google Cloud Client Java Libraries. We recommend to use Eclipse Java EE IDE for Web Developers to run the code.

Before running the code, you need to do authentication so that you have the right access to Google Cloud. For Cloud Storage, please follow “Setting up authentication” session in https://cloud.google.com/storage/docs/reference/libraries. To access VMs, please follow following steps to authenticate.

1. Generate the client secret json file
   a. Go to https://console.cloud.google.com/apis/credentials
   b. Then click on create credentials
   c. Select Create OAuth client ID
   d. Select Web application
   e. Provide a valid name
   f. Click create.
   g. Copy the client_id and client_secret and modify the client_secretes.json file under resources folder in the project.
2. Generate service account json file
   a. Go to https://console.cloud.google.com/apis/credentials
   b. Then click on create credentials

c. Select Service account key
d. Provide required information
e. Download the file and modify the serviceAccount.json file under resources folder in the project.

To access Cloud Storage, you first need to prepare files named with numbers starting from 1 in folder "upload" in the project. The number of files needed depends on later on the number of parallel transfer processes. Then please run "Runner.java". It creates a bucket, uploads files, downloads files, removes files, and removes the bucket. For single transfer, please run it without any arguments. For simultaneous parallel transfer, please run it with "<number of processes> 0" and for parallel transfer with 5s interval, please run it with "<number of processes> 1". You may need to modify the sleep time in the program base on your network situation. (In our benchmark tests, the files we downloaded are different from the files we uploaded so that we could have bigger size download files as download is quicker than upload.)

To create VM remotely, you can run either "CreateInstance.java" or "ListInstances.java" after changing the project id field in the files to your own.

For TCP/UDP access to VM, you need to upload the server java files to your VM and replace the server IP in the client java files to your own VM external IP and run the client java files on your PC.

References

1. RFC 6817 - Low Extra Delay Background Transport. https://tools.ietf.org/html/rfc6817
2. Reale R., Roverso R., El-Ansary S., Haridi S. (2012) DTL: Dynamic Transport Library for Peer-to-Peer Applications. In: Bononi L., Datta A.K., Devismes S., Misra A. (eds) Distributed Computing and Networking. ICDCN 2012. Lecture Notes in Computer Science, vol 7129. Springer, Berlin, Heidelberg
3. Github repository of the open source Ledbat implementation: https://github.com/justas-/pyledbat
4. Kompics Networking Tutorial, § Serialization: http://kompics.sics.se/current/tutorial/networking/basic/basic.html#serialisation