Fall 2022

Performance Analytics of Cloud Networks

Derek Phanekham
Southern Methodist University, dphanekham@gmail.com

PERFORMANCE ANALYTICS OF CLOUD NETWORKS

Approved by:

Dr. Suku Nair
Professor

Dr. Jennifer Dworak
Professor

Dr. Nageswara Rao
Corporate Fellow

Dr. Michael Hahsler
Clinical Professor

Dr. Jeff Tian
Professor

PERFORMANCE ANALYTICS OF CLOUD NETWORKS

A Dissertation Presented to the Graduate Faculty of the

Lyle School of Engineering

Southern Methodist University

in

Partial Fulfillment of the Requirements

for the degree of

Doctor of Philosophy

with a

Major in Computer Science

by

Derek Phanekham

B.S., Computer Science, Southern Methodist University
M.S., Computer Science, Southern Methodist University

December 17, 2022

Acknowledgments

I would like to thank and acknowledge all of those who have supported me during this journey. So thank you to my mother and father, Kelly and Samuel Phanekham, my partner, Hayley Moore, and my sister, Jamie Phanekham.

Phanekham, Derek B.S., Computer Science, Southern Methodist University
M.S., Computer Science, Southern Methodist University

Performance Analytics of Cloud Networks

–

Advisor: Dr. Suku Nair
Doctor of Philosophy degree conferred December 17, 2022
Dissertation completed November 10, 2022

As the world becomes more inter-connected and dependent on the Internet, networks become ever more pervasive, and the stresses placed upon them more demanding. Similarly, the expectations of networks to maintain a high level of performance have also increased. Network performance is highly important to any business that operates online, depends on web traffic, runs any part of their infrastructure in a cloud environment, or even hosts their own network infrastructure. Depending upon the exact nature of a network, whether it be local or wide-area, 10 or 100 Gigabit, it will have distinct performance characteristics and it is important for a business or individual operating on the network to understand those performance characteristics and how they affect operations.

To better understand our networks, it is necessary that we test them to measure their performance capabilities and track these metrics over time. In our work, we provide an in-depth analysis of how best to run cloud benchmarks to increase our network intelligence and how we can use the results of those benchmarks to predict future performance and identify performance anomalies. To achieve this, we explain how to effectively run cloud benchmarks and propose a scheduling algorithm for running large numbers of cloud benchmarks daily. We then use the performance data gathered from this method to conduct a thorough analysis of

the performance characteristics of a cloud network, train neural networks to forecast future throughput based on historical results and detect performance anomalies as they occur.

TABLE OF CONTENTS

ACKNOWLEDGMENTS iii

LIST OF FIGURES xii

LIST OF TABLES xv

CHAPTER

1 Introduction 1

1.1. Motivation 2

1.2. Objectives and Contributions 5

1.3. Organization 6

2 Related Work 8

2.1. Network Performance Benchmarking 8

2.1.1. Performance Measurements and Benchmarking Tools 10

2.1.1.1. Network Throughput Measurements 10

2.1.1.2. Network Latency Measurements 12

2.1.1.3. Benchmark Suites 13

2.1.2. Cloud Benchmarking 13

2.2. Network Performance Analytics 15

2.3. Network Performance Forecasting 18

2.4. Time Series Analysis 18

2.4.1. Time series statistics 18

2.4.1.1. Stationarity, Seasonality, and Trends 19

2.4.2. Moving Average 21

2.4.3. ARIMA 21

2.4.4. Support Vector Regression 22
2.5. Throughput Prediction 22
2.5.1. Formula-based TCP Throughput Prediction Methods 22
2.5.2. History-based TCP Throughput Prediction Methods 23
2.5.3. Neural Networks Applied to Time Series Prediction 24
2.6. Anomaly Detection Techniques 26
3 Network Measurements and Performance Benchmarking 29
3.1. Accurate Network Measurements 29
3.1.1. RTT latency 30
3.1.2. TCP Throughput 31
3.1.2.1. Maximum Transmission Unit 31
3.1.2.2. TCP Buffer Size 32
3.1.2.3. Hardware Limitations 32
3.1.3. TCP Congestion Control 33
3.1.3.1. HTCP 33
3.1.3.2. TCP CUBIC 34
3.1.3.3. BBR 34
3.1.3.4. BBRv2 Alpha 35
3.2. Executing Cloud Benchmarks 35
3.3. PerfKit Benchmarker 36
3.3.1. PerfKit Benchmarker Basic Example 37
3.3.2. PerfKit Benchmarker Example with config file 38
3.3.3. PerfKit Benchmarker Configurations 39
3.3.3.1. Latency ping and TCP_RR 40
3.3.3.2. Throughput with iPerf and Netperf 41

3.3.3.3. Packets per Second 42
3.3.3.4. On-Premise to Cloud Benchmarks 43
3.3.3.5. Cross-cloud Benchmarks 44
3.3.3.6. VPN Benchmarks 45
3.3.3.7. Kubernetes Benchmarks 48
3.3.3.8. Intra-zone Benchmarks 49
3.3.3.9. Inter-zone Benchmarks 49
3.3.3.10. Inter-Region Benchmarks 50
3.3.4. Inter-Region Latency Example and Results 51
3.3.5. Viewing and Analyzing Results 54
3.3.5.1. Visualizing Results with BigQuery and Data Studio 54
4 Architecture and Execution Efficiency of Benchmarks 58
4.1. Introduction 58
4.2. Efficiency in Benchmark Execution 59
4.2.1. Statistical Confidence 60
4.2.2. Data Observations 63
4.2.3. Confidence Interval Calculations 66
4.2.4. Experiment Setup 68
4.2.5. Results 68
4.3. Efficiency in Benchmarking Architecture 72
4.3.1. Batch Scheduling Optimization 74
4.3.2. Batch Scheduling Problem 74
4.3.3. Scheduling solution 76
4.3.4. Architecture of PKB_Scheduler 76
4.3.5. Graph Representation of Benchmarks and Virtual Machines 77

4.3.6. Efficient Scheduling 80
4.3.7. Maximum Matching 81
4.3.8. Edge Redistribution 84
4.3.9. Benchmark Execution 85
4.3.10. Experimental Results 86
4.4. Conclusions and Future Work 89
5 Network Analytics of Cloud Networks 92
5.1. Introduction 92
5.2. Throughput Profiles 94
5.2.1. Utilization-Concavity Coefficient 95
5.3. Measurements Collection 97
5.3.1. Measurement Tools 97
5.3.2. Google Cloud: Data Center Connections 98
5.3.3. Testbed with Hardware Network Emulators 100
5.4. Results 100
5.4.1. Google Cloud Connections 102
5.4.1.1. Outliers and Variance 103
5.4.2. Testbed Network Profiles 104
5.4.3. Parallel Flows 104
5.4.4. Congestion Control 106
5.4.4.1. Comparison of Congestion Control Algorithms 107
5.4.5. Retransmissions 108
5.4.6. Emulated Loss 110
5.4.6.1. Loss rate 110
5.4.7. Coefficients: Ideal and Loss Conditions 112

5.5. Conclusions and Future Work 115

6 Performance Forecasting and Anomaly Detection in Cloud Networks 117

6.1. Introduction 117

6.1.1. Throughput Prediction 117

6.1.2. Performance Anomaly Detection 118

6.1.3. Motivations 119

6.2. Data Collection and Analysis 120

6.2.1. Analysis of Data Features 122

6.3. Prediction Models 125

6.3.1. Error Metric 127

6.4. Throughput Prediction Results 128

6.5. Anomaly Detection 131

6.5.1. Data Preparation 132

6.5.2. Training 134

6.5.3. Anomaly Detection Results 134

6.6. Conclusions 135

7 Conclusion 137

7.1. Future Work 140

APPENDIX

A Source Code 143

Acronyms 144

GLOSSARY 145

BIBLIOGRAPHY 148

LIST OF FIGURES

Figure Page

2.1 Time series showing monthly airline passenger data 19

2.2 Time series decomposition of airline passenger data 20

3.1 PerfKit Benchmarker Architecture Diagram 36

3.2 Inter-Region Latency results for Google Cloud. All numbers are in milliseconds 53

3.3 Example of JSON output from PerfKit Benchmarker 54

3.4 Example of JSON output from PerfKit Benchmarker 56

4.1 Confidence Interval for 240 samples in region pair C. As we add more samples, the width of the CI becomes smaller until it reaches $\overline{x} \pm 2.5\%$. 62

4.2 Throughput by Region Pair 64

4.2a Region pair vs 1 flow throughput 64

4.2b Region pair vs 32 flow throughput 64

4.3 Average iPerf reported 32 flow throughput from us-east1 to us-west1 over 120 seconds with 0.5 second intervals 66

4.4 Throughput Results for iPerf both with and without autostop 69

4.4a 1 Flow Throughput 69

4.4b 32 Flow Throughput 69

4.5 Total Transfer for iPerf both with and without autostop 70

4.5a 1 Flow Total Transfer 70

4.5b 32 Flow Total Transfer 70

4.6 Throughput with calculated confidence interval 73

4.7 Flow diagram of PKB_scheduler program logic 78

4.8 Maximum matching of nodes and deletion of finished nodes 82

4.9 Nodes vs completion time of maximum matching and Edges vs completion time of maximum matching 83

4.10 Capture of VM usage and Allocation during each round of benchmarks 89

4.11 Quota usage for selected cloud region during each benchmarking round. 90

5.1 Throughput profiles and coefficients 96

5.1a Convex-concave $\mathcal{C}_{CC}$ 96

5.1b Utilization coefficient $\mathcal{C}_U$ 96

5.2 Diagram of connections in Virtual Private Cloud (VPC) network on google cloud. 98

5.3 Google Cloud with lines representing logical connections between region pairs, with RTTs in [1-350] ms range. 99

5.4 Physical testbed for target measurements. 101

5.4a dedicated 10GigE connection 101

5.4b servers, switches and hardware connection emulators 101

5.5 Throughput profiles of public cloud network using internal and external IPs for routing and their corresponding emulated testbed measurements for CUBIC. 102

5.5a Cloud Internal IP 1 Flow 102

5.5b Cloud Internal IP 5 Flows 102

5.5c Cloud Internal IP 10 Flows 102

5.5d Cloud External IP 1 Flow 102

5.5e Cloud External IP 5 Flows 102

5.5f Cloud External IP 10 Flows 102

5.5g Emulated Network 1 Flow 102

5.5h Emulated Network 5 Flows 102

5.5i Emulated Network 10 Flows 102

5.6 RTT measurements in milliseconds. 103

5.6a Google Cloud connections 103

5.6b testbed connections 103

5.7 Throughput profiles for measurements on public internal cloud network for CUBIC, HTCP and BBR. 106

5.8 Throughput profiles for measurements on public external cloud network for CUBIC, HTCP, and BBR. 106

5.9 Throughput profiles for measurements on testbed for CUBIC, HTCP, and BBR with no error 106

5.10 Throughput profiles for measurements on testbed for CUBIC, HTCP, and BBR with 1/1000 Error Rate 107

5.11 Public cloud network with TCP retransmissions for each TCP variant 109

5.11a TCP Retransmissions vs Latency (Internal IPs, 1 Flow) 109

5.11b TCP Retransmissions vs Latency (Internal IPs, 10 Flow) 109

5.12 Effect of loss rate on throughput. 111

5.13 Emulated network C_{UC} vs parallel flows 113

5.13a C_{UC} vs parallel flows in emulated network with CUBIC 113

5.13b C_{UC} vs parallel flows in emulated network with HTCP 113

5.13c C_{UC} vs parallel flows in emulated network with BBR 113

5.14 Comparison of C_{UC} of different TCP variants on emulated (with no error) and public cloud network 114

6.1 Throughput measurements for randomly selected connections in our dataset from mid February to mid August 2020. 121

6.2 Scatter plot of throughput and latency from all points in our dataset. A second degree polynomial line of best fit shows a slightly convex curve. 124

6.3 Comparison of predicted vs actual throughput for various models on a 50 day sample of test data 131

LIST OF TABLES

Table Page

3.1 SQL Schema for PerfKit Benchmarker results 55

4.1 Region pairs 63

4.2 Calculated Confidence Interval Statistics For 32 Thread Throughput Data 67

4.3 Percentage Change Between iPerf Mean Throughput And iPerf With Autostop Mean Throughput 72

4.4 Running time for each number of benchmarks 87

4.5 Running Time for different methods of benchmark scheduling 88

6.1 Correlation of variables at 1 time step ahead single stream Iperf throughput ... 123

6.2 Table Detailing the architectures of each tested model 126

6.3 Result Error and Correlation metrics for all models for multivariate iPerf 1 stream throughput 129

6.5 Result Error and Correlation metrics for all models for multivariate iPerf 32 stream throughput 129

6.7 Result Error and Correlation metrics for all models for univariate iPerf 1 stream throughput 130

6.9 Anomaly detection F1, Precision, and Recall scores for iPerf 32 stream throughput anomalies 135

Chapter 1

Introduction

Cloud and cloud-connected networks continue to expand to meet emerging demands as more applications move to being fully hosted in cloud environments. It has become increasingly clear that it is essential that users, businesses, and providers understand the performance dynamics of their networks at any point in time and are able to leverage that knowledge to maintain a robust and stable network. This includes taking accurate performance measurements, forecasting network performance at different time resolutions that are of interest, identifying performance anomalies as they occur. However, some performance measurements can be costly so we also must ensure that we are taking measurements as efficiently as possible while maintaining a high level of accuracy.

For a individual user or company, hosting your own servers and network infrastructure can be a costly and complex endeavor involving hardware, real estate, maintenance, and labor costs as well as additional unforeseen expenses. Over the past decade, there has been a move towards using third party services providers. These providers host a variety of services from databases and file storage, to high performance compute clusters and virtualized infrastructure. Collectively these services are known as cloud services and the service providers as cloud providers. These services largely rely on economies of scale for profitability. By moving services to the cloud, users trade large upfront costs and continuous maintenance for predictable expenses.

Additionally, services on the cloud have the ability to scale and expand to new regions easily. Large cloud providers have globe-spanning networks of data centers often connected by a network of private fiber optic links. Thus it becomes relatively simple to deploy an

application on servers close to the target audience no matter their location. This also has advantages when it comes to data backup and disaster recovery scenarios. We are focused specifically on these cloud networks, both connecting Virtual Machines (VMs) in a single data center as well as connecting VMs in geographically distinct data centers.

1.1. Motivation

Network intelligence is useful in scenarios ranging from high performance file transfers to cloud computing requiring orchestration of computations across cloud resources distributed in data centers across the globe. Cloud network connections tend to be multi-tenant and, consequently, the competing traffic partially determines the achievable performance at any given time, which can be a fraction of the total connection capacity and can have significant variance.

With the growing dependence on cloud resources, there has been a significant amount of research into performance when running in cloud environments. Most of this has tended to be towards application specific performance such as web server [1] or micro-service performance [2] in specific environments. Others have examined the network variability inherent in a multi-tenant environment [3] [4] and how to provide guarantees on network performance [5] [6]. There is less research on actually benchmarking cloud network performance and using that data in a constructive manner. There is also a significant amount of effort in analyzing and improving the performance of dedicated networks and private clouds such as High Performance Computing (HPC) data centers [7] and OpenStack implementations [8]. In Chapter 2, we will further explore previous work in this field.

In an ideal world, we would have a complete view of all aspects of a network and use all of the routing, traffic, and congestion information on all the routers to determine the throughput we are likely to be able to achieve on each specific route, such as might be the case with the control layer in a Software Defined Network (SDN) [9]. However, as users

operating on a cloud network, we often have a more constrained view. We only have the information that we can access via our endpoints and whatever network statistics are given to us from the cloud provider, which can be very limited. In this situation, analyzing network performance becomes a more difficult, yet not impossible, task. Given this more limited view of the network, we can still gather adequate data to draw conclusions about the performance characteristics of the network.

Additionally, the cloud providers themselves require comprehensive performance benchmarking of their cloud as a whole, both for their own operations and to provide numbers to inquiring customers. Currently, most of the benchmarking being done on these clouds is internal to the cloud provider and is not executed by third parties. As a part of our research, we perform an outside validation of these internal statistics with transparency to our configurations and reproducible benchmarks. These same benchmarks, methods, and data can be used by other cloud users to ensure that their network performance is adequate for their application needs and meets the minimums set by any Service Level Agreements (SLA) they may have with the cloud or network service provider. It is of great importance to both parties that any resources meet the requirements of their respective SLAs. Network benchmarking is an essential part of this. It allows the cloud provider to be confident that they can meet the minimum agreed upon performance metrics, as well as informing them which areas are in need of improvements or upgrades. Simultaneously, if the service does not meet the required metrics, the customer can use the benchmarks as proof of breach of contract.

All of this may sound like a relatively straightforward proposition of running a few tests, but performance benchmarking especially in a cloud environment can quickly grow in complexity and scope if we are unsure of exactly which metrics we may care about. We are mainly concerned with two important network performance metrics: network latency and throughput. Latency and throughput are, respectively, the amount of time packets take to

reach their destination and how much data we can transfer in a specified time frame. Even with a limited set of metrics, cloud performance is still quite a large area of study; we could look at private network performance, local to cloud performance, or performance across public internet. We focus our work primarily on cloud-to-cloud network performance using Transmission Control Protocol (TCP). This also becomes more complicated given the vastly different implementations of different cloud providers and the many options and possible configurations for each. It also includes 2 distinct levels of connections that have very different performance characteristics: intra-zone and inter-region. Intra-zone connections are high bandwidth and low latency, taking place usually within the same data center. Inter-region connections are between data centers located at different geographic locations, which causes higher latencies and more variable throughput. As such, part of our research also entails figuring out how to achieve the best possible network performance for the relevant situation. Benchmarking with sub-optimal configurations will always lead to sub-optimal results.

In order to take accurate measurements, we must be familiar with the benchmarking tools we are using and also be familiar with the needs of the system or application that we are benchmarking for. As much as we focus on running benchmarks here, we recognize that they do not exist in a vacuum. It is essential for the configurations of the tests we run to reflect the needs, or predicted needs, of the application. For example, if our application must send out millions of messages of a very small size in a short amount of time, it would not make sense to test the throughput of the server using a large packet size. This would give likely give higher throughput results, but these results might not reflect the performance of the application in reality. In fact the metric you are concerned with may not be bulk throughput, even with a small packet size, but instead packets per second.

If we are able to take consistent and accurate measurements, we can use those in a variety of ways. As previously discussed, the measurements can allow us to accurately determine our likely performance for a specific application or system. We can also use these measurements

taken at specific intervals over a period of time to monitor changes in performance over time. We can use historical data to create a model to predict future performance. An example application of this is to use current and predicted measurements to create a alert system for when it is the optimal time to perform a bulk transfer operation. Historical data can also be used for anomaly detection. If we have a historical baseline for network performance we can create a model to determine when our current measured performance deviates from our baseline.

1.2. Objectives and Contributions

To achieve a high level of network intelligence, we will meet the following objectives. First, we will define a way to effectively and efficiently benchmark network connections. This should include both how to take precise, accurate measurements on cloud environments and a way to find the optimal network parameters to give us the best possible network performance consistently. We will also consider how to best automate provisioning of virtual cloud infrastructure and the setup and execution of benchmarks. Another factor we must keep in mind is the price of running benchmarks and how to minimize it so that we can run benchmarks for the smallest cost possible, which becomes necessary when benchmarking at scale. To meet this objective, we will present a method of running throughput benchmarks for the minimum required time to achieve our desired level of accuracy.

Next, we will gather a large sample size of network performance data across a long period of time. To do this, we need to be able to run and schedule a large number of benchmarks to provide coverage over diverse cloud network connections. We also need to schedule these benchmarks to fit within the constraints of each Cloud API that we use and their various quotas and limitations. We will present a way to efficiently batch schedule a large number of network benchmarks in different regions while minimizing setup and tear down time of each individual benchmark as well as the total time of benchmark execution. This will help

ensure both that we can run all the benchmarks we need to in a limited time frame and that as little time is devoted to benchmarking as possible instead of other tasks.

After the infrastructure to run benchmarks is in place, we need to capture network performance data over several weeks and then we thoroughly examine the performance analytics from what we have gathered. This includes graphing throughput profiles, examining performance using different parameters and TCP variants, and comparing their concavity-utilization coefficients [7]. We also compare this data to emulated dedicated network connections with similar settings.

Using what we learned from this analysis, we can create a model to forecast future performance. To do this, we use several deep learning techniques that have not previously been applied to this problem. We will then use the performance data previously gathered to train and test these models and attempt to create the most accurate model possible. We will also discuss how this and similar techniques can be applied to network performance anomaly detection. Both of these uses for our data are important for our understanding and management of our network resources and we will show practical example applications that our forecasting and anomaly detection techniques can be used for.

1.3. Organization

The remainder of this dissertation will be organized as follows. In Chapter 2, we will be examining a variety of other research that we build upon or is related to this topic. Then in Chapter 3, we will explain how to effectively test and gather performance data. In Chapter 4, we discuss how we can gather this performance data more efficiently in regards to time, cost, and data transfer. Then we look at the throughput profiles and dynamics of cloud networks and how we can use emulated networks to draw conclusions above actual networks in Chapter 5. In Chapter 6 we look at using the network performance data we have gathered to predict future performance and detect anomalies in the network. Lastly, in Chapter 7,

we review the work that we have completed and reflect on the contributions that we have made.

Chapter 2

Related Work

Performance analysis of network is a topic of study that has existed for as long as computer networks themselves and similar to how networks have evolved since the days of ARPANET, so have the techniques used to test and analyze their performance. In this chapter, we will discuss the previous and similar work done in the areas relating to this work.

2.1. Network Performance Benchmarking

Let us begin by discussing the field of performance benchmarking as it pertains to networks. It was clear early into the advent of the Internet and large scale computer networks that it would be necessary to develop tools to assist in measuring network performance. One early example is in 1987 when Aronoff et al. describe tools developed by the National Bureau of Standards (NBS) for taking consistent transport layer performance measurements on a network testbed [10]. The purpose of these tools was to make taking measurements and configuring experiments across distributed systems easier. A large part of the the reason why we perform benchmarks is to test our network and system configurations to ensure that we are achieving optimal performance and to test the system before it is stressed by real world scenarios.

When it comes to benchmarking there are two general categories: fine grain, and coarse grain [11]. Fine grain benchmarks focus on measuring individual operations, such as number of Floating Point Operations per Second (FLOPS) or Round Trip Time (RTT) latency for packets across a network path. Coarse grain benchmarks tend to involve something like running a program with a given set of inputs and measuring the execution time and other

metrics about the program's execution such as CPU utilization. When deciding on how to measure the performance of our system or network, we must consider which of these is more appropriate. In either case the benchmark should represent real world workloads as closely as possibly. To get as close to real workloads as possible, we can perform application layer benchmarking [12]. This would be a form of coarse grain benchmarking where we test using the actual application of interest. If we are concerned about file transfer performance using File Transfer Protocol (FTP), we would test how FTP performs under various circumstances and configurations.

On the other hand, fine grain testing might be preferable if there are a wide variety of applications we are interested in that all depend on a certain operation. For example, we perform fine grain network throughput tests to ascertain the average or maximum throughput we are likely to achieve across a specific network path. We do this as opposed to executing actual program workloads because TCP network throughput performance is applicable to many programs and has the potential to be a bottleneck for those programs. In some sense, this actually makes fine grain benchmarks more generalizable than coarse grain benchmarks in many instances.

If we can not perform measurements on an actual network, we can instead choose to simulate or emulate a network and perform our measurements there [13] [14]. This can allow us to tweak our network configuration before deploying to a physical network and can also be more cost efficient than buying hardware such as in the case of Caini et al. evaluating the performance of different TCP variants across a simulated satellite network [15]. It is much more efficient to simulate the characteristics of a satellite network than to perform extensive benchmarking across the production network. One fairly widely used network simulator is Mininet, which allows users to simulate large Software Defined Network (SDN) on their personal computers [16]. If we can correctly simulate the characteristics of the original

network, these can provide us fairly accurate data of how our applications or protocols will perform on the actual network.

2.1.1. Performance Measurements and Benchmarking Tools

To actually measure network performance, we can either gather passive or active measurements. Passive measurements record existing traffic on a network and uses that to estimate throughput and latency statistics. Active measurements inject additional traffic into a network [17]. These have the disadvantage of adding congestion, but have the benefit of producing repeatable measurements. The type of performance measurements we gather depend on whether our network can handle the additional workload of active measurements. Our work relies on these active measurements and we believe they are worth the cost of the potential disruption if performed in a conservative and efficient manner. In fact, multiple later chapters focus on minimizing the workload and impact of these active measurements. The remainder of this section will focus on active measurement techniques and tools.

2.1.1.1. Network Throughput Measurements

For network throughput, active measurement techniques can further be divided into estimation and direct measurement methods. Both involve adding additional workloads to the network, but estimation methods generally involve significantly fewer packets. One common type of estimation is available bandwidth estimation [18]. Available bandwidth is the portion of a path's total end-to-end capacity that is not used during a specific interval [19]. This is usually estimated using various techniques relating to the difference in delay between TCP packets sent in a sequence [20]. Estimation is usually lighter weight than direct measurement, but can also be less accurate. In particular, these delay based estimates are much less accurate on multi-Gigabit network paths [21], which encompass the cloud network paths we are primarily interested in.

In addition to available bandwidth estimations, there are also studies for estimating throughput in specific circumstances. De Silva et al. looked at estimating throughput specifically on mobile networks. They were able to use 1MB of data to predict the throughput for 5-20MB downloads. This estimate relied on an explicit model of the dynamics of TCP CUBIC [22]. This seems to estimate throughput accurately, but is limited to one variant of TCP operating under very specific circumstances. If we want to estimate throughput for another TCP congestion control algorithm, we would need to model that as well.

Our work is focused on direct achieved throughput measurements. Achieved throughput is the rate of network data transfer we were actually able to attain during a specified period. This is different from bandwidth, which is the maximum theoretical capacity a link is capable of. For example, the stated bandwidth of a link might be 10 Gbits/sec, but the achieved throughput we actually measure might only be 8 Gbits/sec. Throughput is dependent on the protocol being used, the congestion control algorithm, and variables such as the TCP buffer size and MTU. One unfortunate side effect of measuring throughput in this manner is the stress it can put on a network. Performing a large amount of throughput tests to find the maximum capacity of a network could disrupt other traffic on that network and take up a large amount of resources that could be otherwise used. When performing tests on a cloud provider, it can also be expensive. Most providers have charges based on the amount of data that egresses from their network or moves between their data centers. This is fine when measuring throughput between machines in the same data center, but the cost can quickly add up when taking measurements from machines in different data centers or to a machine outside the provider's network.

There are several tools available for measuring achieved network throughput. The idea behind these tools are all quite similar. To measure achieved throughput, you execute a sender and a receiver program on two machines. The direct measurement tool then generates traffic on the sender machine and sends it across the network to the receiver machine.

Throughput is then easily calculated as the average amount of data in bytes/sec sent/received during the test. Though on their surface the tools operate in a similar manner, their implementations have some important details that can effect the measured results particularly for high latency or high bandwidth links. The first tool that we use to measure throughput is iPerf, specifically iPerf2 [23]. Though there also exists an iPerf3 [24], iPerf2 is also still being actively maintained and improved at the time of writing. iPerf2 tends to be better than iPerf3 for high bandwidth use cases. When measuring throughput on high bandwidth links, one often has to use multiple parallel streams of data to reach the maximum possible throughput. iPerf2 creates subprocesses to send multiple streams each with their own data generator, while iPerf3 uses the same process for all of its streams. This can be a bottleneck in some cases, leading to inaccurate measurements of the true throughput of the network link. As such, when we refer to iPerf in this document, we are referring to iPerf2.

We also make use of Netperf [25] to perform throughput measurements. We have found this to be a very versatile and powerful tool. It has a variety of protocols that can be used in different directions of flow and has the best capabilities for performing packets per second testing and multi-server testing from the tools we have used.

2.1.1.2. Network Latency Measurements

To measure latency, or RTT, the standard tool implemented and ubiquitous on all command-line interfaces is ping. This utility measures latency using packets. By default it sends 1 packet per second for a specified period of time. There are other similar implementations such as nping, and hping that offer additional features such as using different packet types like TCP. These can be useful as networks are not uniform in how they handle different types of packets. They might treat ICMP packets differently to TCP packets. This is a problem we have encountered before when gathering measurement data, as one of the networks we were using completely blocked or dropped our ICMP packets from ping, so we

had to use TCP packets with nping. Because of these potential performance differences, it can be useful to measure latency using the protocol that you are interested in. Latency can also be subject to congestion on the network and the endpoints. If there is delay in processing the packets, that can show up as an increase in the RTT.

2.1.1.3. Benchmark Suites

Beyond these tools, have been several efforts to make benchmarking easier and more convenient for users. These tools, often referred to as benchmark suites, compile several types of measurements or benchmarks and provide a convenient wrapper for their usage. They can also encapsulate a reporting and configuration environment. Some examples of these include BenchIT [26] and HPCBench, which is a benchmark suite for high performance computing clusters [17]. Perhaps the most widely used benchmark suites are those produced and maintained by the Standard Performance Evaluation Corporation (SPEC). They provide benchmark suites for a variety of scenarios, including Cloud IaaS environments [27].

2.1.2. Cloud Benchmarking

Cloud computing is a relatively new field that has exploded in size. With this rise in use, there has been extensive research done on how to adapt the technologies people use on local systems, with those being hosted in the cloud. The topic of many papers in regards to Cloud Computing deal with how professionals should classify cloud technologies, and how they can adapt current standards to the new platform. Needless to say, we are not the first to explore the topic of performing benchmarks in cloud environments.

For benchmarking, cloud computing represents a new challenge because the physical architecture is obscured from the user. Not being able to see all of the specifications of cloud machines has forced researchers to find other ways to benchmark cloud machines. Many

research topics today deal with this new way of benchmarking and analyze the most efficient and accurate ways to rate cloud machines. With the move from physical machines to virtual ones, benchmarks had to be overhauled to accurately analyze the new systems. Klaver et al. propose an independent cloud monitor that can be used to accurately benchmark and compare virtual machines from different cloud providers [28]. This would allow cloud consumers to hold cloud providers accountable for the machine performance they are paying for, and would allow them to switch to other providers who offer better performance. In their paper, *To Cloud or Not to Cloud: A Study of Trade-offs between In-house and Outsourced Virtual Private Networks* [29], the authors chose to investigate the optimal Virtual Private Network (VPN) that can allow end users to access cloud resources. The paper details how pfSense can offer users the greatest throughput, but Cisco's ASA 5520 VPN offers easy setup, LDAP compatibility, and throughput that is sufficient for work.

In *Fair Benchmarking for Cloud Computing Systems*, the authors looked into micro benchmarks and determined if cloud virtual machines were performing as they were advertised [30]. Their research revealed that cloud machines often had a variation in their performance, even when the same machine was made on the same cloud service. They proposed making a web portal for comparing providers and benchmarks to increase the transparency between the cloud provider and consumer. An insight the researchers provided was a switch from paying a standard price for a machine, to paying for performance, measured by Quality of Service and Service Level Agreements.

Many of the new benchmarks being done are manual, where machines have to be spun up, specifications and dependencies downloaded, the benchmark run, and then the machine is spun down. This process can pose serious issues for organizations that must run benchmarks across hundreds of different machines. In response to this challenge, automation tools have been created that can automate this process and allow researchers to select a benchmark and then run it against virtual machines that are automatically spun up and down for the

tests. In the paper, *Smart CloudBench*, researchers detail the automating of benchmarks, from a highly manual process, to one that is nearly hands-off [31]. They developed an application that could automate the process of running benchmarks so that researchers would not need to manually setup the machines, dependencies, and network rules. There have been multiple similar efforts to this, including Cloud WorkBench [32] and the Automated Performance Benchmarking Platform [33]. All of these tools have the same goal of making cloud benchmarking more accessable to the end user by automating large parts of the process. For the cloud benchmarks executed for this work, we have chosen to use the tool PerfKit Benchmarker (PKB) [34]. PerfKit Benchmarker is an open source benchmarking tool that automates the benchmarking process of cloud-based virtual machines and services. It works with all of the major cloud service providers and provides simple and configurable access to a wide range of benchmarks. This is a project that we have contributed to over several years and we discuss it in extensive detail in Chapter 3.

2.2. Network Performance Analytics

Once we have obtained measurements from our cloud benchmarking efforts, we must analyze this data. In this section, we will be discussing work in various types of network performance analysis. The problem of achieving high performance, however it is defined, across networks is a common and persistent one. As such, it has been previously examined by numerous groups.

A large part of this analysis is comparing different settings and configurations to figure out how we can achieve the best performance possible. A major contributing factor to this is the congestion control algorithm that we choose to use. Our study includes tests of widely used TCP congestion control algorithms (also referred to as TCP variants) such as TCP CUBIC, as well as more recent TCP congestion control algorithms like BBR and BBRv2.

In [35], the authors looked at performance on high bandwidth optical networks and tested different TCP congestion control algorithms and TCP parameters across this type of network. They found that STCP performs best for single flow, and CUBIC is most fair. There are several other studies that also compare the efficacy of different TCP variants cross different network types [36] [37].

Multiple studies also focus exclusively on BBR. Crichigno et al. look at how maximum segment size and the number of parallel streams affects large file transfer performance when using TCP BBR, finding that BBR shows a larger performance increase than other TCP variants when using multiple parallel streams [38]. They found that BBR performs favorably against loss based congestion control algorithms and that BBR shows more improvement than its competitors when using parallel and an increased MSS. Jaegar et al. show how to take reproducible measurements of TCP BBR [39] and Cao et al. examine what situations BBR performs best in [40]. Ha et al. examined the performance of BBR in cloud networks and found that BBR under-performs significantly when running on a shared CPU that is scheduled between multiple users [41]. We have not observed this as our tests are on VM types that have dedicated processors. It also may be much less fair when competing with other congestion control algorithms [42]. Likewise, Kfourney et al. [43] look at at BBRv2 performance on an emulated network and focuses on how it differs from the original BBR in areas such as parallel flows, bufferbloat, fairness to other variants of TCP, and how it deals with packet loss.

In our work, the cloud network links we benchmark all have at least 10 Gigabits of advertised bandwidth. Often analysis done on much smaller capacity links is not as applicable to these larger links. There are several studies involving TCP performance over 10 Gigabit connections [35] [44], including studies about the performance of specific TCP variants, such as CUBIC [45], BBR [46], and BBRv2 [47]. Settlemyer et al. [44] looks at TCP transfers over 10 Gbps links for wide-area dedicated connections and emulated connections. They

used differential regression to compare measured collected on physical and emulated links and found that measurements on the emulated infrastructure could be used to generate good estimates for the physical infrastructure at a fraction of the cost.

Likewise, several papers have covered the evaluation and comparison of different TCP congestion control algorithms [36] [37] [48] on 10 Gigabit networks; however, there has been relatively little work involving measurements on production cloud networks. Ganji et al. looked at which TCP variant to use as a cloud tenant based upon application type [49]. This study focused on application traffic for specific workloads, such as streaming and distributed IO in a single availability zone. Jouet et al. explored how to optimally tune TCP parameters for cloud networks [50]. Our work differs from these in that we test multiple TCP congestion control algorithms across a range of RTTs (between availability zones) on both a 10 Gbps public cloud network as well as an emulated network of the same capacity.

Other researchers have looked at end to end data movement across a system, both with physical and emulated connections [51]. They have also examined regression methods to estimate the differences between these types of connections [44]. File transfer dynamics tends to be more complex, since there are more possible bottlenecks to consider. For our research, we are only looking at network performance, or memory to memory transfers. There are also multiple studies that focus on the effects of using parallel TCP streams for improved throughput performance [52] [53]. There has also been work in creating and using coefficients for comparing different networks [7] [54]. This includes the utilization concavity coefficient, which we use to compare our network throughput profiles and will explore more in Chapter 5.

2.3. Network Performance Forecasting

Another key part of our work is using the data we have gathered from performance benchmarking to gain insight into how these network perform over time, how they might perform in the future, and whether they are performing how they have historically.

2.4. Time Series Analysis

Before we discuss efforts into throughput prediction for networks, we should first discuss the topic of time series and some of the methods that are commonly used to analyze them. This will help us understand techniques we use in later sections.

Time series analysis refers to methods relating to working with Time series data, which refers to any data set where the data points are indexed by time. Examples of this include a wide range of domains from stock prices, to climate data, to network throughput and latency data. Time series analysis consists of the study of data to find patterns, statistics, and meaningful insight, as well as using existing data to try to predict future data points, which is also called time series forecasting. There is a second branch of time series analysis which is primarily concerned with signal analysis and related topics which are not immediately relevant to our discussion, so we will not be covering them. This is not a new field and some of its methods, many of which are still widely used, originate decades ago.

2.4.1. Time series statistics

In this section, we will discuss different statistical properties that are essential to understand when dealing with time series data.

2.4.1.1. Stationarity, Seasonality, and Trends

For time series, stationarity is the property that the mean and variance of a series does not change over time and there is also no seasonality. Seasonality is when the data goes through cycles that repeat regularly over time, such as one might see when looking at the daily average temperature over the course of several years. It is also possible that a time series has a trend over time. This is where the moving average of the series either increases or decreases. In Fig. 2.2, we show an often used example dataset consisting of monthly passengers for an airline over time from 1949 to 1960. Here, we can easily see that this dataset has both a trend and seasonality and is not stationary. The number of passengers tends to peak in the summer months and dips in the winter. This is the seasonality. Year over year, the number of passengers increases. This is the trend.

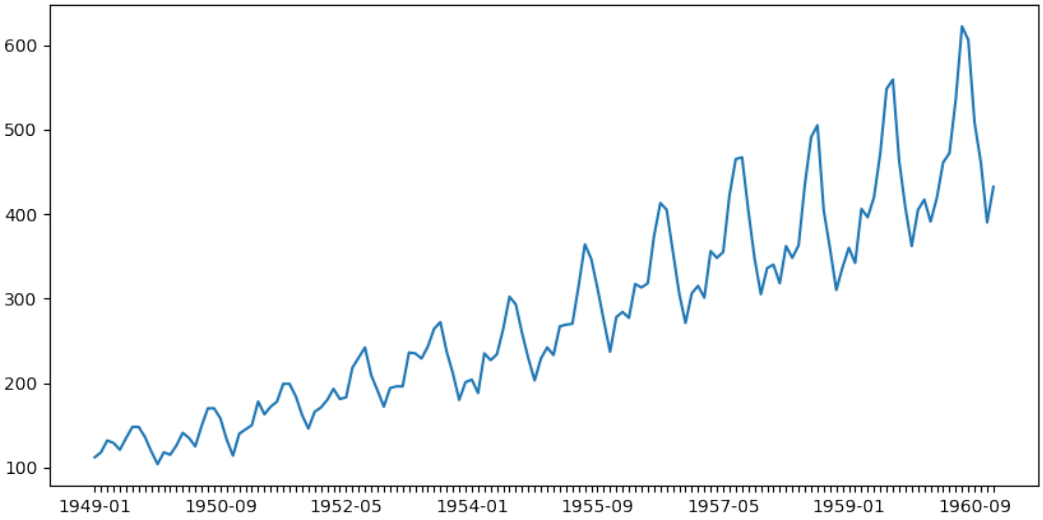


Figure 2.1: Time series showing monthly airline passenger data

We can also decompose a time series into its component parts. We show the decomposition of the airline passenger dataset in Fig. 2.2. Using this method, we end up with a trend component, a seasonal component, and then a residual component. The residual component shows variability that is not accounted for by either the seasonal or trend components.

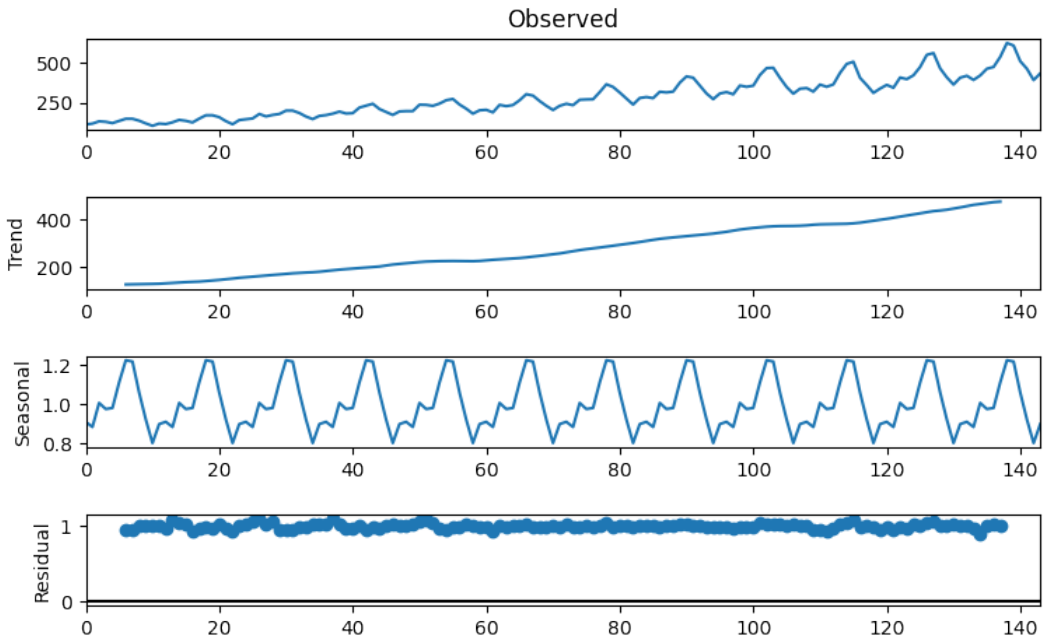


Figure 2.2: Time series decomposition of airline passenger data

If a time series is not stationary and the algorithm we are using requires it to be, we can usually induce stationarity through mathematical transformations. This can be done in several ways, depending on the nature of the series. If the series has one of more distinct jumps in value, like a step function, we can divide the dataset into windows, where each window is stationary. If a series has a trend, we can try to de-trend the data. We can do this by finding the rolling mean of the series at some given window size, k. Then for each time step t, we subtract the rolling mean at that time step.

$$\Delta t_i = t_i - \frac{\sum_{j=i-k}^{i-1} t_j}{k} \tag{2.1}$$

We can also try to difference the data. This is computing the difference between each consecutive observation. Then the values of the series become the change from the previous value. For for time step t, the new value of that time step would be

$$\Delta t_i = t_i - (t_{i-1}) \tag{2.2}$$

2.4.2. Moving Average

A moving average, or a Simple Moving Average (SMA) is about as simple of a prediction method as you can get. It simply predicts the next value based off of the average of the previous n values. We can also use this to smooth our data, reducing the variance in each individual data point to see a clearer picture of the trend.

2.4.3. ARIMA

ARIMA, one of the most well known and prolific models, has been used by several studies for throughput prediction on different types of networks [55] [56]. These models have the advantages of being simple and often very effective. They combine a simple moving average with an autoregressive component to predict a future value based off of the weighted summation of a number of lagged values and forecasted errors.

They do require that the time series be stationary. This property is definitely not guaranteed to be found in any arbitrary time series, but as we have discussed, the data can often be transformed or normalized to 'induce' stationarity. There are also specific variants of ARIMA such as SARIMA that are designed to deal with seasonality. Just like the classic ARIMA model, the other history based methods here all use a function based on some number of lagged values for prediction.

2.4.4. Support Vector Regression

Support Vector Regression constructs a curve of best fit from past values. To predict future results, it tries to fit them to the same curve. In Mirza et al, the authors employ a support vector regression model that uses previous throughput as well as loss and queuing delay measurements to help predict throughput [57], which they found increased the overall accuracy of the model. A similar multivariate approach can be taken with ARIMA, which is referred to as Vector ARIMA or VARIMA.

2.5. Throughput Prediction

Now let us look at times series analysis as it applies to the domain of throughput prediction. There have been numerous studies that have looked at the problem of TCP throughput prediction and prediction methods generally fall into one of two broad categories: formula based methods, and history based methods.

2.5.1. Formula-based TCP Throughput Prediction Methods

Several formulas have been developed and often used to predict throughput between two endpoints [58] [59]. These usually show throughput as a function of network and endpoint traits, such as round trip latency, loss rate, and TCP buffer size, and the dynamics of the congestion algorithm. These metrics are plugged into an equation to yield the predicted throughput at a given point in the future. Their accuracy suffers when there are network or metric changes that can't be accounted for in the formula's variables or occur during the TCP transfer. That is to say, they can generally be less adaptable. On the positive side, they benefit from being very quick, which is useful when predicting throughput on small timescales, milliseconds or seconds in the future. These methods usually work better when developed for and applied to a specific area, such as throughput for mobile network users [60].

Overall, we like these methods for their speed, but they are generally not applicable to our dataset, which shows throughput at the scale of days instead of seconds.

2.5.2. History-based TCP Throughput Prediction Methods The other category of throughput prediction is history based methods. What unites these methods and defines this category is the use of some number of past samples to predict future network throughput. This includes classical time series analysis models such as a simple moving average (SMA), an autoregressive integrated moving average (ARIMA) or one of its many derivatives, linear support vector regression (SVR) [61], as well as more modern machine learning methods and various architectures of neural networks.

ARIMA, one of the most well known and prolific models, has been used by several studies for throughput prediction on different types of networks [55] [56]. These models have the advantages of being simple and often very effective. They combine a simple moving average with an autoregressive component to predict a future value based off of the weighted summation of a number of lagged values and forecast errors. They do require that the time series they are used with is stationary, meaning that the mean and variance does not change over time and there is no seasonality. This property is definitely not guaranteed to be found in any arbitrary time series, but data can often be transformed or normalized to 'induce' stationarity. There are also specific variants of ARIMA such as SARIMA that are designed to deal with seasonality. Just like the classic ARIMA model, the other history based methods here all use a function based on some number of lagged values for prediction.

Support Vector Regression relies on using past values to construct a curve of best fit, then trying to predict future results by fitting them on the curve. In Mirza et al, the authors employ a support vector regression model that uses previous throughput as well as loss and queuing delay measurements to help predict throughput [57], which they found increased the

overall accuracy of the model. A similar multivariate approach can be taken with ARIMA, which is referred to as Vector ARIMA or VARIMA.

A custom machine learning based method is proposed to identify concave-convex regions of the throughput profile as a function of connection round trip time in [62] for data transport infrastructures with dedicated connections. Several conventional machine learning based methods are shown to fail in capturing the critical concave-convex regions.

Additionally, among most of the research on throughput prediction, each focuses on a relatively narrow use case, such as streaming across mobile networks or throughput for specific purpose-built applications. We focus on the use case of day to day bulk throughput prediction in a cloud environment. This comes with its own set of challenges. Most of the existing research in this area look at throughput at a much finer granularity, predicting on the time scale of seconds rather than days. From our experience, TCP tends to be much more predictable on those smaller timescales, which is one reason why formula based methods that operate at this timescale have found success.

2.5.3. Neural Networks Applied to Time Series Prediction

If we slightly widen our view beyond the topic of throughput prediction, we find a significant volume of work detailing different methods for time series prediction that could easily be applied to the throughput prediction problem. In this section, we will examine neural network architectures for time series prediction, any of which could possibly be applied to our use case. As with any time series prediction method, the goal is to find the most accurate model and hopefully outperform traditional statistical models like ARIMA, which can often be surprisingly difficult.

While attempts have been made to apply standard multi-layer perceptrons (MLP) to time series forecasting, a fully connected neural network architecture is not terribly conducive to

this task as it struggles with adequately modeling the necessary temporal relations. Recurrent neural network (RNN) based approaches evolved in part to address this inadequacy. RNNs have a recursive structure that allows connections and learning across temporal sequences. This often proves advantageous over MLPs when it comes to time series prediction. As such, they have frequently been applied to time series problems including throughput prediction [63]. However, RNNs also suffer a widely known flaw: the vanishing gradient problem [64], which becomes apparent when you have to perform back-propagation across a large number of recurrent layers. The further back you propagate, the smaller the gradient becomes. This can make training ineffective and leads to RNNs having an effectively 'short memory'.

There are two well known improvements on RNNs that attempt to alleviate the vanishing gradient problem. The first is the gated recurrent unit (GRU) network [65] and the second is the long short-term memory (LSTM) network [66]. Both of these approaches have been shown to exhibit improved performance over a simple RNN when predicting on time series data [67].

Though most commonly used for problems such as image recognition and classification, convolutional neural networks (CNNs) have also been applied to time series problems [68]. Similar to how they identify patterns on images, CNN's are able to extract patterns and correlations from multivariate time series data.

In practice, such architectures are usually not employed on their own and are combined into a hybrid model, such as one that combines a CNN and an RNN or LSTM [69]. The CNN is useful for finding correlations between variables and the LSTM finds temporal relations. Together, they can collaborate into a valuable tool for multivariate time series. We also find significant support for the idea of combining neural networks with more traditional time series analysis techniques, such as a proposed hybrid ARIMA-ANN architecture [70] and a

hybrid model that combines a CNN, a GRU, and a linear autoregressive element that operate in parallel in an architecture called LSTNet [71].

2.6. Anomaly Detection Techniques

We can also use our collected network performance data to help us detect anomalies in our data. Previously anomaly detection techniques have been applied to both performance related anomalies and security anomalies such as with intrusion detection. In this subsection we will look at a variety of other work in anomaly detection. This includes detection techniques ranging from time tested statistical tests to more experimental deep learning models. It also includes anomaly detection papers in our specific area of interest.

All of the techniques we discuss share a common methodology. That is, to figure out what 'normal' data should look like and create a measurement such that we can determine how 'normal' or 'abnormal' a particular data point is. If it is dissimilar enough to our other data, we can label it an anomaly. This can apply to both a single data point, an outlier, or a longer range or pattern of data.

There are several commonly used methodologies to identify anomalies and outliers in time series, as defined in [72]. The first of these that we will discuss are statistical methods. Some examples of these include using the euclidean distance or Kalman filtering [73] such as AnomalyDetect [74], which is designed to detect anomalies in cloud based virtual machines. The advantage of these methods is that the probability distribution can be easy to intuitively explain. However, they often do not work as well if there is a large number of outliers or it is a more complex, multivariate dataset.

Clustering based anomaly detection uses k-means based clustering to detect outliers and anomalies. By partitioning data into clusters that are most similar, we can identify outliers by observing how far outside of existing clusters they are, such as authors did in [75] and [76]. [77].

We can also use classification algorithms for anomaly detection such as SVMs or logistic regression. Using a SVM for this task has the advantage of working well in high dimensional space and being memory efficient [78].

There are also several neural network based models that have been applied to anomaly detection. These include MLPs, LSTM networks [79] [80], Auto-Encoders [81] to Generative Adversarial Networks (GANs) [82] [83]. LSTMs help extract temporal features from a time series dataset. Auto-Encoders help us ignore noise in data and reproduce its most important features by first encoding data in a reduced dimensional form and then attempted to reconstruct the original input from that reduced form. When using this for anomaly detection, we are looking for instances when the reconstruction performs worse than usual, as the network will be trained using only normal samples and will be less able to accurately reconstruct anomalous data. For all of these architectures, their specific parameters such as the number of layers, number of LSTM cells, window size, and anomaly threshold can have a significant effect on their accuracy. Therefore, it is essential that no matter what architecture we end up using, we perform a thorough parameter sweep to obtain the most optimal model.

One of the more interesting models in this category is a LSTM-GAN-XGBOOST [80] model that uses first uses an LSTM to extract temporal features, then a GAN to perform an efficient dimensionality reduction. It then puts these extracted features through an ensemble learning algorithm, XGBOOST, to classify the features and assign a score based on the likelihood of a data point being anomalous.

We also consider anomaly detection research that specifically applies to networks and performance. There has been research exploring anomaly detection applied to various types of networks and metrics. Many of these focus on security and DDoS detection. We are more interested in the ones that focus on performance anomalies for the sake of maintaining consistent performance for Service Level Agreements. Qin et al. used an LSTM network for anomaly detection for network throughput on a large IP bearer network [84]. They found

it performed with high precision except in cases where a port did not have stable periodic throughput. Gaikwad et al. [85] looked at performance anomalies for scientific workloads on the cloud. They used an auto-regression (AR) based anomaly detection technique.

Chapter 3

Network Measurements and Performance Benchmarking

In order to predict, or even analyze our network performance, we first need to be able to measure performance. There are several ways to do this and many tools have been created provide similar functionality. Before we discuss gathering measurements though, it is important to figure out what we even care about measuring. In this chapter we will discuss how we perform cloud benchmarking and how we tune machines for optimal network performance.

3.1. Accurate Network Measurements

When performing network tests it is important that we get as accurate of results as possible. Accurate, however, is relative to the target that we are trying to measure. For our performance studies, we are trying to find the optimal performance that the machine or network under test is capable of delivering under any specific circumstances. This means that in order for us to obtain accurate results, we must ensure that we are able to get optimal performance. For performance, we are measuring network throughput and round trip latency. These metrics depend on several factors and we try to adjust our network and host parameters to achieve the best possible throughput and latency results that the machines are capable of. Often, using default configurations will result in sub-optimal performance, so we will be discussing how we can adjust these parameters to improve performance.

3.1.1. RTT latency

For improving inter-region latency there is not a lot we can do from a software perspective. It depends largely on hardware and infrastructure setup both in a data-center and across a wide area network (WAN). If we have a server and want to improve our latency with clients, the best thing we can do is move our server closer to those clients. This is the core idea behind content delivery networks (CDNs). For specific applications, caching can also help by removing the latency associated with disk I/O. Congestion also has an effect on latency, as the more time packets spend waiting in queues, the higher the latency will be. This effect can sometimes be minimized by using a congestion control algorithm like BBR, which attempts to operate at the level of minimum latency. However, if the network path is very congested regardless, this my not help latency.

If we are concerned with intra-zone performance (within the same data center) there is more that we can do to reduce latency such as disabling host firewalls and enabling busy-polling. These changes can shave nanoseconds off our latency, which may be useful in tightly-coupled HPC environments. This will only result in a noticeable performance improvement if the latency of the network path was already in the sub-millisecond range. These should also only be used in specific circumstances because disabling firewalls can have adverse security implications.

To measure RTT accurately, we need to take certain external factors into account and ensure that we are using the correct settings for our chosen tool, be it ping, nping, hping, or Netperf TCP_RR. If the machines are very close together, i.e. housed in the same datacenter or have a sub-millisecond RTT, the interval time between pings can have a significant effect on our measured RTT. Ping, which has a 1 second interval between transactions by default, will show a significantly higher RTT than Netperf TCP_RR, which sends the next transaction immediately upon completion of the previous transaction.

For both ping and Netperf, a flag exists to manually set the interval time (-i for ping and -w for Netperf). So, if we are using ping to measure sub-millisecond latency, we should reduce the interval size to 1 millisecond to obtain an optimal RTT measurement. This is also often more consistent with the transaction rate of real world applications. If we are using Netperf in the same situation, we can use default settings to get a optimal measurement, or use the -w flag if we want to send at a specific interval spacing.

3.1.2. TCP Throughput

3.1.2.1. Maximum Transmission Unit

To achieve optimal throughput, we must use the correct settings for the network path we are using. Specifically, we will discuss TCP throughput. The first thing we should do is increase the MTU, or the largest size of packet or segment that can be passed along the network. This is the frame size minus overhead of the frame header. On standard Ethernet connections, the frame size is 1518 bytes and the overhead is 18 bytes, giving us an MTU of 1500 bytes. In most cases, the MTU for the machines on a throughput test should be set to this maximum of 1500 bytes or sometimes 1460, depending upon what the network supports. In some cases, the network may support larger frame sizes, or jumbo frames. If this is the case, the MTU should be increased to maximum allowable size. This can result in significantly higher data throughput, because a smaller percentage of the bandwidth will be dedicated to header data. If you are primarily concerned about the performance of an application that sends out many smaller packets, a smaller MTU may be sufficient.

3.1.2.2. TCP Buffer Size

The next settings we need to adjust are the TCP send and receive buffer sizes. On most Linux based operating systems, these are located in /etc/sysctl.conf as net.ipv4.tcp_wmem and net.ipv4.tcp_rmem, respectively. Here we can set a minimum, default, and maximum size for each buffer in bytes. By default, these buffers tend to be quite small, around 64 KB. We can get a good idea what to set the maximum buffer size to by using the Bandwidth-Delay Product. This is found using the theoretical maximum throughput of a connection multiplied by the RTT of the connection.

$$BDP = BW \times RTT \tag{3.1}$$

It is essential to increase these buffer sizes when we have a high RTT, a high maximum bandwidth, or both. If we do not, we will be severely limiting our achievable throughput. Generally, we only have to set the maximum buffer size and leave the default and minimum sizes as they are. If TCP window scaling is enabled, the operating system should increase the TCP send and receive buffers to an appropriate size up to the maximum automatically. For most of our tests, which often include long fat connections, we use maximum TCP send and receive buffers of at least 500 MB.

3.1.2.3. Hardware Limitations

If attempting to achieve multi-gigabit throughput performance, we may also be limited by the machine we are using or the network itself. High throughput can be both CPU and memory intensive, so we should check on our CPU utilization to ensure that we are not bottlenecked by our CPU performance or number of CPUs. In a cloud environment, this can usually be remedied simply by provisioning a machine with more CPU cores.

Network capacity can also be limited by other cloud parameters such as which specific machine type we are using and whether or not we are using additional premium features.

3.1.3. TCP Congestion Control

We also need to consider the exact variant of TCP we are using. Different TCP variants use their own algorithms to determine at what rate to send data. These algorithms attempt to determine how congested a network is and increase or decrease their rate accordingly. If performing multiple tests simultaneously across the same link, we might also want to consider how fair the TCP variant we are using is to other streams. Some variants tend to dominate a connection and squash competing traffic. For most of our measurements we use two common and well performing congestion control algorithms: CUBIC and BBR. We also take some measurements using HTCP and BBRv2 Alpha. To understand the performance differences between these algorithms, it is essential that we know a little about the mechanisms each one uses to control congestion. Here, we will briefly describe each of the congestion control algorithms that we use in this thesis.

3.1.3.1. HTCP

HTCP uses time since the last loss to set its congestion window. The longer since the last loss or congestion event, the faster the window increases in size. It also sets the decrease factor by a function of RTTs. This is to estimate the queue size of current flow in the network path. The congestion window size increase for HTCP is calculated as follows:

$$W(t+1) = W(t) + \frac{2(1-\beta)f_\alpha(\Delta)}{cwnd},\ \beta = \frac{RTT_{min}}{RTT_{max}} \tag{3.2}$$

$$f_\alpha(\Delta) = 1 + 10(\Delta - \Delta_L) + 0.25(\Delta_i - \Delta_L)^2 \tag{3.3}$$

Where Δ_i represents time since the last window decrease and Δ_L is a threshold between lowspeed and highspeed modes [86]. This function is quadratic and leads to quick recovery times but does not handle large buffers well.

When congestion events (packet loss) occur, it decreases the congestion window proportional to RTT, giving it better fairness between flows with different RTTs.

3.1.3.2. TCP CUBIC

CUBIC also uses time since the last loss event and uses a cubic function to set the congestion window size to quickly recover

$$W(t) = C\left(t - \sqrt[3]{\frac{W_{max} \cdot \beta}{C}}\right)^3 + W_{max} \tag{3.4}$$

Where C is a cubic parameter, W_{max} is the previous window size before it was reduced, and t is the time since the window was reduced [87].

This leads to CUBIC increasing the window quickly at first, slowing down when it approaches previous max window size. If the window size increases past the previous maximum with no congestion event, the window size increases progressively faster.

3.1.3.3. BBR

BBR moves away from the loss based model and attempts to pace packets based on an estimate of the bandwidth delay product (bdp), where $bdp = BtlBw \cdot RTprop$. Here, $BtlBw$ is the estimated bottleneck bandwidth of the network and RTprop is the estimated

physical latency of the network, not including queuing delays. This is supposed to reduce the burstiness and limit queuing delay in large buffers [88].

BtlBw is estimated as the maximum delivery rate over a period of time, which is usually around 6-10 RTTs. $BtlBw \approx max(deliveryRate)$, where $deliveryRate = \Delta_{delivered}/\Delta_{time}$ and RTprop is estimated as the $min(RTT)$ over a window of several seconds [89].

It then paces packets with a maximum of $2 * bdp$ inflight at once instead of using a congestion window. Unlike HTCP and TCP CUBIC, BBR does not use loss as a congestion signal or react with a multiplicative decrease.

3.1.3.4. BBRv2 Alpha

BBRv2 is a refinement of BBR that attempts to correct some of the unintended issues of its predecessor, such as a very high rate of retransmissions under some circumstances [40]. The underlying theory and algorithm remain very much the same with a few additions. BBRv2 adds a loss threshold. If losses exceed a certain amount, the pacing of packets will be reduced. It also adds the ability to react to explicit congestion notifications (ECN), but this requires devices on the network to have this ability implemented and enabled. Yang et al. [**?**] found BBRv2 to be less aggressive than the original in networks with shallow buffers with somewhat lower throughput in these cases and more fair when sharing the network with loss-based congestion control algorithms. They also found that both BBR and BBRv2 perform worse in networks with high jitter.

3.2. Executing Cloud Benchmarks

When choosing a cloud provider, users are often faced with the task of figuring out which one best suits their needs. Beyond looking at the advertised metrics, many users will want to test these claims for themselves or see if a provider can handle the demands of

their specific use case. This brings about the challenge of benchmarking the performance of different cloud providers, configuring environments, running tests, achieving consistent results, and sifting through the gathered data. Setting up these environments and navigating the APIs and portals of multiple different cloud providers can escalate this challenge and takes time and skill. Despite the sometimes difficult nature of this, benchmarking is a necessary endeavor. This section demonstrates how to run a variety of network benchmarks on the largest public cloud providers using PerfKit Benchmarker (PKB). We begin with an overview of the PKB architecture and how to get started running tests, then describe specific test configurations to cover a variety of deployment scenarios. These configurations can be used to immediately compare the performance of different use cases, or run on a schedule to track network performance over time.

3.3. PerfKit Benchmarker

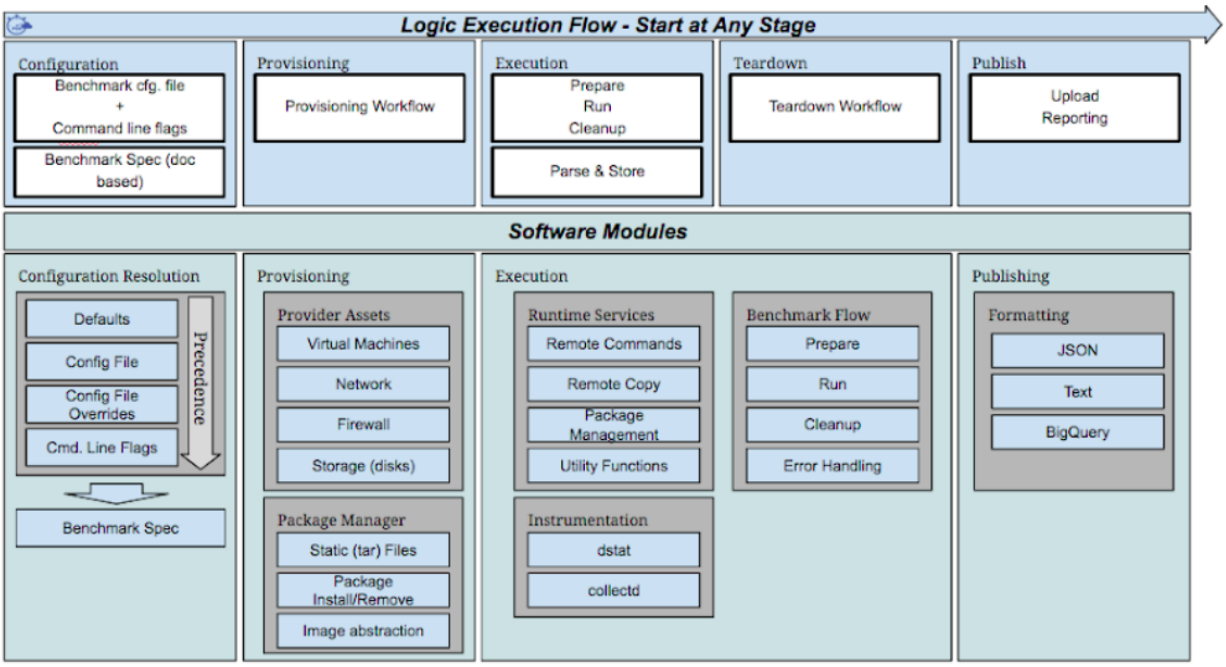


Figure 3.1: PerfKit Benchmarker Architecture Diagram

Specifically for running cloud based benchmarks, we use PerfKit Benchmarker (PKB). PerfKit Benchmarker is an open source tool created at Google that allows users to easily run

benchmarks on various cloud providers without having to manually set up the infrastructure required for those benchmarks. PerfKit Benchmarker follows a 5 step process, consisting of Configuration, Provisioning, Execution, Teardown, and Publish to automate each benchmark run. The Configuration phase processes command line flags, configuration files, and benchmark defaults to establish the final specification used for the run. The Provisioning phase creates the networks, subnets, firewalls, and firewall rules, as well as virtual machines, drives, and other cloud resources required to run the test. Benchmark binaries and dependencies like datasets are also loaded in this phase. The Execution phase is responsible for running the benchmarks themselves, and Teardown releases any resources created during the Provision phase. The Publishing phase packages the test results into a format suitable for further analysis such as loading into a reporting system. The metadata returned from the Publishing phase can include verbose details about the actual infrastructure used during the test and timing information for each phase of the run along with the metrics returned from the benchmark itself, providing the level of detail needed to understand the benchmark results in context.

3.3.1. PerfKit Benchmarker Basic Example

Once PKB has been downloaded and its dependencies have been installed, running a single benchmark with PerfKit Benchmarker is simple. We give it the benchmark we want to run and where we want to run it. For example, here is a ping benchmark between two VMs that will be located in zone us-east1-b on Google Cloud:

```
./pkb.py --benchmarks=ping --zone=us-east1-b --cloud=GCP
```

If the zone or cloud is not given, a default value will be used. We can also specify the machine type with the **--machine_type** flag. If this is not set, a default single CPU VM will be used.

3.3.2. PerfKit Benchmarker Example with config file

For more complicated benchmark setups, users define configurations using files in the YAML format, as shown in the following example. At the top of the config file is the benchmark that is being run. Next, we give it the name of a flag matrix to use, in this case we'll call it **fmatrix**. Then we define a filter to apply to the flag matrix and define the flag matrix itself. PKB works by taking the lists defined for each flag in the matrix (in our case this is zones, extra_zones, and machine_type) and finding every combination of those flags. It will then run the benchmark once for each combination of flags defined under fmatrix, as long as it evaluates to true with the flag matrix filters. The flags defined under **flags** at the bottom will be used for all benchmarks runs.

```
netperf:
  flag_matrix: fmatrix
  flag_matrix_filters:
    fmatrix: "zones != extra_zones"
  flag_matrix_defs:
    fmatrix:
      zones: [us-west1-a, us-west1-b,us-west1-c]
      extra_zones: [us-west1-a, us-west1-b,us-west1-c]
  flags:
    cloud: GCP
    netperf_histogram_buckets: 1000
    netperf_benchmarks: TCP_RR,TCP_STREAM,UDP_RR,UDP_STREAM
    netperf_test_length: 30
    netperf_num_streams: 1,4,32
    machine_type: n1-standard-16
    netperf_tcp_stream_send_size_in_bytes: 131072
```

This config file can be run with the command:

```
./pkb.py --benchmarks=netperf --benchmark_config_file=interzone_us_west1.yaml
```

Using this config file will run netperf TCP_RR, TCP_STREAM, UDP_RR and UDP_STREAM between pairs of n1-standard-16 instances in the us-west1-a, us-west1-b and us-west1-c zones. Because of the flag matrix filter, it will exclude tests from the same zone. Each test will be of 30 seconds duration and will be repeated for 1, 4, and 32 parallel streams. So from one config file and command line, we will get 72 benchmarks run (6 zone combinations * 4 netperf benchmarks * 3 different stream counts).

In the following subsections, we will see several more examples of how to run specific tests with PKB. Generally, they all use this same format; the structure and parameters of the benchmark are defined in a config file and a relatively simple command is used to start the benchmark with the specified config file.

3.3.3. PerfKit Benchmarker Configurations

All of the benchmarks that are presented here are simple and easy to reproduce should anyone want to run their own tests. In this section we will discuss the configurations for various test runs.

There are several general types of network benchmarks users may want to run, including: same zone (intra-zone), cross zone (inter-zone), and cross region (inter-region) tests. Intra-zone tests are between VMs within the same zone, which usually means that they are situated in the same datacenter. Inter-zone tests run between VMs in different zones within the same cloud region and Inter-region tests run between VMs in separate cloud regions. These kinds of groupings are necessary as network performance can vary dramatically across these three scales.

Additionally, benchmarks can be run to test network performance across VPN connections, on different levels of network performance tiers, using different server operating systems, and on Kubernetes clusters.

In this subsection, we cover the basic flags and configurations that are most commonly used for network tests. These benchmarks are fairly standard and used to gather metrics on latency, throughput, and packets per second. The tools being used by PerfKit Benchmarker here are the same ones we explored in Section 2.1.1.

3.3.3.1. Latency ping and TCP_RR

Ping is a commonly used utility for measuring latency between two machines and uses ICMP. The flag we should know for running a ping benchmark is **--ip_addresses= INTERNAL/EXTERNAL/BOTH**. Just as the name implies, this will tell PKB to get latency results using either internal IP addresses, external IP addresses, or both.

```
./pkb.py --benchmarks=ping --ip_addresses=BOTH \
--zone=us-central1-a --zone=us-west1-b --cloud=GCP
```

Ping, with its default once-a-second measurement is quite sufficient for inter-region latency. If we wish to measure intra-region latency (either intra-zone or inter-zone) a netperf TCP_RR test will show results that are more representative of application-level performance. We will explore this topic further in Section 3.1.

```
./pkb.py --benchmarks=netperf --netperf_histogram_buckets=1000 \
--netperf_benchmarks=TCP_RR --netperf_test_length=60 \
--zone=us-west1-b --cloud=GCP
```

3.3.3.2. Throughput with iPerf and Netperf

Both iPerf and Netperf can be used to gather throughput data about both TCP and UDP connections with various numbers of parallel streams, so that we can test single stream throughput performance as well as aggregate. The relevant flags for iPerf are as follows:

```
--iperf_runtime_in_seconds=60
--iperf_sending_thread_count=<num_threads>
```

The first sets the length of time the throughput tests are run (default: 60s) and the second flag sets how many threads iPerf will use to send traffic (default: 1).

```
./pkb.py --benchmarks=iperf --iperf_runtime_in_seconds=120 \
--iperf_sending_thread_count=32 --zone=us-central1-a --cloud=GCP
```

To perform UDP tests or a request/response test in PKB, we should use netperf. We can also set the number of streams, the test length in seconds, which netperf benchmarks are being run, and how many buckets are in the optional histogram.

```
./pkb.py --benchmarks=netperf \
--netperf_histogram_buckets=1000 \
--netperf_benchmarks=TCP_STREAM,UDP_STREAM \
--netperf_test_length=30 \
--netperf_num_streams=4 \
--zone=us-central1-a --cloud=GCP
```

For any of the example benchmark configurations in sections 3.2 and after, we can use iperf instead of ping, ping instead of netperf, etc. depending on what type of metrics we would like to gather.

3.3.3.3. Packets per Second Packets per second tests are performed using a script that runs multiple instances of netperf UDP request/response (UDP_RR) using small message sizes to achieve the maximum possible packets per second the VM can achieve in the configured situation. In PerfKit Benchmarker, it is called netperf_aggregate and uses 1 client machine and multiple server machines to test packets per second performance, as can be seen in the configuration file:

```
netperf_aggregate:
  vm_groups:
    servers:
      vm_spec:
        GCP:
          machine_type: n1-standard-4
          zone: us-east4-b
      vm_count: 2
    client:
      vm_spec:
        GCP:
          machine_type: n1-standard-4
          zone: us-east4-c
```

This config file can be run with the following command:

```
./pkb.py --benchmarks=netperf_aggregate \
--benchmark_config_file=/path/to/config.yaml
```

3.3.3.4. On-Premise to Cloud Benchmarks

On-premise to cloud benchmarks are highly specific to the user's location, so unlike most cloud to cloud benchmarks, we can't simply look up results on a table online. PerfKit Benchmarker makes it simple to setup our own benchmarking for our on-premise situation. There are two ways to perform On-Prem to Cloud Benchmarks within the paradigm of PerfKit Benchmarker. The first is to use a Static, On-Prem System (either VM or bare-metal). This requires us to set up said on-prem system and can ssh to it. Then in a config file, we can specify that machine be the static VM we have set up, and the other will be a VM that will be created on the cloud provider of our choice. A config file to run a netperf test between a sample static VM and a n1-standard-2 machine in GCP zone us-central1-a would look the following:

```
netperf:
  vm_groups:
    vm_1:
      static_vms:
        ip_address: 192.168.0.1
        ssh_private_key: <ssh_key>
        user_name: <username>
        zone: local
    vm_2:
      vm_spec:
        GCP:
          machine_type: n1-standard-2
          zone: us-central1-a
```

A potentially quicker option is to use Docker. If we have Docker installed, we can run tests between a Docker container running locally and a VM in the cloud. For this, the config file to use would look something like this:

```
netperf:
  vm_groups:
    vm_1:
      cloud: Docker
    vm_2:
      vm_spec:
      GCP:
        machine_type: n1-standard-2
        zone: us-central1-a
```

In the config file, specify two vm_groups: vm_1 and vm_2. In vm_1, tell it to use Docker as the cloud. In vm_2, use vm_spec to set the machine_type and zone manually, as shown in the example. Doing this will create a new Docker image if we have not used the Docker provider previously and a new Docker container on our local machine (wherever we execute PKB from) that will function as a VM for the benchmark. The command to run the benchmark from either of the preceding config files would be

```
./pkb.py --benchmarks=netperf --benchmark_config_file=/path/to/config.yaml
```

3.3.3.5. Cross-cloud Benchmarks

If we use multiple cloud providers, it may be of interest to run cross cloud benchmarks. With PKB, this can be achieved simply with a config file similar to the one we used for the on prem to cloud with Docker benchmark.

```
netperf:
  vm_groups:
    vm_1:
      cloud: AWS
      vm_spec:
        AWS:
          machine_type: m4.4xlarge
          zone: us-east-1a
    vm_2:
      cloud: GCP
      vm_spec:
        GCP:
          machine_type: n1-standard-16
          zone: us-central1-a
```

This will create one VM on AWS and another on GCP with the specified machine types in the specified zones and run netperf between them. The command to run the benchmark would be:

```
./pkb.py --benchmarks=netperf --benchmark_config_file=/path/to/config.yaml
```

3.3.3.6. VPN Benchmarks

Running benchmarks across an IPSec VPN is possible using the PKB VPN service. Base requirements for IPSec VPNs across the Internet:

1. Public IP address on both ends of the tunnel.

2. Unique subnet ranges behind each VPN GW. CIDRs can't overlap unless using multiple tunnels.

3. Pre-shared key

By default, GCP and some other providers in PKB run benchmarks from within a single VPC and subnet range. To meet the requirement for mutually exclusive subnet ranges, we can distinguish using the CIDR vm_group property in our benchmark config file as follows:

```
iperf:
  description: Run iperf on custom cidr
  vm_groups:
    vm_1:
      cloud: GCP
      cidr: 10.0.1.0/24
      vm_spec:
        GCP:
          zone: us-west1-b
          machine_type: n1-standard-4
    vm_2:
      cloud: GCP
      cidr: 192.168.1.0/24
      vm_spec:
        GCP:
          zone: us-central1-c
          machine_type: n1-standard-4
```

Then to establish the VPN for a benchmark config we can add **–use_vpn** to the flags passed to PKB and include the desired parameters to the vpn_service section of the configuration:

```
ping:
  flags:
    use_vpn: True
    vpn_service_gateway_count: 1
    vpn_service:
    tunnel_count: 2
    ike_version: 2
    routing_type: static
  vm_groups:
    vm_1:
    cloud: GCP
    cidr: 10.0.1.0/24
    vm_spec:
      GCP:
        zone: us-west1-b
        machine_type: n1-standard-4
    vm_2:
    cloud: GCP
    cidr: 192.168.1.0/24
    vm_spec:
      GCP:
        zone: us-central1-c
        machine_type: n1-standard-4
```

3.3.3.7. Kubernetes Benchmarks

There are two ways to execute Kubernetes tests on a cloud provider. The first is to create a Kubernetes cluster in the cloud provider and provide its config to PKB via the **–kubeconfig=⟨/path/to/.kube/config⟩** flag. Using this method, PKB handles the setup and teardown of the Kubernetes pods, in the cluster we have setup manually. This will work for most benchmarks that we want to run on a cluster. The second method involves using a config file that looks like the following with the benchmark container_netperf. Using this benchmark will set up a Kubernetes cluster for us and deploy pods that use a specialized netperf container image. In the config file, we have to specify the specs of both our containers that will be deployed and the cluster itself.

```
container_netperf:
  container_specs:
    netperf:
      image: netperf
      cpus: 2
      memory: 4GiB
  container_registry: {}
  container_cluster:
    vm_count: 2
    vm_spec:
      GCP:
        machine_type: n1-standard-4
        zone: us-east1-b
```

The command to run this benchmark will be:

```
./pkb.py --benchmarks=container_netperf \
--benchmark_config_file=</path/to/config.yaml>
```

3.3.3.8. Intra-zone Benchmarks

To run an intra-zone benchmark (two VMs in the same zone), we can simply specify the zone we want both VMs to be in and any other flags we want to specify. The following example runs an intra-zone netperf TCP_RR benchmark in GCP zone us-central1-a with n1-standard-4 machines.

```
./pkb.py --benchmarks=netperf --cloud=GCP --zone=us-central1-a \
--machine_type=n1-standard-4 --netperf_benchmarks=TCP_RR
```

3.3.3.9. Inter-zone Benchmarks

Inter-zone tests, like most other tests can be executed in one of two ways. It can be done entirely from the command line using the –**zone** flag, as follows:

```
./pkb.py --benchmarks=iperf --cloud=GCP --zone=us-east4-b \
--zone=us-east4-c --machine_type=n1-standard-4
```

The same Inter-zone benchmark can also be set up using a config file:

```
iperf:
  vm_groups:
    vm_1:
      cloud: GCP
      vm_spec:
        GCP:
          machine_type: n1-standard-4
          zone: us-east4-b
    vm_2:
      cloud: GCP
      vm_spec:
        GCP:
          machine_type: n1-standard-4
          zone: us-east4-c
```

This config file can be run using the command:

```
./pkb.py --benchmarks=iperf --benchmark_config_file=/path/to/config.yaml
```

3.3.3.10. Inter-Region Benchmarks

Inter-Region benchmarks (between VMs located in separate geographic regions), can likewise be run using command line flags or with a config file.

```
./pkb.py --benchmarks=iperf --cloud=GCP --zone=us-central1-b \
--zone=us-east4-c --machine_type=n1-standard-4
```

The same Inter-Region benchmark can also be set up using the following config file:

```
iperf:
  vm_groups:
  vm_1:
    cloud: GCP
    vm_spec:
      GCP:
        machine_type: n1-standard-4
        zone: us-central1-b
  vm_2:
    cloud: GCP
    vm_spec:
      GCP:
        machine_type: n1-standard-4
        zone: us-east4-c
```

And this config file can be run with the following command:

```
./pkb.py --benchmarks=iperf --benchmark_config_file=/path/to/config.yaml
```

3.3.4. Inter-Region Latency Example and Results

As an illustrative example, we present the actual results of our Google Cloud all-region to all-region round trip latency tests, as shown in Fig. 3.2. This chart shows the average round trip latency between regions from benchmarks run over the course of a month. The benchmarks were all executed on n1-standard-2 machine types with internal IP addresses.

The statistics are all collected using PerfKit Benchmarker to run ping benchmarks between VMs in each pair of regions.

To reproduce this chart, we can run the following pkb command with the following config file. If we want to run a smaller subset of regions, just remove the regions we don't want included from the zones and extra_zones lists.

```
ping:
  flag_matrix: inter_region
  flag_matrix_filters:
    inter_region: "zones < extra_zones"
  flag_matrix_defs:
    inter_region:
      zones: [<list_of_all_regions>]
      extra_zones: [<list_of_all_regions>]
  flags:
    cloud: GCP
    machine_type: n1-standard-2
    ip_addresses: INTERNAL
```

We can also add the **–run_processes=⟨# of processes⟩** to tell it to run multiple benchmarks in parallel, but this will still likely take awhile (>12 hours). If we run too many benchmarks in parallel, we may run into quota issues, such as regional CPU quotas and per project subnet quotas, which limits us to around 8 processes. If we exceed a quota while running PKB, it will tell us the exception that was thrown and the benchmark will fail. Additionally, we can use the **–gce_network_name=⟨network name⟩** flag to have each benchmark use a GCP VPC that we have already created, so that each benchmark does not

make their own, which adds up to a significant amount of time. This will also ensure that we don't run into subnet quota issues.

```
./pkb.py --benchmarks=ping --benchmark_config_file=/path/to/config.yaml
```

milliseconds	*receiving_region*																			
sending_region	asia-east1	asia-east2	asia-northeast1	asia-northeast2	asia-south1	asia-southeast1	australia-southeast1	europe-north1	europe-west1	europe-west2	europe-west3	europe-west4	europe-west6	northamerica-northeast1	southamerica-east1	us-central1	us-east1	us-east4	us-west1	us-west2
asia-east1		13	36	37	107	46	139	282	250	244	258	253	262	183	289	153	185	175	118	128
asia-east2	13		48	48	95	37	127	294	263	257	267	266	274	195	302	164	197	190	130	142
asia-northeast1	36	49		9	127	67	114	254	220	215	226	222	234	154	260	122	156	146	89	98
asia-northeast2	37	48	9		138	79	122	260	228	224	235	231	241	163	269	131	165	156	98	106
asia-south1	108	95	127	138		60	151	374	341	337	346	348	359	276	384	245	276	267	212	220
asia-southeast1	46	37	67	77	60		92	318	285	278	290	286	296	215	323	187	219	209	152	161
australia-southea	139	127	114	122	152	92		303	271	264	276	273	283	203	302	171	196	196	160	137
europe-north1	283	296	253	261	374	318	303		33	40	32	31	39	115	241	132	124	113	165	167
europe-west1	251	263	219	228	342	285	271	33		7	7	7	14	82	210	99	92	81	131	134
europe-west2	245	258	215	224	337	279	265	40	7		13	10	20	77	203	94	87	76	127	129
europe-west3	259	268	226	234	349	290	276	32	7	13		7	8	88	214	105	98	87	137	140
europe-west4	254	267	223	231	348	286	273	30	7	10	7		14	84	211	102	94	83	136	137
europe-west6	263	275	234	242	358	297	283	39	14	20	8	14		95	221	112	105	93	147	147
northamerica-nor	183	196	154	163	277	216	203	115	82	77	88	84	95		142	31	25	14	67	66
southamerica-ea	290	303	260	269	383	323	301	241	210	204	215	212	221	142		144	117	130	172	166
us-central1	154	164	122	131	245	188	172	131	99	94	105	103	111	31	143		35	25	34	35
us-east1	186	198	157	165	276	220	197	124	92	87	98	94	104	26	118	35		12	67	60
us-east4	176	190	146	156	267	209	196	113	81	76	86	84	93	14	129	25	12		58	60
us-west1	119	131	89	98	212	153	161	164	131	127	137	136	147	67	172	34	67	58		25
us-west2	128	142	98	106	219	161	137	166	133	129	140	137	146	66	166	36	60	60	26	

Figure 3.2: Inter-Region Latency results for Google Cloud. All numbers are in milliseconds

In the matrix shown in Fig. 3.2, The labels on the y-axis (left side) represent the sending region and the labels on the x-axis (across the top) represent the receiving region. So if we look at the intersection of asia-east2 on the y-axis and asia-east1 on the x-axis, this represents the average of results from ping benchmarks executed from a VM in asia-east2 to a VM in asia-east1.

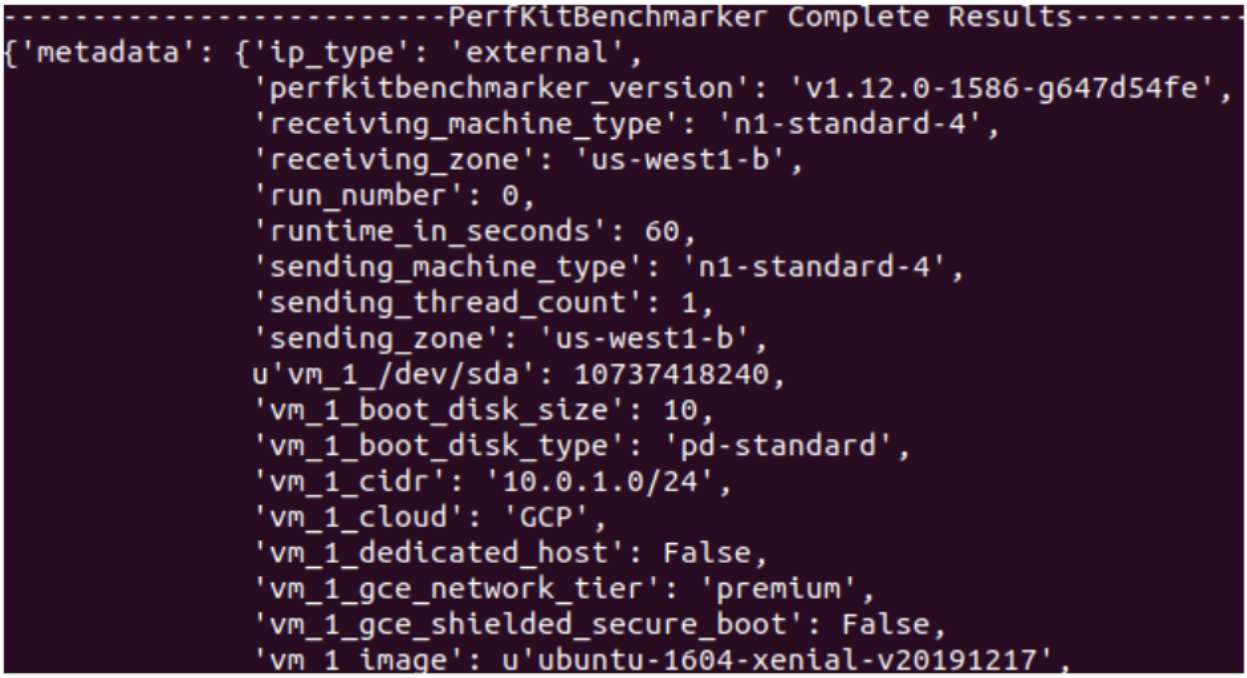


Figure 3.3: Example of JSON output from PerfKit Benchmarker

3.3.5. Viewing and Analyzing Results The report generated from a PKB run includes the results of the benchmark test along with a significant quantity of metadata about the test environment. The raw report is a JSON formatted dictionary of key:value pairs. The default location for this file is ⟨tmp_dir⟩/perfkitbenchmarker/runs/⟨run_uri⟩/perfkitbenchmarker_results.json. PKB includes a number of publishing targets as well, which can be specified on the command line when the test is launched to store the results in a backend like BigQuery or ElasticSearch automatically. It is then possible to query these runs from a dashboard provider to visualize the data.

3.3.5.1. Visualizing Results with BigQuery and Data Studio To use the BigQuery PKB publisher, first create a BigQuery table in our GCP project (the schema will be created when we first push a sample), and then include the table name and project name in the PKB run flags:

```
./pkb.py --benchmarks=iperf --benchmark_config_file=/path/to/config.yaml
--bigquery_table=<bq.table> --bq_project=<bq.project>
```

Table 3.1: SQL Schema for PerfKit Benchmarker results

FIELD NAME	TYPE	MODE	DESCRIPTION
unit	STRING	NULLABLE	Unit type of the test/metric. (sec, ms, Mbit/sec, etc)
labels	STRING	NULLABLE	Catch all field that stores any information about the benchmark that does in any other field. This will contain a variety of information depending on the specific benchmark and test setup
timestamp	FLOAT	NULLABLE	Timestamp of benchmark in epoch time
product_name	STRING	NULLABLE	Name of the testing tool (this will always be 'PerfkitBenchmarker')
test	STRING	NULLABLE	Name of the specific benchmark that is being run (iperf, netperf, ping, etc)
official	BOOLEAN	NULLABLE	This will always be false
metric	STRING	NULLABLE	The specific metric that the value and unit type is for. (Avg latency, TCP Throughput, etc). A test can have multiple metrics.
value	FLOAT	NULLABLE	The value of the specific test and metric
owner	STRING	NULLABLE	The user who executed PerfKitBenchmarker
run_uri	STRING	NULLABLE	A unique value assigned to each test run
sample_uri	STRING	NULLABLE	A unique value assigned to each metric in each test run

The schema for each sample published by a run is described in Table 3.1. Each run can (and usually does) produce multiple samples. In a network test like ping for example, the latency from zone_1 to zone_2 and the latency from zone_2 to zone_1 are recorded in separate samples. Likewise, there are separate samples created when using public and private networks, as well as samples that describe system metadata like lscpu and procmap. All of the samples for a single run share the same run_uri and can be joined on this field for grouping in queries.

Once the table is populated we can query run results directly for reporting. If we are capturing several test types or tests with different parameters in the same table it may be useful to create views for each test used in our reports. The following BigQuery Standard SQL query shows how we can capture specific key:value pairs nested in the labels field and how to work with the time format for use in reporting.

```
SELECT
 value,
 unit,
 metric,
 test,
 TIMESTAMP_MICROS(CAST(timestamp * 1000000 AS int64)) AS thedate,
 REGEXP_EXTRACT(labels, r"\|sending_zone:(.*?)\|") AS sending_zone,
 REGEXP_EXTRACT(labels, r"\|receiving_zone:(.*?)\|") AS receiving_zone,
FROM <PROJECT>.<dataset>.<table>
```

To create a visualization using Data Studio, we start by adding a connection to the BigQuery table we specified above. If using separate views, we can make each view its own data source.

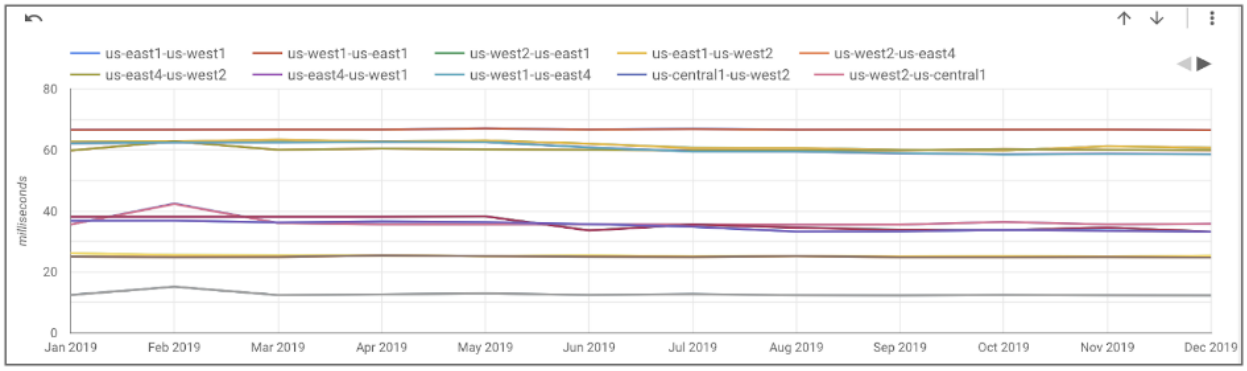


Figure 3.4: Example of JSON output from PerfKit Benchmarker

Once Data Studio can see the PKB results table, we can design our charts and visualizations accordingly using the full range of reporting tools available. The example report in Figure 3.4 shows inter-region ping latency results over time:

Chapter 4

Architecture and Execution Efficiency of Benchmarks

4.1. Introduction

In the previous chapter, we discussed how we run cloud benchmarks and how to optimize our networks to be able to measure the maximum possible throughput. In this chapter we will discuss how we can run these benchmarks more efficiently. This includes making each individual benchmark more efficient, as well as running a large number of benchmarks efficiently.

First we will examine at how we can reduce the time and cost of individual benchmarks. Here we focus on network throughput benchmarks, but we believe the techniques we use are more widely applicable. We want to be able to run network throughput tests but reduce the overall cost of these tests while maintaining accurate results

We will then be examining how to schedule and run a large number of network benchmarks in a cloud environment. The motivation for this comes from a practical dilemma we faced when running benchmarks. To support research in network analytics both for this thesis and an outside research project, we are running a large number of benchmarks on a daily basis on multiple cloud computing platforms. Scheduling these benchmarks in a naive manner with bash scripts and PerfKit Benchmarker quickly led to a situation where there were more benchmarks we needed to run daily than we had time or capacity for.

4.2. Efficiency in Benchmark Execution

It is essential to the operation of network focused applications to be able to estimate the achievable throughput across a link. This could be for bulk data transfer operations such as with large scientific datasets, data streaming, or applications that need to sustain consistent throughput. Similarly, if we are operating on a cloud provider, they may provide an upper bandwidth limit or SLA, but it is often uncertain how much throughput can be achieved at any point in time. This leaves it in the hands of the user to determine their network throughput. To do this, we need to run throughput benchmarks to gain an accurate understanding of the performance capabilities of our network. These tests may need to be performed on a regular basis to monitor changing network conditions.

Unfortunately, throughput benchmarks are often costly to perform. If we are running these tests on a cloud provider's network, we can accumulate large fees for data transfer or egress. Other networks such as mobile or home networks may be subject to data caps and a throughput test may use a significant amount of the available data. Additionally, we need to think about the quality of the measurement. Generally, the longer we run a test, the more confident we can be that the reported throughput value is accurate and representative of what we can expect from the network. This is the primary motivation for the work in this section. To reduce the cost of these tests, we need to reduce the total amount of bandwidth they consume or network traffic they produce. Reducing traffic is also beneficial for the rest of the network as we do not want to cause too much disruption to actual traffic going across the network, which a bulk throughput test has the ability to do, especially at scale in a cloud or multi-user environment with a shared network.

To reduce the total bandwidth consumed, we can reduce the duration of tests. This has the possibility of also reducing the granularity and accuracy of our data. So, if we reduce the testing in this manner we must do so in a way that affects the quality of our data as little as possible. To this end, we have come up with a method that can both decrease our data

consumed and maintain statistically good test results. On the other hand, if we run tests that are too short we may obtain results that are not accurate. This method also overcomes the challenge of determining how long a throughput test should last for good results, as our solution does this automatically. Our method takes iPerf2, an existing and widely used throughput testing tool, and adds user specified confidence intervals. Instead of setting a static test length, this allows the user to run a test until the gathered throughput samples meet their specified level of confidence and interval width. If the given width is reached before the maximum test time, the test will stop early. The user can set the maximum test duration to the same amount of time they would normally run a throughput test for and then in the worst case scenario, our auto-stopping feature will result in the exact same test run as when not using it. If the test ends early, this can help lower testing costs and durations.

A confidence interval is often a more accurate way to define throughput on a link, especially if that link is on a public network with large amounts of cross traffic and congestion. Given the ever fluctuating state of the network, it is difficult with any sense of accuracy to assign a single authoritative number for throughput on a link. We can say we achieved x amount of throughput for a specific test, but that defines just that link at that single point in time. To discuss a link more broadly, we must include some sort of error term, e.

4.2.1. Statistical Confidence

A confidence interval is a range of values that are estimates for an unknown mean. If we took a large number of samples over time and calculated the confidence intervals for each, the confidence level would be the percentage of these confidence intervals that actually contain the true mean. So for a confidence level of 95%, 95% of the confidence intervals should contain the true mean [90].

To construct a confidence interval, we need the confidence level that we are using (i.e. 95%), a sample mean that estimates the population mean, and error bounds for the population mean. From the confidence level, we can use the standard normal distribution to calculate a Z-score. Then we can we can calculate the error bound as:

$$\epsilon = Z\frac{\sigma}{\sqrt{n}}, \tag{4.1}$$

where Z is the Z-score, $\overline{x}$ is the sample mean, σ is the population standard deviation, and n is number of samples. Of course, in many situations, σ is not known, so we must estimate σ using the standard deviation for the sample, which is denoted as s. When this is the case, we can express the error bound as:

$$\epsilon = t_{n-1}\frac{s}{\sqrt{n}}, \tag{4.2}$$

Here, the Z term has been replaced by t_{n-1}. This is because using s, an estimation of σ, often leads to a less accurate confidence interval. To correct for this, we use a t-distribution instead of the normal distribution. The exact t-distribution to be used depends upon the degrees of freedom for the sample, which is defined as $(n-1)$. As we add more samples to the calculation, the T-distribution we use gets close to a normal distribution and t_{n-1} gets closer to Z for the confidence level we require. If we have over 30 samples, the T-distribution is considered to be very close to the normal distribution.

Then the bounds for this confidence interval can be expressed as the sample mean plus and minus the error bound

$$CI\_bounds = \overline{x} \pm \epsilon \tag{4.3}$$

When we discuss this interval, we can say that, with 95% confidence, we estimate the true mean is in the range $(\overline{x} - \epsilon, \overline{x} + \epsilon)$.

In Fig. 4.1, we show measurements from one of our test paths and how the confidence interval is applied to it. The confidence width is the distance between the mean and the upper and lower bounds of the interval.

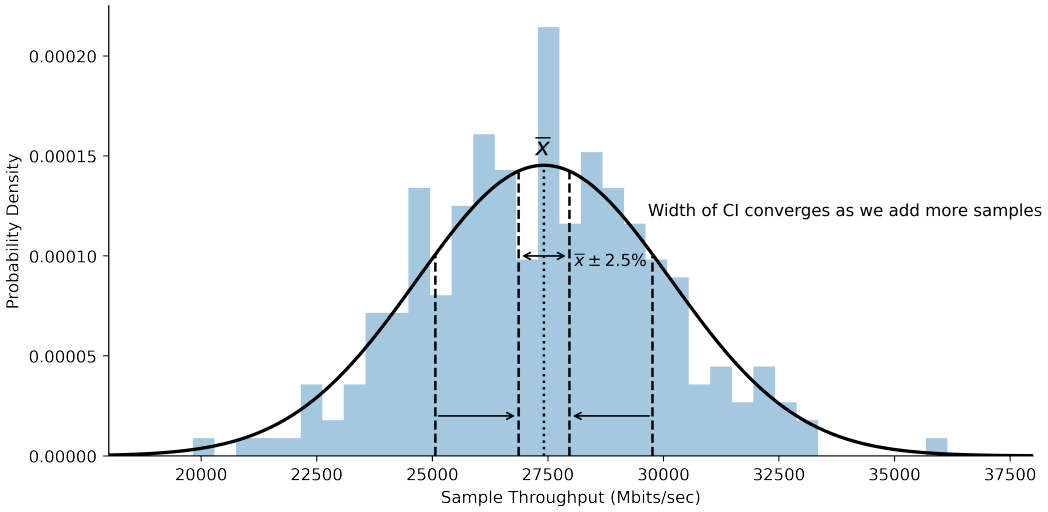


Figure 4.1: Confidence Interval for 240 samples in region pair C. As we add more samples, the width of the CI becomes smaller until it reaches $\overline{x} \pm 2.5\%$.

If we set a given target width for the confidence interval, either as a numeric or a percentage difference from the mean, we can continue to gather samples until we meet the target width. The width of the interval is based on the number of samples collected and the variability in the samples. If the target width is too low and the variability of the sample too high, the target width may not be achieved before the test ends.

To apply these concepts to our network tests, we assume that for a specific network link with specific system and network configurations, there exists a mean throughput value. If we were to run a throughput test for an arbitrarily long time (a large number of measurements) we could find this mean throughput value, which is equivalent to the population mean. When we run a throughput test, we test for a much shorter period of time (a smaller number of

measurements) and are able to find a sample mean. If we run a test for 120 seconds and collect the average throughput over every 0.5 second interval, we end up with a sample of 240 measurements which we can use to calculate the confidence interval. If we recalculate the confidence interval after each 0.5 second measurement period, we can stop the test after the confidence interval has reached our target width.

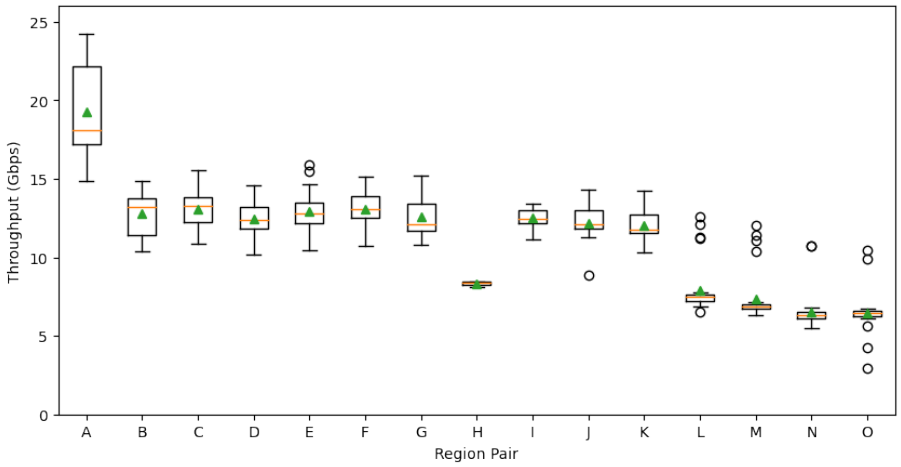
4.2.2. Data Observations

Table 4.1: Region pairs

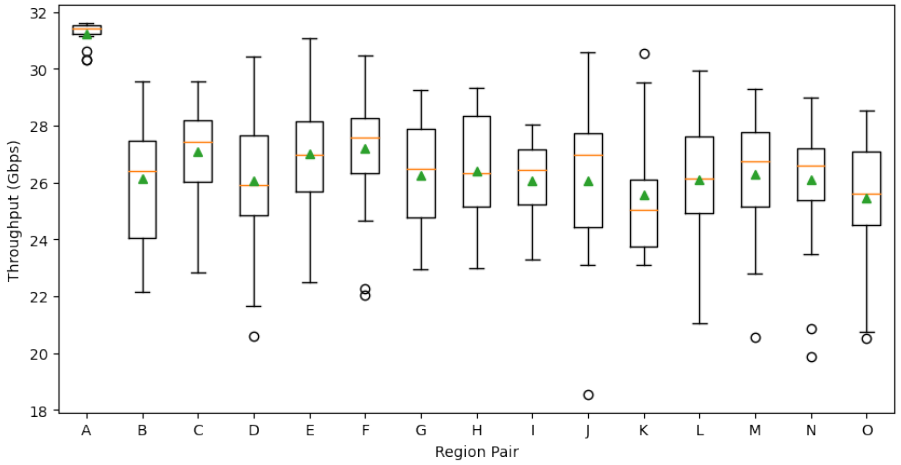
Region 1	Region 2	Key	RTT (ms)
us-east1	us-east1	A	0.29
us-east1	us-east4	B	12.82
us-central1	us-east4	C	26.28
us-central1	us-east1	D	32.616
us-west1	us-central1	E	32.72
us-west1	us-east4	F	57.61
us-west1	us-east1	G	64.14
asia-southeast1	asia-northeast2	H	82.28
northamerica-northeast1	europe-west1	I	82.56
us-central1	europe-west4	J	102.60
europe-north1	us-east4	K	103.17
asia-east1	us-west1	L	119.44
us-central1	asia-east1	M	150.66
us-east4	asia-east1	N	178.47
asia-east1	us-east1	O	190.68

To examine the effect of the test length on the measured throughput value, we have run a large number of network throughput tests using iPerf across several different region pairs in Google Cloud that represent a range of round trip times (RTTs) from <1ms to ∼190ms. These tests took place between pairs of n1-standard-16 virtual machines, each of which has 16 vCPU cores, 60 GB of Memory and are running Ubuntu 20.04 LTS. Each test had a total

time of 120 seconds and data was collected for every 0.5 second interval in the test. This gives us 240 total measurements for each test, or sample. This test duration was chosen because it gives us a relatively small confidence interval even with our region-pairs with the highest latency and variance. In Table 4.1, we show each region pair along with a short key that we will use to represent the pair and the average RTT between each pair, as measured by multiple ping tests.



(a) Region pair vs 1 flow throughput



(b) Region pair vs 32 flow throughput

Figure 4.2: Throughput by Region Pair

In Fig. 4.2, we show the the throughput from each sample for each of the region pairs under test. The names of the region pairs are abbreviated for space. Fig. 4.2a shows throughput for 1 flow and we can see that for many of the region-pairs, there are fairly tight

bounds on our throughput samples with few outliers. The exception is us-e1 to us-e4, which are quite close together geographically and have correspondingly low latency. Here there is a positive skew to the distribution of our samples.

Fig. 4.2b shows aggregate throughput for 32 parallel flows. The achieved throughput from these samples is uniformly higher than the single flow samples, but there is also more variance.

In Fig. 4.3, we show the average reported aggregate throughput over a 120 second iPerf test with 32 parallel flows. This chart shows the average for each 0.5 second interval of 10 tests between the us-east1 and us-west1 Google Cloud regions. The chart that we show is very similar to ones from other region pairs. There are a few noticeable features to discuss. The first is the large spike from less than 10 Gbps to over 60 Gbps and then back down to around 27 Gbps. This is a common feature we see on these throughput charts. Since for these measurements we are using BBR [88], this spike could show the initial startup and then drain phases of BBR, followed by much steadier throughput thereafter. With a single flow there is not nearly this much over estimation, but because this is the aggregate throughput of 32 parallel flows, these add up to a large spike in throughput that is immediately constrained.

The startup phase of most TCP variants do not reflect how the sustained throughput looks after that phase and usually only lasts a few seconds. If we want to measure the average throughput for a long lasting TCP flow and we are aware that this startup phase is non-representative of a long lasting flow, then it is likely that our result will converge much more quickly if we simply discount the first few interval measurements from our results. Some throughput testing tools have this capability such as iPerf3.

The other noticeable feature is a small drop in throughput about every 10 seconds. This is fairly consistent across tests. We believe this is also an artifact of multiple BBR flows,

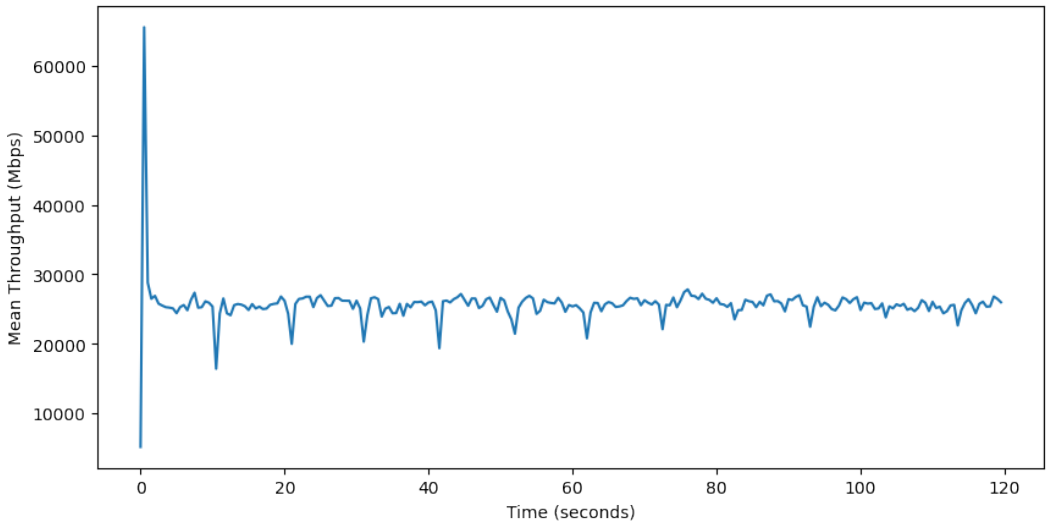


Figure 4.3: Average iPerf reported 32 flow throughput from us-east1 to us-west1 over 120 seconds with 0.5 second intervals

all attempting to go through their ProbeRTT phase simultaneously [38]. We do not see the same dips when using other TCP variants such as CUBIC.

4.2.3. Confidence Interval Calculations

For each of these tests, we looked at all 240 half second samples and calculated how many samples it would take to achieve a 95% confidence interval with a width of ±2.5%. We also compared average throughput value from the full 240 sample test, which we will call $\tilde{x}$, to the range of the confidence interval, CI, if and when it achieves the target width to find the percentage of samples where $\tilde{x}$ falls outside the range of CI, which we will call ϕ. Put more simply, ϕ is the percentage of tests where we achieved the target width for the interval before then end of the test, but the average throughput for the entire sample fell outside of the interval.

So for n total tests, ϕ is calculated as:

$$\phi = \frac{\sum_{k=0}^{n} i = \begin{cases} 1, & \text{if } \tilde{x}_k > CI_{max} \text{ or } \tilde{x}_k < CI_{min} \\ 0, & \text{otherwise} \end{cases}}{n} \tag{4.4}$$

Table 4.2: Calculated Confidence Interval Statistics For 32 Thread Throughput Data

min samples	skip	avg n	WR	ϕ	%Δ
0	0	117.67	0.74	0.05	0.009
0	5	80.128	0.79	0.15	0.014
10	0	117.86	0.74	0.05	0.009
10	5	82.30	0.79	0.13	0.013
20	0	118.85	0.74	0.04	0.008
20	5	90.41	0.79	0.10	0.011

We show the results of these confidence interval calculations combined for all region-pairs under test in Table 4.2. For each row we calculate the confidence interval using a different minimum number of measurements and skipping either 0 or 5 measurements at the beginning of the test. We also show n, the average number of measurements needed to achieve the target width, the percentage of samples where the target width was reach by the end of the test which we will call WR, ϕ as calculated in (4.4), and the average percent difference between the calculated mean of the confidence interval when it reaches the target width and the average of the complete test with all 240 measurements which we will call %Δ.

When we skip the first 5 measurements, the average number of samples needed to meet the target confidence interval width is reduced by at least 23%. It also increases WR, meaning more samples met the target width before the end of the test. Skipping initial measurements also led to a higher ϕ values. This could simply be because $\tilde{x}$ includes the

skipped measurements which, depending on how far above or below that initial spike is from the rest of the measurements, can throw of the mean of the entire sample.

4.2.4. Experiment Setup

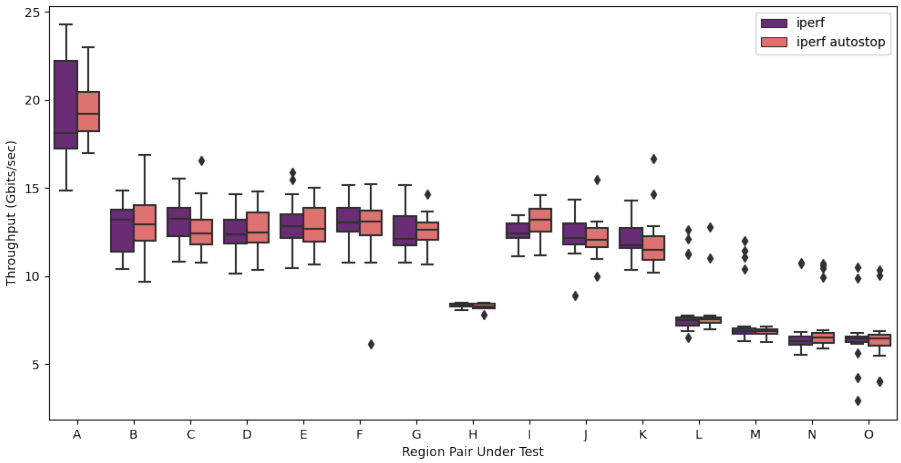
In order to ascertain the veracity of this method, we ran a series of tests to compare using iPerf2 to using iPerf2 with the addition of our auto-stopping feature. We conducted tests across a range of RTTs on Google Cloud. We picked several pairs of cloud regions with different RTTs ranging from 1ms to 190ms. The region pairs that we selected and their associated RTT are listed in Table 4.1

Between each pair of region we ran both iPerf2 and iPerf2 with auto-stop. iPerf2 was set to have a test time of 120 seconds and iPerf2 with autostop was set to have a maximum test time of 120 seconds, an interval size of 0.5 seconds, a minimum of 20 intervals, and to skip the first 5 intervals. The confidence level was set at 95% and the target width was ±2.5%. We tested each using both 1 flow and 32 parallel flows. The first is to find the throughput for a single TCP flow and the second is to find the maximum aggregate throughput for the link.

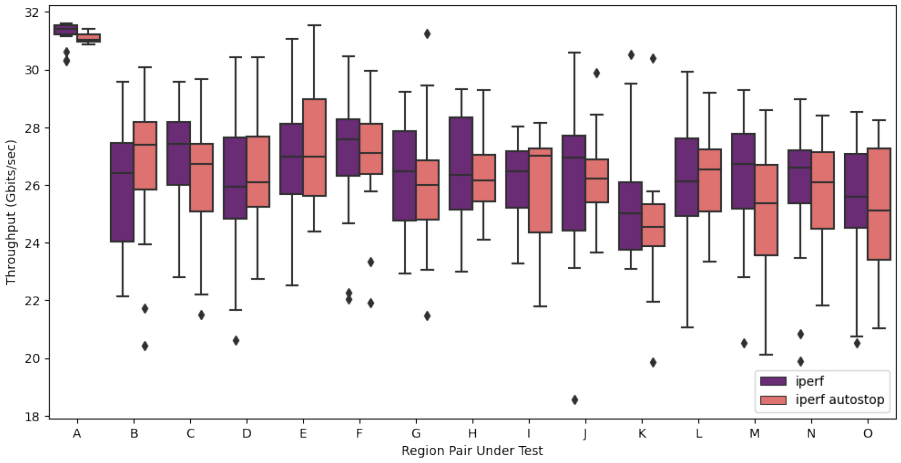
For all of these tests we used n1-standard-16 virtual machines running Ubuntu 20.04, which have 16 vCPU cores, 60 GB of memory, and a maximum of 32 Gbits/sec of network egress. We ran these tests once daily for 3 weeks at a different time each day for account for any temporal variance.

4.2.5. Results

In Fig. 4.4, we show the results of all of our throughput tests for both iPerf2 and iPerf2 with autostop. Fig. 4.4a shows throughput for 1 flow and Fig. 4.4b shows throughput for 32



(a) 1 Flow Throughput

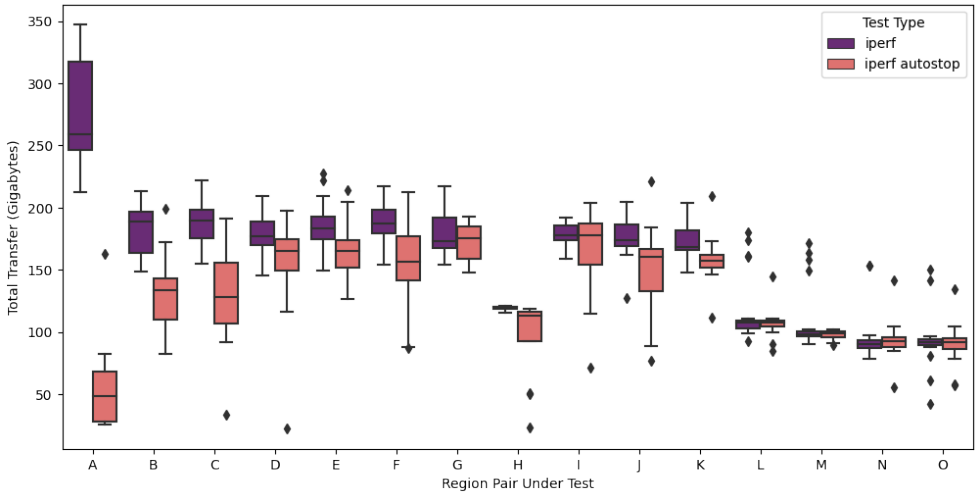


(b) 32 Flow Throughput

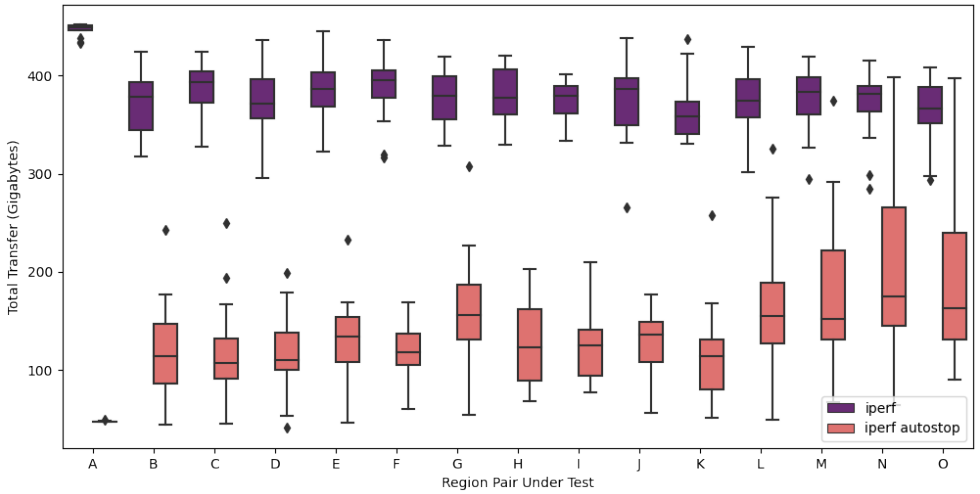
Figure 4.4: Throughput Results for iPerf both with and without autostop

parallel flows. The interquartile range reported by iPerf2 running for 120 seconds and iPerf2 with autostop are similar for each region pair.

In Fig. 4.5 we show the total amount of gigabytes that were transferred to achieve each test result. Here, in every case using the autostop feature results in a similar or lower amount of total gigabytes transferred. This result is especially clear with 32 parallel flows show in Fig. 4.5b. It might be obvious upon inspection that we could achieve similar throughput results for iPerf using a shorter test time for some of these region-pairs, but using the autostop



(a) 1 Flow Total Transfer



(b) 32 Flow Total Transfer

Figure 4.5: Total Transfer for iPerf both with and without autostop

feature removes the guesswork and also allows the test to run for the set maximum duration if needed to achieve a good result.

In Table 4.3, we list the percentage change between the median throughput from our measurements using iPerf and our measurements using our autostop feature (%Δ) for both 1 and 32 parallel flows. We also show the average number of measurements, n, needed for each sample to achieve the target width for the confidence interval. The regions and RTTs

that correspond to each Region-pair key are listed in Table 4.1. Here, they are listed in order of increasing RTT.

For 1 flow in 12 out of 15 region-pairs, the median throughput from samples with autostop was within 5 percent of the mean of those without autostop. For 32 flows, this figure is 14 out of 15 region-pairs. A large error here possibly means that we should increase the minimum number of measurements for each sample.

Looking at the average number of measurements needed for each sample, there is trend where the number of measurements needed is positively correlated with the RTT of the region pair. This makes sense, as generally higher RTT connections will have higher throughput variance, which can lead to the confidence interval For region-pair M with 1 flow, the average n was 235. This is the maximum length of the test (excluding the 5 initial skipped measurements), which means that the tests for M took the maximum amount of time for every samples. This may mean that we actually need a longer test here, as we are not ever achieving the target width for our confidence interval.

Using autostop also reduced the test time in most cases. For 1 flow, the average time it took to achieve the target confidence interval width ($\pm 2.5\%$) across all regions was 101.08 seconds. For 32 parallel flows, it took an average of 85.4 seconds to reach the target width. Both of these are a significant reduction from the 120 second maximum test length.

Of course, these bytes transferred and time reductions were not constant across all of the region-pairs that we tested. For example, our tests from the asia-east region to the us-central region took the full 120 seconds every time and never met the target width, having an average end confidence interval width of ($\pm 3.54\%$). This could mean that it is a high latency link with high throughput variance. We might need to set the maximum test length higher for this region-pair, or alternatively it might never reach the target width because of its variance. Once we know that, we can adjust our tests accordingly.

Table 4.3: Percentage Change Between iPerf Mean Throughput And iPerf With Autostop Mean Throughput

Region-pair Key	%Δ 1	%Δ 32	avg n 1	avg n 32
A	05.84	-01.17	43.08	20.08
B	-01.96	03.80	167.71	69.62
C	-06.21	-02.63	169.08	68.75
D	00.97	00.63	203.75	66.83
E	-00.93	00.03	210.58	75.21
F	00.31	-01.64	201.92	67.54
G	04.26	-01.80	227.00	98.75
H	-01.33	-00.70	189.58	76.00
I	06.26	02.08	207.71	75.00
J	-00.65	-02.69	198.75	76.17
K	-02.48	-01.92	217.79	73.57
L	00.19	01.53	226.46	95.54
M	00.56	-05.11	235.00	112.33
N	02.99	-01.94	222.15	124.12
O	-00.01	-01.86	231.46	127.04

In Fig. 4.6, we have picked a random 32 flow aggregate throughput sample that was captured using iPerf2 with autostopping. Superimposed over the 0.5 second throughput measurements are the confidence intervals. In red is the confidence interval calculated without skipping the first 5 measurements and in blue is the confidence interval calculated with the first 5 measurements skipped. Because of how different these initial measurements are to the later phase of sustained throughput, the confidence interval with skip converges to the target width more quickly.

4.3. Efficiency in Benchmarking Architecture

Not that we have examined how to make a single benchmark more efficient through the use of confidence interval calculations, we now need to discuss how to run a larger number of benchmarks efficiently. If we have some large number of benchmarks that need to be executed, in most situations we can not run them all simultaneously or we will run into

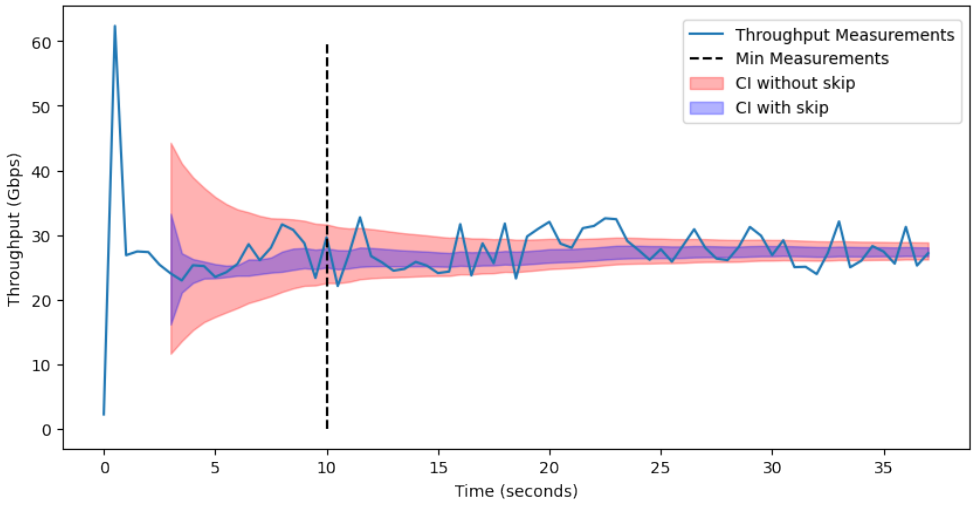


Figure 4.6: Throughput with calculated confidence interval

issues such as resource exhaustion or benchmarks interfering with each other. It may also be impractical to run the benchmarks in sequence, as we may have time constraints that this method would exceed. In this situation, there is a need to be able to dynamically schedule batches of benchmarks so that they stay within the resource constraints without interfering with each other, while executing in a time efficient manner. This is the situation we have when running benchmarks across multiple cloud providers.

Each of these cloud providers has different constraints and limits, usually on a regional basis (which, in the terminology of most cloud providers, means per each geographically distributed datacenter). These constraints include limits on IP addresses, virtual networks and subnets, virtual machines, CPU cores, storage, and bandwidth. Before the proposed solution here, we manually scheduled batches of benchmarks to run, where each batch would not exceed any resource limits. We needed to also ensure that all of the batches were able to run in the specified time window, which we set as 25 hours.[1] The manual approach of benchmark schedule turned out to be tedious, time consuming, and error prone. It also had to be updated whenever there was a change in resource limits or new benchmark requirements.

[1]Here the extra hour is so the benchmarks start running at a different time every day, which over the long term will hopefully average out any effect of running the same benchmark at the same time of day every day.

4.3.1. Batch Scheduling Optimization

In essence, our benchmarking problem can be thought of as a scheduling problem with special constraints. The solution to these types of problems are generally approximated by heuristics because they are considered to be NP-Hard. We will show it is NP-Hard in subsection 4.3.2. It would be extremely time consuming to calculate every possible combination of VM allocation and benchmark schedule, so a solution that gets close to optimal is considered good enough. One approximation method is dynamic programming, for example M. Alsaih et. al looks at dynamic scheduling in a cloud environment and, similar to our work, schedules jobs based on the jobs characteristics and the characteristics of the machines that job requires [91]. This could also be solved through integer programming or constraint optimization using one of several available solvers available, such as we demonstrate with GNU Linear Programming Kit (GLPK) [92] in Section 4.3.10.

However, due to the nature of our problem and how it can be modelled using a graph, we have chosen to use a graph algorithm, maximum weight matching, to approximate the optimal schedule. This is not the first time that maximum weight matching has been used as a scheduling solution. It has long been used as a packet scheduling algorithm for switches and routers, where it has been shown to given the maximum possible throughput [93].

4.3.2. Batch Scheduling Problem

Our problem can also be formulated as a 0-1 integer programming problem. Given a set of benchmarks that we need to run, N, a list of batches that we can assign each benchmark to, $1 \dots M$, and a set of Regions, R, that each have a specified capacity, with

$$w_i = \textit{weight} \text{ of benchmark } i \text{ for } i \in N$$

$$r_{jk} = \textit{capacity} \text{ of region } k \text{ in batch } j \text{ for } k \in R, j \in M$$

Here, the weights represent the resources each benchmark takes up (vCPUs). For simplicity, each benchmark has a single weight, where in reality each benchmark could have multiple weights representing different sizes of VMs allocated for the benchmark. This would make the formulation slightly more complex. M is a number of batches that is higher than what we will need. Additionally, there are assignment restrictions, given by the sets $A_k \subseteq N$, where A_k specifies which benchmarks can be run in region k.

For our objective function, we want to minimize the number of batches in set M that have any benchmarks in set N assigned to them. More formally, we can define this as:

$$\begin{aligned} \text{minimize} \quad & \sum_{j=1}^{M} y_j \\ \text{subject to} \quad & \sum_{j=1}^{M} x_{ij} = 1, && \text{for all } i \in N \\ & \sum_{i \in N} w_i x_{ij} \leq r_{jk}, && \text{for all } j = [1 \dots M], k \in R, i \in A_k \end{aligned} \tag{4.5}$$

$$x_{ij} = \begin{cases} 1, & \text{if benchmark } i \text{ is assigned to batch } j \\ 0, & \text{otherwise} \end{cases}$$

$$y_j = \begin{cases} 1, & \text{if } \sum_{i \in N} x_{ij} > 0 \\ 0, & \text{otherwise} \end{cases}, \text{for all } j \in M$$

If we remove the objective function and the variable y, we end up with a 0-1 integer program with only constraints that must be satisfied. This will return true if it is satisfiable and false if not. That is, it will return true if we can fit the benchmarks successfully into M batches and false otherwise. This fits the definition of the $0-1$ integer programming problem from Karp's 21 problems and is NP-Hard [94] [95].

4.3.3. Scheduling solution

To solve this problem we have implemented a solution that we are calling **PKB_scheduler**. It works as a layer on top of PerfKit Benchmarker, the tool we use to execute cloud based benchmarks, which we previously discussed in Subsection 3.2. PKB_scheduler takes into account all relevant resource usage and limits, dynamically updates the resource usage if another workload is running or any other changes have occurred, and schedules all of the benchmarks into batches of benchmarks that can run simultaneously without interfering with each other or exceeding resource limits. To do this, our tool organizes the benchmark configurations into a graph data structure, which represents Virtual Machine setups as nodes (vertices) and benchmarks as edges. This is useful as almost all of our tests are network tests that take place between two machines.

When we have all of the virtual machines and network benchmarks represented as a undirected multi-graph, we can use existing algorithms to find the maximum cardinality matching of the graph. This matching represents the greatest set of edges that share no nodes. In terms of our what our graph represents, this is the maximum amount of benchmarks we can run at once, without running any two benchmarks from any of the same virtual machines at the same time.

This is a necessary condition, as if we were to run two benchmarks simultaneously that each used at least one of the same virtual machines, that could skew the results of both benchmarks. For example if we ran two iPerf benchmarks to measure throughput and both benchmarks ran from the same client VM, they might both only see half the throughput they otherwise would. This would obviously lead to inconsistencies in the data which can effect our analysis, prediction models, and anything else that uses the data we are gathering with this tool. Using the maximum matching guarantees that this scenario will not occur.

4.3.4. Architecture of PKB_Scheduler

The architecture diagram in Figure 4.7 details how our testing setup functions. PKB_scheduler is a Python program that ingests configuration files of benchmarks that need to be run and dynamically creates batches of benchmarks to run. It then runs them by calling PerfKit Benchmarker (PKB) for each benchmark configuration [34]. This vastly cuts down on the amount of complexity that we need to deal with and prevents us from trying to 'recreate the wheel' by abstracting away all the code that actually creates the cloud infrastructure and executes the benchmarks. We can focus on batch scheduling instead.

Part of what prompted the idea of creating a batch scheduling tool for benchmarks was that when performing benchmarks with PKB, running a single benchmark such as iPerf would take as long as 8 minutes with default settings on Google Cloud. Of that time, 4 minutes would be used to create the virtual machines, firewall rules, and provision the machines with the appropriate software, while only a few minutes would be used to run the benchmark and the rest of the time was used to tear down cloud infrastructure previously created. The amount of time dedicated to each task depends on the benchmark being executed and the options selected for it, but in all cases there is a significant amount of time devoted to creation, preparing, and cleaning up resources.

In order to reduce the amount of time spent on these auxiliary tasks, we devised a method to efficiently reuse the same VMs for multiple benchmarks that share identical virtual machine and resource specs. If benchmarks require different specifications, unique virtual machines will be created for each unique specification. For example, a 2 CPU machine and a 4 CPU machine both in the same region are different configurations and thus we cannot reuse one in place of the other. This method involves representing our benchmark and VM configurations as a graph.

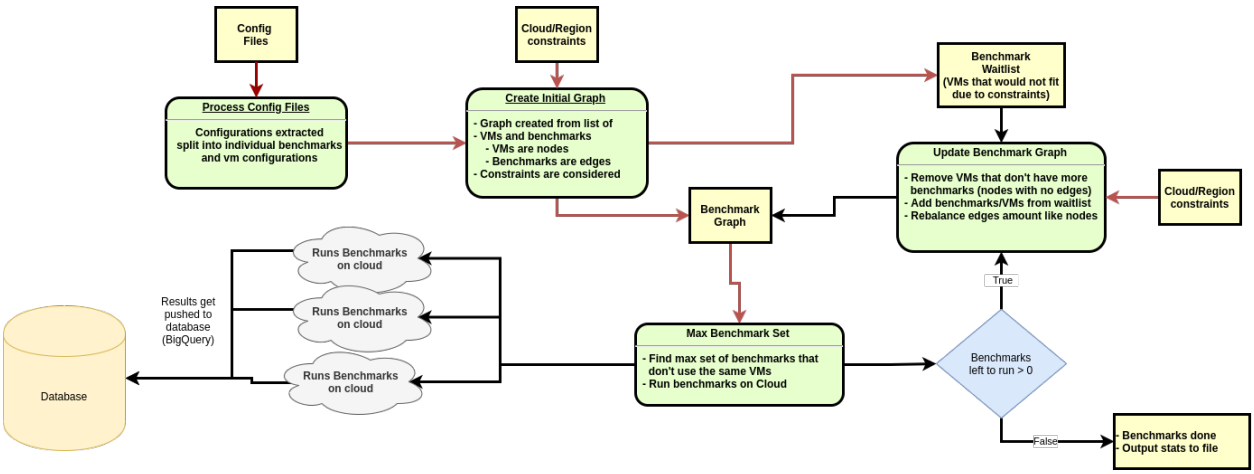


Figure 4.7: Flow diagram of PKB_scheduler program logic

4.3.5. Graph Representation of Benchmarks and Virtual Machines

Benchmark configuration files are parsed and turned into a multi-graph representation, where nodes represent virtual machines and their associated resources and edges represent benchmarks that run between the virtual machines. Because it is a multi-graph, there can be multiple edges between the same nodes, representing multiple different benchmarks to run between the same pair of virtual machines. The graph can also contain self-loops, representing benchmarks that require only a single virtual machine. We also keep track of multi-edge sets, representing benchmarks that use more than 2 virtual machines. The tool also keeps track of the available resources for each cloud provider and cloud region being tested to make sure that we do not try to schedule benchmarks or stand up VMs that would cause us to exceed the resource limitations. This gives us two problems: allocating resources most efficiently in each cloud and cloud region and scheduling benchmarks efficiently using the allocated resources.

Benchmarks and their associated VMs are added to the graph from a list of configurations in a greedy manner. Any benchmarks that require resources that currently exceed those available will be placed on a wait-list until other benchmarks finish and those resources are freed. Resource limit tracking and wait-listing are essential for two scenarios. The first is

when we have multiple benchmarks that need to use VMs with different specs in the same region that has a limited quota. One benchmark and its associated VMs will be added to the graph and the other will be added to the wait-list. After the first benchmark finishes, it will be removed from the graph and the other will be added. If these two benchmarks had used VMs with the same specs, they could have reused the same VM and the benchmark would have been added as a second edge between the two original nodes. The second scenario is when we have several benchmarks that use the same VM configurations. We could add them to the graph as new edges between the same pair of nodes, in which case they would execute sequentially. However, if we have the resources available, we can allocate duplicate VMs with the same configurations, in which case they will execute in parallel. It is also possible that a mixture of the two could happen if there are some available resources but not enough for all benchmarks. Some benchmarks will be allocated duplicate VMs which will be added as new nodes on the graph, and some benchmarks will be added as new edges between existing nodes.

This problem of allocating VMs for benchmarks in regions which often have multiple different quotas associated with them as well as cloud wide quotas could be thought of as several connected multiple constraint bounded knapsack problems. Each region is a knapsack, its quotas are the constraints and the VMs are the items to be placed in each knapsack. We seek to maximize the number of VMs of the types we need in each region up to the number of benchmarks we need to run with those VMs. However, many benchmarks require multiple VMs in different regions, so many VMs do not stand alone and must have associated VMs in other regions. So we could not consider any one region on its own, we must maximize the number of VMs for all regions in use. Because of the complexity of this problem, we have chosen to use a simple greedy approach, where benchmarks are sorted largest first by the relative amount of each quota they take up and then the VMs for each benchmark configuration on the list is either allocated or not depending on whether

there is still space. This is similar to the greedy algorithm developed by George Dantzig to approximate a solution to the knapsack problem [96].

It should also be noted that the resource limits and quotas that the tool tracks are only the ones we can view as a user. There may be "invisible" resources such as the bandwidth for a host machine that could be saturated if duplicate VMs happened to be allocated on the same host. These would be function as a type of silent interference that we are unable to detect. Some such effects can be mitigated, however, if the cloud under test allows you to define a placement policy that would spread out new VMs across host machines.

4.3.6. Efficient Scheduling

Now with an understanding of how benchmarks and VMs are represented in our data structure, we can discuss the process of how benchmarks are scheduled and executed. First, configuration files for the benchmarks are parsed in and resource constraints are gathered from the cloud environment(s) under test. Then a graph is created from the benchmark configurations in the manner described in subsection 4.3.5. Any benchmarks and virtual machines that do not fit in the graph due to resource constraints are added to a wait-list. Once the initial graph and wait-list are created, the program starts the benchmarking process.

This process is divided into rounds. In each round, first, a maximum matching is found to give us our set of edges/benchmarks to use. This will be further explored in subsection 4.3.7. The benchmarks associated with the edges in the maximum matching are then executed using PerfKit Benchmarker. Edges and their associated benchmark are removed from the graph after they have completed. When a node has no edges left (i.e. has a degree of 0), we remove the node from the graph and shutdown its associated virtual machine and related resources. Then we attempt to add any benchmarks from the waitlist now that resources

may be available from shutting down finished VMs. Finally, we attempt to redistribute the edges in the graph. If a node has significantly more edges than nodes with equivalent virtual machines, we can try to redistribute the edges among them. We have found that this is essential to reducing overall execution time. After the round is finished, we start the next round and continue until there are no more benchmarks left to run. This will be apparent as both the graph and wait-list will be empty.

4.3.7. Maximum Matching

In this model, benchmarks are scheduled into batches by finding the maximum matching (or maximum cardinality matching). This is a fundamental problem in graph theory and is well solved. It can be formally defined as follows [97]:

Given a graph $G = (V, E)$

Definition 1 (Matching) *A set of edges $M \subseteq E$ is a matching if no V in G is incident to more than one edge in M.*

Definition 2 (Maximum Matching) *M is a maximum matching if, for any other matching M', $|M| >= |M'|$, where $|M|$ is the maximum sized matching.*

Or more simply put, a matching is a set of edges where no edge shares a vertex and the maximum matching is the matching with the largest set of edges possible.

Edges in this algorithm are benchmarks to be run next, and vertices are virtual machines that have been spun up with specific hardware in specific regions to run the benchmarks. We use this because it ensures that we don't run two benchmarks from same VM at the same time. This is an important requirement for the benchmarks to produce unbiased results.

If there is sufficient capacity, multiple VMs can be created in the same zone to increase the amount of tests we can run simultaneously. On a cloud with enough capacity, this would result in a pair of nodes created for each benchmark and all benchmarks would be run in parallel.

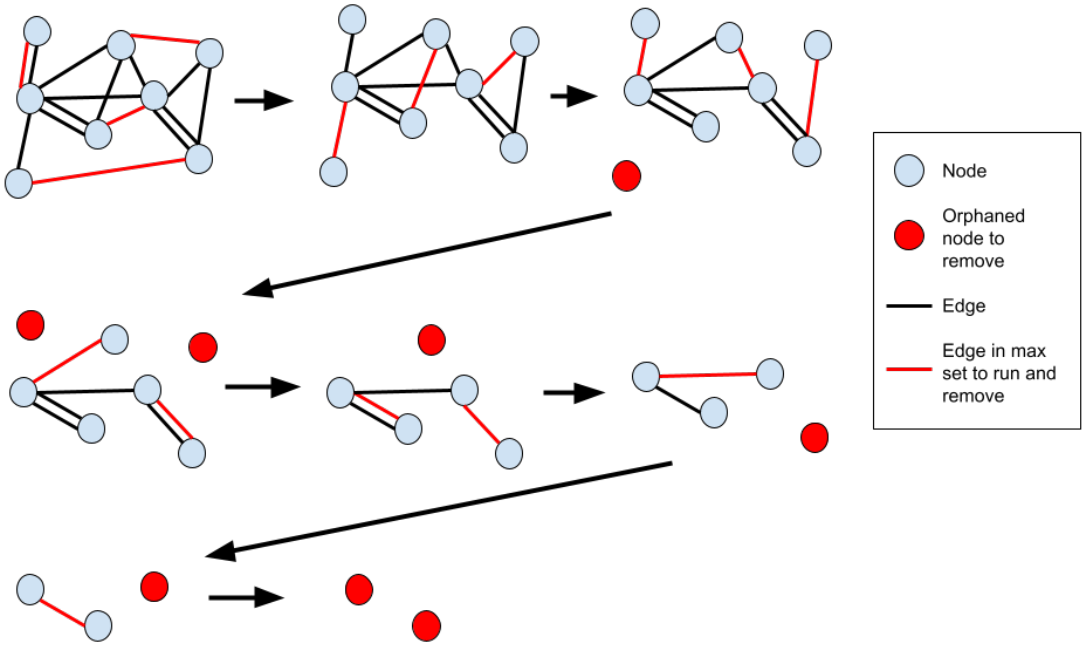


Figure 4.8: Maximum matching of nodes and deletion of finished nodes

In Figure 4.8, we show a step by step example of how the process of test selection and pruning of edges and nodes works. In the first step, the maximum matching is found and and those tests are run. The benchmarks being run are represented by the edges highlighted in red. After they are run and results are stored, those edges are pruned from the graph. VMs (nodes) are then also removed from the graph if they have no more benchmarks (edges) to run. These VMs to be removed are highlighted in red. This process loops until there are no more edges or nodes in the graph. At this point we know that we have run all of our benchmarks.

To find the maximum matching for PKB_scheduler, we are using the Blossom algorithm, implemented in networkx's max_weight_matching function [98]. This algorithm takes an

undirected graph, and the weight of each edge, and will return the maximum matching for the graph. There are two theories that mainly make up this method: the blossom algorithm and the "primal-dual" method. The blossom algorithm is utilized for finding augmenting paths where each vertex in the graph has at most one edge, and the number of edges is maximized [99]. This algorithm is optimal for our matching because it is a maximum-size matching algorithm that runs for polynomial computation time. The primal-dual method is utilized to reduce weighted optimization problems to combinatorial, unweighted problems.

To allow the use of benchmarks which require more than 2 VMs, we keep a list of what we refer to as multi-edge sets. We augment the set of benchmarks returned by the maximum matching function by replacing benchmarks with edges that are incident with nodes in multi-edge sets with the benchmarks for those multi-edge sets if that would result in a larger total set of benchmarks. During this process, we also ensure that our constraint that no two benchmarks share the same VM in a batch is not violated.

In terms of time efficiency, we have found this method to have an acceptable overhead. It has a time complexity of $O(|E||V|^{1/2})$. In Figure 4.9, we can see that for 124 nodes and 776 edges, matching takes less than 0.13 seconds.

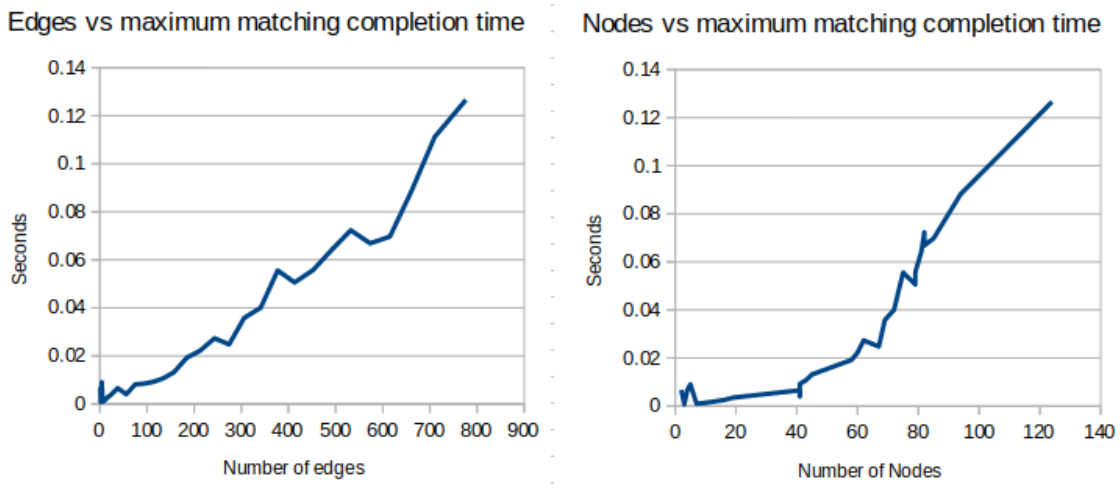


Figure 4.9: Nodes vs completion time of maximum matching and Edges vs completion time of maximum matching

4.3.8. Edge Redistribution

As mentioned in 4.3.5, initially, benchmarks and virtual machines are allocated in a greedy manner. If there is space to allocate all virtual machines for a benchmark, they are allocated. If we are running two benchmarks that require the exact same virtual machine specifications, two sets of those virtual machines will be created if there is enough space. If not, the second benchmark will reuse the same virtual machines as the first. In our graph representation, the first scenario would be represented by two sets of nodes with each set connected by an edge, but no edges between sets. The second scenario would be represented by one set of nodes connected with multiple edges. If there is insufficient space and many benchmarks that use a single VM configuration, this can lead to a situation where one node has many edges. Due to how maximum matching works, all of the benchmarks for these edges will have to execute in sequential rounds, leading to a possibly long execution time.

To alleviate this, we can redistribute the edges between nodes that have VMs with the same specs or create duplicate VMs if resources have become available after each round. We have found that this significantly improved execution time and the number of rounds that needed to be performed. The process of edge redistribution can be seen in Algorithm 1, equalize_graph. In this algorithm, we first find the node with maximum degree in the graph. (This is actually somewhat simplified for the sake of space. In our actual implementation, we perform this procedure for the nodes with the top n degrees in the graph.) When then find a list of nodes that represent equivalent VMs. Then we loop through those equivalent nodes and attempt to redistribute the edges from the original nodes to the equivalent nodes to make them all have similar degrees.

Algorithm 1 equalize_graph

```
1: procedure EQUALIZE_GRAPH(graph G)
2:     max_n ← max_degree_node(G)
3:     equivalent_list ← equivalent_nodes(max_n.vm)
4:     sort(equivalent_list)                                  ▷ sort by degree, desc
5:     equality_improved ← True
6:     while equality_improved == True do
7:         equality_improved ← False
8:         for n in equivalent_list do
9:             if n.degree < max_n.degree − 1 then
10:                transfer_edges(max_n, n)
11:                equality_improved ← True
12:            end if
13:        end for
14:    end while
15:    if max_n.degree > 1 then
16:        new_n ← duplicate(max_n)
17:        transfer_edges(max_n, new_n)
18:    end if
19: end procedure
```

4.3.9. Benchmark Execution

To those familiar with PerfKit Benchmarker (PKB), you may note that this is not generally how PKB runs tests. Usually with PKB, benchmarks follow the pattern of: network/infrastructure setup, VM setup, benchmark execution, VM teardown, and finally network/infrastructure teardown. This is done for each benchmark that is run. With PKB_scheduler, we might run many benchmarks between VMs before tearing them down.

To get our architecture to work with PKB, we use a couple of existing constructs as well as a slightly modified version of PerfKit Benchmarker. For the creation and deletion of VMs we use a purpose built "benchmark" in PKB called vm_setup that does just what its name implies: it sets up whatever VM configuration is passed to it through a config file. PKB is split into phases which can be one at a time and stopped between phases. So to setup a VM we call the setup and provision phases of the "vm_setup" benchmark in PKB. Then to

teardown a VM we call the teardown phase of the same "vm_setup" benchmark, which we can do using a unique id generated for each run of PKB.

Once we have VMs setup using the "vm_setup" benchmark, we need to run actual benchmarks between them. To do this, we use the static VM feature of PKB. This allows us to run benchmarks on VMs that are already created given we have their IP addresses and ssh credentials. Essentially we treat the VMs we created earlier as static VMs and run benchmarks between them using this method. Then once a VM has no more benchmarks to run, we call the teardown phase of "vm_setup" for that VM.

Using PerfKit Benchmarker in this manner, though somewhat unconventional to its standard use case, allows us to focus on the scheduling aspect of this tool, rather than implementing the benchmarks we want to run from scratch. Another way that we could have done this is to use an infrastructure as code tool like Terraform to handle VM creation and deletion and use PKB to run benchmarks between VMs created by Terraform. We chose this path mainly because of our familiarity with PerfKit Benchmarker.

If you want to avoid this behavior entirely and not share VMs across multiple benchmarks, there is an option in the tool to do that. When using this option, it will still schedule benchmarks in the same manner, but will not share VMs across multiple rounds of benchmarking. VMs will be created before and destroyed after each benchmark, which is the default behavior of PerfKit Benchmarker. This option can be used if the benchmarks you are running alter the VM in some manner without resetting it at the end. This way each subsequent benchmark starts off with a fresh VM. Generally using this option is somewhat slower because of the compounded setup and teardown times, but benchmarks are still efficiently scheduled and VMs allocated based on maximum weight matching and resource limits.

4.3.10. Experimental Results

In this section we use our tool against several sets of benchmark configurations. We will compare those benchmarks running with our shared VM optimizations vs without. We will also see how efficiently this greedy method works to pack benchmarks.

Table 4.4: Running time for each number of benchmarks

Number of benchmarks	**Reuse VMs**	**No Reuse VMs**
1	573s	520s
2	571s	541s
4	576s	530s
8	913s	1073s
16	1577s	2081s
32	2909s	4189s
64	5585	8331s

For the sake of consistency all of our benchmarks in this section will have a duration of 5 minutes, not including setup and tear down. In Table 4.4, we can see the running time of various numbers of benchmarks when using PKB_scheduler both with reusing VMs and without. For this experiment we ran the same benchmark with the same configuration multiple times. The difference in execution time shown is setup and tear down of virtual machines. For running up to 4 benchmarks, using the option to not reuse VMs is actually quicker, but when running more benchmarks than this using the option to reuse VMs is quicker. This is explained by the VM size we were using and the resource limit in the cloud region under test. Only 4 machines of the specified size would fit in this region. Because of this, up to four benchmarks could run in parallel using duplicate VMs, so the run times for these tests are all roughly similar. Reusing VMs here takes slightly longer because there is some overhead with how we separate VM creation from benchmark execution. When running more than 4 benchmarks, the VMs can be reused, which we can see results in lower execution times than not reusing VMs. This averages to a time savings of around 150 seconds per round of reused VMs.

For a more realistic test, we have run a network benchmark between pairs of VMs in all regions of a cloud environment using multiple machine types. Each of these regions has varying resource limits that must be accounted for. We ran this set of benchmarks in 3 ways: 1) using PKB_scheduler with maximum matching scheduling and sharing VMs across rounds. 2) using PKB_scheduler with maximum matching scheduling, but not sharing VMs across rounds. 3) Using just PerfKit Benchmarker and specifying a number of benchmarks that can be run in parallel without exceeding resource limits for even the region with the least available resources. The results are listed in Table 4.5.

Table 4.5: Running Time for different methods of benchmark scheduling

Method	**Completion Time**
PKB_scheduler Share VMs	16595s
PKB_scheduler No Share VMs	22296s
PerfKit Benchmarker No optimizations	90302s

Using PKB_scheduler with sharing VMs enabled was the quickest to complete. Even without sharing VMs, our graph based scheduling was a considerable improvement over no optimization.

From that same test with sharing VMs enabled, we also captured data detailing how many VMs were running during each round, and how many were being used, which can be see in Figure 4.10. Here, more than 80% of allocated VMs are used in each round and in 5 of 6 rounds, we have over 90% usage.

In Figure 4.11, we can see the quota usage for a cloud region for each round of benchmarks. We have only included a single region, because they all looked vastly similar to this example. For the first few founds they are efficiently packed with VMs, as demand is high. As there are fewer benchmarks left to run, usage tappers off quickly. This shows that our method of VM allocation works to efficiently place VMs, leaving little to no unused resources when there is sufficient demand.

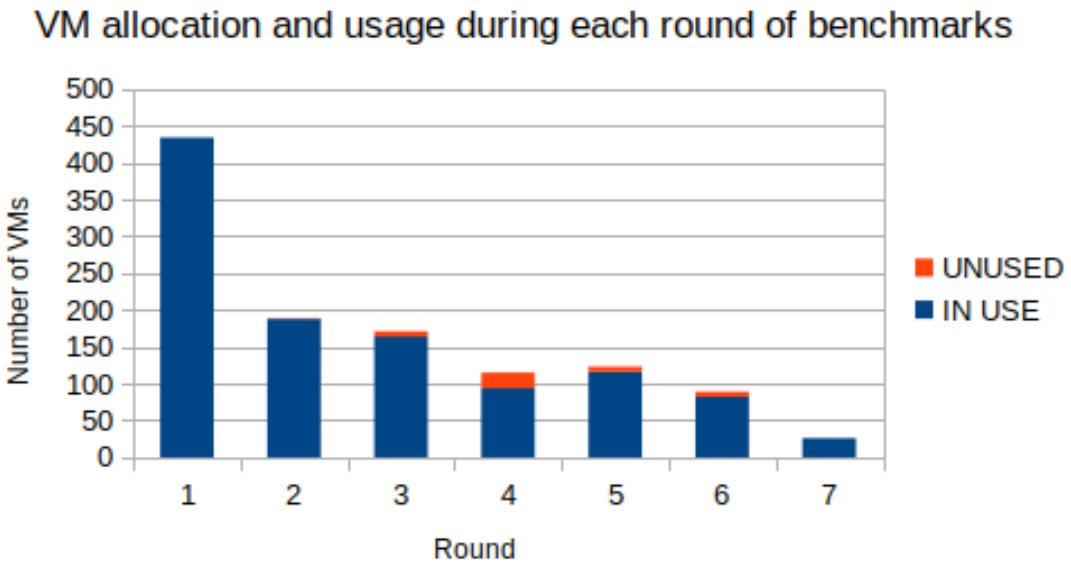


Figure 4.10: Capture of VM usage and Allocation during each round of benchmarks

Generally, the results of this method are the best when used with a set of benchmarks that share many repeated VM configurations, such as when performing network benchmarks between all distinct pairs in a set. This situation allows the most reuse of the existing VMs. On the other hand, if working with a small set of benchmarks, or benchmarks that do not share repeated configurations, there will be little speed-up from VM reuse, but can still benefit from efficient scheduling.

4.4. Conclusions and Future Work

This chapter we discussed how we can increase the efficiency of our benchmarks, both on an individual level and at larger scales. For individual benchmarks, we focused on taking accurate throughput measurements while minimizing the amount of time and data spent on those measurements. We proposed using confidence intervals with a user specified confidence level and width and automatically stop the network throughput test when the target width is reached. We have shown that this can reduce the test length and data consumed in a variety of cases and the worst case scenario in this regard is that the test runs for the full

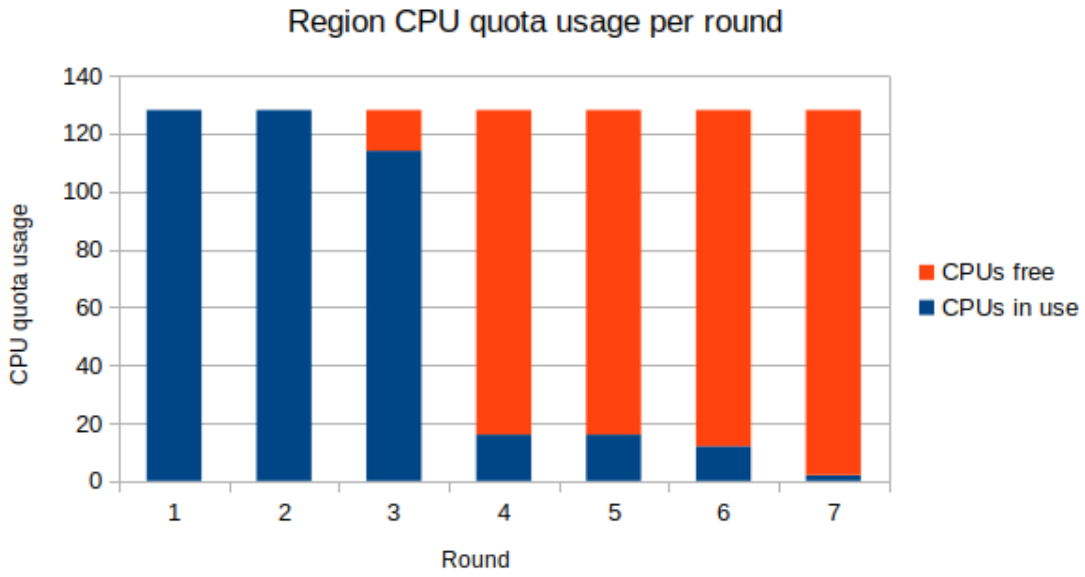


Figure 4.11: Quota usage for selected cloud region during each benchmarking round.

specified test time and the specified width is not reached. This would be equivalent to simply running a network test with a static length.

We have found this technique to be most useful for high throughput links, where achievable throughput may vary from sample to sample based on network conditions, such as cloud networks. These improvements are not universal. For already short measurements or lower capacity networks, the improvements may not be nearly as dramatic. This method is also highly customize depending upon the performance needs of your operation. It can easily be adjusted for lower confidence or a larger target width. We also suggest displaying throughput as a range is a good idea, especially in environments with high throughput variance. To continue this work, we believe that if we are able to add in additional historic data from previous tests, or an estimate of what we think the throughput should be, we may be able to converge to an accurate result more quickly than our current method.

For running larger numbers of benchmarks, we have created a graph based benchmark scheduling tool. We have shown that our tool can be an effective and efficient means of run-

ning a large number of benchmarks on a cloud environment. We tested multiple methods of batch scheduling and found that all of our methods were better than a naive approach. From our tests PKB_scheduler creates the same number of batches as using integer programming optimization and when using VM sharing across batches, PKB_scheduler can outperform this approach in terms of execution time by around 17%, depending on the length of benchmarks being executed.

However, there are still several optimizations that we would like to implement. These include tweaking the scheduling algorithm to consider benchmark time. Currently, most of our benchmarks take around the same amount of time to complete. If you attempt to schedule benchmarks with vastly disparate durations, VMs running benchmarks with shorter durations would be forced to sit idle while the longer benchmarks finish. This could be solved in several ways, the simplest of which would be to include in the configuration files the approximate time benchmark execution takes and use them as edge weights during maximum matching. We would also like to improve the allocation cloud resources from a relatively simple greedy approach to a dynamic programming algorithm like those often used to approximate the knapsack problem.

Chapter 5

Network Analytics of Cloud Networks

5.1. Introduction

Data center and scientific computing applications are becoming increasingly distributed across geographically separated compute and storage facilities. Computing and data nodes may be dispersed across different sites to serve various clients, for example, to widely disseminate scientific data and support web searches. Widely distributed scientific and cloud computing applications can often face communication challenges when they need to sustain high bandwidth data transfers between sites, such as when distributing large datasets or media. To distribute very large datasets specialized applications such as Globus [100] are often used, but it is also essential that the underlying network can support high bandwidths with a low error rate.

These scenarios require efficient data transport networks, which can be readily deployed by utilizing virtual machines (VMs) at cloud sites and connecting them using virtual IO and virtual links. The application performance critically depends on an efficient transport of data over the wide-area networks that connect these sites. Extensive data transport networks are being built, connecting data center and supercomputer sites to support these scenarios. These types of data transfers are becoming increasing common as more businesses move their workloads and data storage to public cloud environments with sites distributed around the world. For scenarios suited for such cloud networks, there are several advantages including fast deployment, flexible pricing, and the infrastructure being maintained by cloud providers.

In this chapter, we analyze the performance of a public cloud network, specifically Google Cloud, in comparison with dedicated physical infrastructures using throughput profiles, which show the throughput for a network across different RTTs. An example of these profiles is shown in Figure 5.5, where the top two rows correspond to the cloud network and bottom two to the dedicated testbed connections. In each of these scenarios, public cloud and emulated, the goal is to maximize TCP throughput performance. This means not only selecting the optimal network parameters such as TCP buffer sizes and TCP congestion control algorithm, but also how many parallel flows we should use to send data. The primary motivation behind this is to better understand how the public cloud network compares to the emulated network and to evaluate how parallel flows, congestion control algorithm selection and loss affect the throughput profiles for these networks.

To support various network transfers, it is helpful to understand the maximum achievable throughput and actual measured throughput between servers and clients over network connections for a range of RTTs. For this objective, we study the throughput and latency characteristics of the underlying network infrastructure and the impact of protocol parameters, RTT and network losses. To accomplish this, we gather throughput measurements for memory-to-memory transfers at RTTs from 1ms to 350ms across the Google Cloud network. We repeat these measurements using from 1-10 parallel flows and with multiple different TCP congestion control algorithms. We take the same set of measurements on our testbed, where we emulate connections with the same RTTs and bandwidths as the Google Cloud connections. Additionally, on our emulated network we test these parameters with various rates of random packet loss, which we cannot control on the cloud network. Both of these data sets are available upon request.

From these measurements, we can also calculate the utilization-concavity coefficient (C_{UC}) for each throughput profile [101]. This coefficient relates to the overall shape of the throughput profile, how concave it is and how efficiently it is using its available band-

width. Here, a throughput profile with a more concave shape indicates closer to optimal performance and a throughput profile that is more concave is indicative of a connection that is not meeting its maximum capacity. The C_{UC} has a range of $[0, 1]$, where closer to 0 is convex and is most likely facing a significantly restricting bottleneck and a C_{UC} closer to 1 is more optimal, where ~ 0.75 represents an ideal profile with no loss under emulated conditions. Using this metric allows us to more easily compare throughput profiles across different networks and parameters.

Overall, our results indicate a comparable performance of cloud networks shown by the concave-convex geometry of profiles that indicate the optimization level with smaller variations due to virtualization of hosts and connections. For practical considerations, the number of parallel flows remains a dominant factor for throughput optimization across various conditions and TCP versions, particularly at higher RTTs.

For a discussion on existing works in transport performance on different networks, refer to Chapter 2.2. The rest of the chapter is organized as follows: In Section 5.2 we show how we calculate the Utilization-Concavity Coefficient for the throughput profiles and how we can use this to compare different networks Section 5.3 details our experimental setup and how we collected our measurements. Section 5.4 presents the results of our throughput tests and examines how each of several variables effects the throughput profile for a network. Section 5.5 summarizes our findings and discusses the future directions of our research effort.

5.2. Throughput Profiles

In this section we define the metrics we use for comparing networks and their configurations. As defined by Rao et. al. [102], the throughput profile for a given set of hosts on a network and their respective configurations and TCP parameters is defined as:

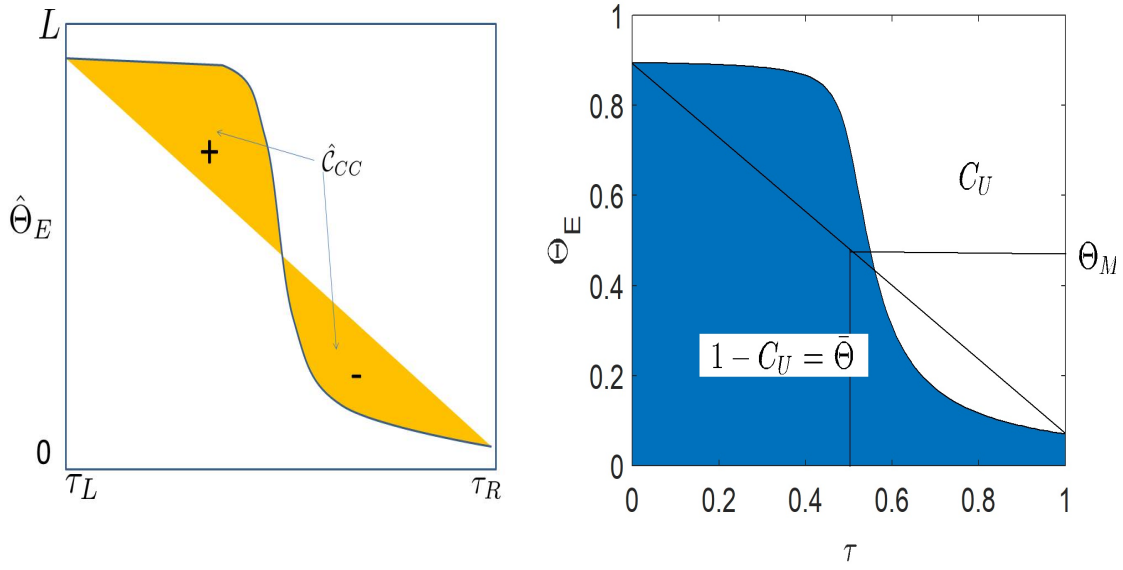
$$\Theta_O(\tau) = \frac{1}{T_O} \int_0^{T_O} \theta(\tau, t)\, dt \tag{5.1}$$

Where $\theta(\tau, t)$ is the throughput of a connection at time t for an RTT τ. The throughput profile, $\Theta_O(\tau)$, is the average of these observations over a period of time T_O. Several examples of throughput profiles that we will discuss more thoroughly later can be seen in Fig. 5.5. It is well established that throughput always has an inverse relationship with RTT across a network. In an ideal situation, we want this throughput curve to be concave, i.e. throughput should decrease steadily and slowly as RTT increases, such as can be seen in Fig. 5.5i. The extent of how sharply throughput drops as RTT rises depends upon several parameters including the TCP buffers, TCP congestion control algorithm, total network bandwidth, and other bottlenecks such as CPU performance and memory. It can also depend upon things that are out of our control such as the amount of competing traffic and network outages, especially in a public cloud environment.

5.2.1. Utilization-Concavity Coefficient

Here, we will be discussing the Utilization-Concavity Coefficient (C_{UC}) [7] and how we can use it to compare the profiles of different networks. First, we must define this coefficient. The C_{UC} of a throughput profile, $\hat{\Theta}$, is a measurement of how well utilized a network is across a range of RTTs. Before we can define it however, we must first examine its components. Here, let L be the maximum capacity for a connection, τ_L is the smallest RTT and τ_R is the largest RTT on the network. With this, we can define the under utilization coefficient as:

$$\mathcal{C}_U(\hat{\Theta}) = \int_{\tau_L}^{\tau_R} \left(L - \hat{\Theta}(\tau) \right) d\tau \tag{5.2}$$



(a) Convex-concave $\mathcal{C}_{CC}$ (b) Utilization coefficient $\mathcal{C}_U$

Figure 5.1: Throughput profiles and coefficients

This summarizes how underutilized connections are on a network across its range of RTTs. To represent the relative concavity of a throughput profile, we have the convex-concave coefficient, which is defined as:

$$\mathcal{C}_{CC}\left(\hat{\Theta}\right) = \int_{\tau_L}^{\tau_R} \left(\hat{\Theta}(\tau) - \left[\hat{\Theta}(\tau_L) + \frac{\hat{\Theta}(\tau_R) - \hat{\Theta}(\tau_L)}{\tau_R - \tau_L}\,\tau\right]\right) d\tau \tag{5.3}$$

$$= (\tau_R - \tau_L)\left[\bar{\hat{\Theta}} - \hat{\Theta}_M\right] \tag{5.4}$$

Putting these two coefficients together, we can have a more complete summary of both the concavity and the utilization of our throughput profile, giving us the utilization concavity coefficient, which can be defined as:

$$\mathcal{C}_{UC}\left(\hat{\Theta}\right) = \frac{1}{2}\left(\left[1 - \mathcal{C}_U\left(\tilde{\Theta}\right)\right] + \left[\frac{1}{2} + \mathcal{C}_{CC}\left(\tilde{\Theta}\right)\right]\right) \tag{5.5}$$

Here, $\tilde{\Theta}$ is a normalized version of the throughput profile $\hat{\Theta}$, with values scaled by L and $[\tau_L, \tau_R$ is scaled to $[0, 1]$. This also bounds the C_{UC} to $[0, 1]$.

We can also define this more compactly as:

$$\mathcal{C}_{UC}\left(\hat{\Theta}\right) = \bar{\tilde{\Theta}} - \tilde{\Theta}_M/2 + 1/4 \tag{5.6}$$

where $\bar{\tilde{\Theta}}$ is the average throughput weighted by RTT and $\tilde{\Theta}_M/2$ is throughput at the midpoint.

The C_{UC} succinctly summarizes the properties of the throughput profile that we are most interested in: total utilization, and the concavity of the curve). Using this coefficient, we can more easily compare throughput profiles on networks that might have different capacities, bottlenecks, and other limitations.

5.3. Measurements Collection

We have taken TCP throughput and RTT measurements from two networks to compare. The first network is Google Cloud, a public cloud network with data centers distributed across the world with a global network to connect them. The second is a emulated network operating on a testbed. With this testbed network we can emulate similar connections as we examine on the public cloud network, but under idealized scenarios with no cross traffic, no loss or induced loss, and the same RTTs to those we see on the public cloud network. The connections under test for Google Cloud are shown in Fig. 5.3. Here, we picked a subset of region pairs that gave us a wide range of RTTs to examine and compare. These connections cover most of their available data center regions. In Fig. 5.4, we can see our emulated network setup. Fig. 5.4a shows the logical and Fig. 5.4b shows the physical setup.

5.3.1. Measurement Tools There are several network testing tools available to measure both throughput and latency with the ability to customize a variety of parameters that can effect network performance. These tools include Nuttcp, Netperf, and iPerf. For our measurements, we chose to use iPerf because of its support for multiple parallel sending threads. To measure latency, we are using ping, which is well known and widely used. For a more detail discussion on network performance measurement tools, refer to Chapter 2.1.1.

5.3.2. Google Cloud: Data Center Connections

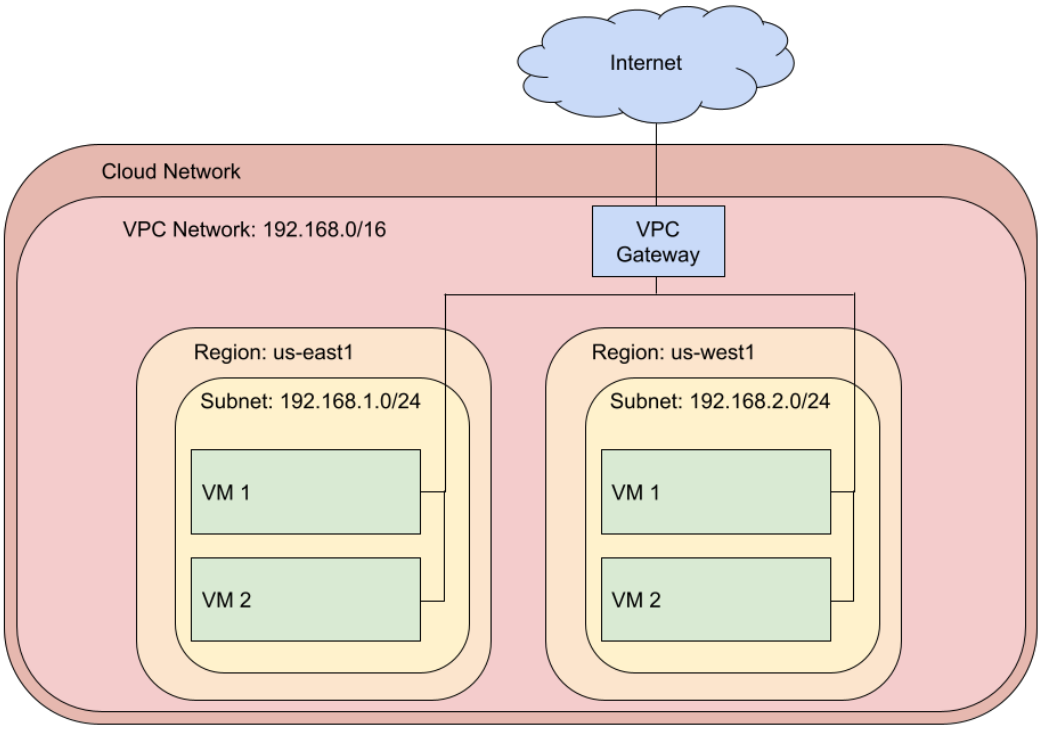


Figure 5.2: Diagram of connections in Virtual Private Cloud (VPC) network on google cloud.

For our tests on Google Cloud, we used n1-standard-4 VMs that have 4 Intel Skylake vCPUs. They all have at minimum kernel version 5.8.0 with Ubuntu 20.04. Network parameters have been tuned to achieve the maximum possible throughput, with TCP buffer maximums set to 250 MB. The default TCP buffer sizes remain default, and we let the operating system adjust the actual window within that range using TCP autotuning. The

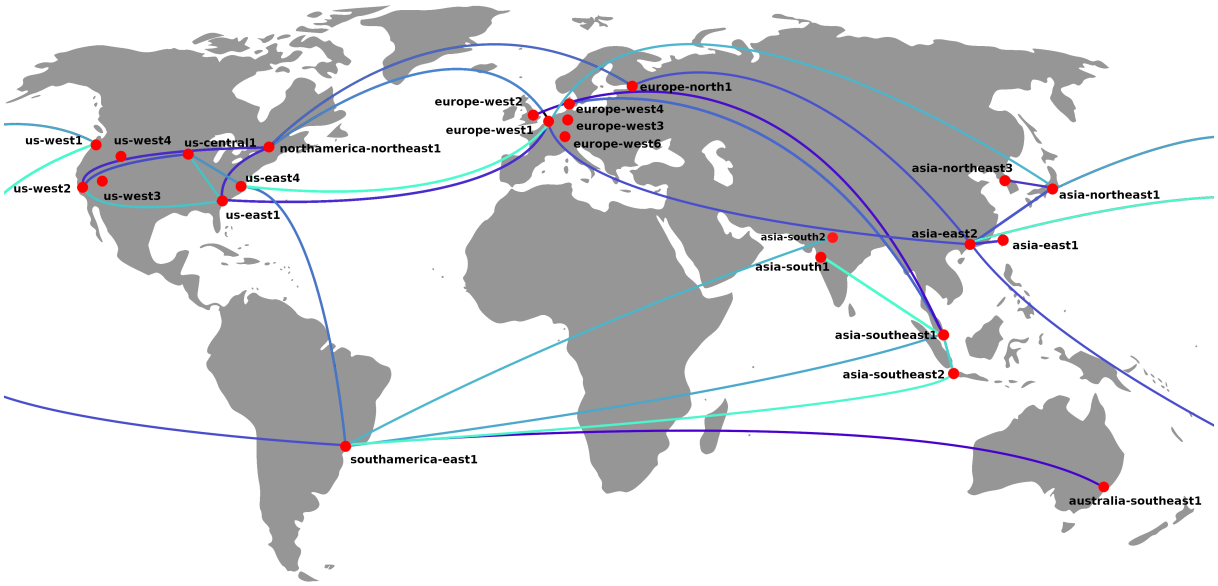


Figure 5.3: Google Cloud with lines representing logical connections between region pairs, with RTTs in [1-350] ms range.

MTU is set to 1500 bytes, which is the current maximum for the network. For each test, we set up 2 VMs in different geographic regions on the same Virtual Private Cloud (VPC) network, which uses Andromeda [103] on the Google Cloud network. This setup can be seen in Fig. 5.2. RTTs for these region pairs under test range from 1 millisecond to just over 350 milliseconds. We executed ping tests for latency and iPerf tests with sending thread counts of [1-10] to gather bulk throughput data. For the iPerf tests, we also collected throughput data for each 1 second interval for the duration of the test. We repeated these throughput tests for three different TCP congestion control algorithms: CUBIC [87], HTCP [86], BBR [104], and the latest version of BBRv2, which is currently still in alpha. We refrained from testing earlier TCP variants such as Tahoe and Reno, as they now represent a very small proportion of total internet traffic [105].

Additionally, we also repeated each of these tests using both Internal (private) and External (public) IP addresses. The main difference between these two addressing options is the bandwidth cap, as they are both often routed along the same physical network path. For internal IP addresses, there is a 10 Gbps egress limit per machine, and for external IP ad-

dresses, there is a limit of 7 Gbps total egress from a single VM as can be seen in Fig. 5.5d-f. We will examine each of these test scenarios separately and compare their performance and dynamics.

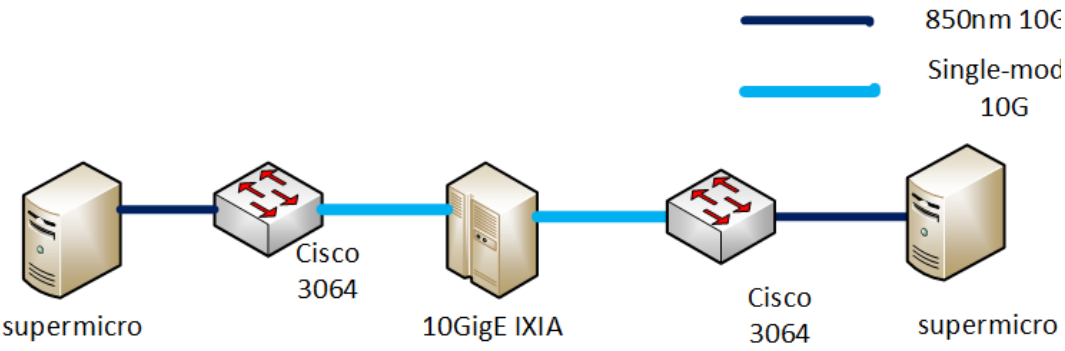
5.3.3. Testbed with Hardware Network Emulators

For the testbed connections, we chose to emulate connections with RTTs that match those that we found on the connections we testing on Google Cloud. For these tests, we are using Redhat 7 and 8 on 32-core Supermicro workstations. Two identical hosts are connected by 10 GigE connections emulated by IXIA hardware emulators that allow us to set the RTT and packet drop rates for the connection.

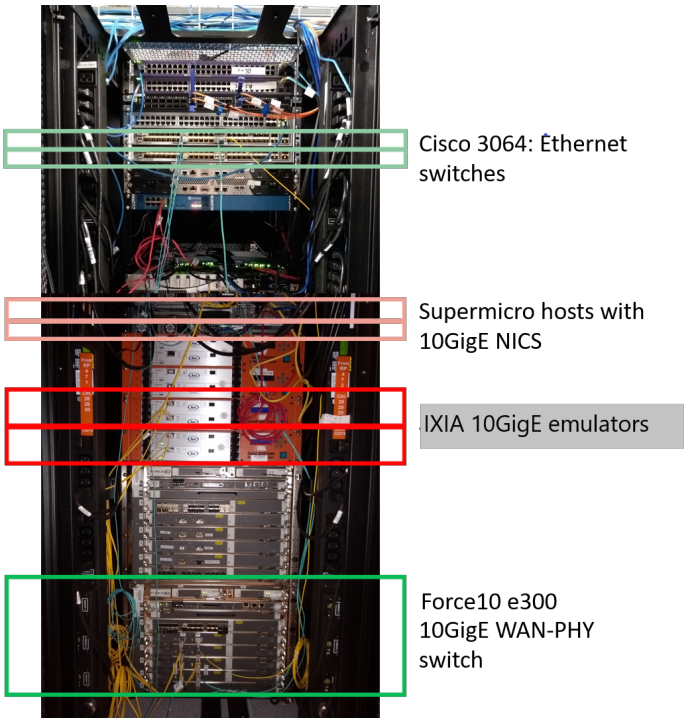
For this setup, Ethernet packets are generated on each host in the same manner we do on VMs on the cloud network. They packets are delayed by RTT and periodically dropped to emulate specific loss rates by the IXIA emulator. We perform memory-to-memory TCP transfers using iPerf (12 TCP versions in total; we show here select CUBIC, HTCP and BBR for space). The collection of measurements is automated with each set taking 1-2 days. The TCP buffer sizes are set to the recommended values for 200 ms RTT and the socket buffer for iPerf is set to 2 GB. These measurements experience no cross traffic or other effects of a multi-tenant environment that may be present in our cloud measurements.

5.4. Results

In this section we will examine the throughput profiles from both Google Cloud and our emulated network and the effect various parameters have on throughput in each of these distinct environments. In Fig. 5.5, we show the throughput profiles for Google Cloud and the emulated network under several circumstances. In Fig. 5.5a-c, we show throughput profile for Google Cloud connections using Internal IP addressing. In Fig. 5.5d-f, is throughput for Google Cloud connections using external IP addresses. Finally, in Fig. 5.5g-i, we show



(a) dedicated 10GigE connection

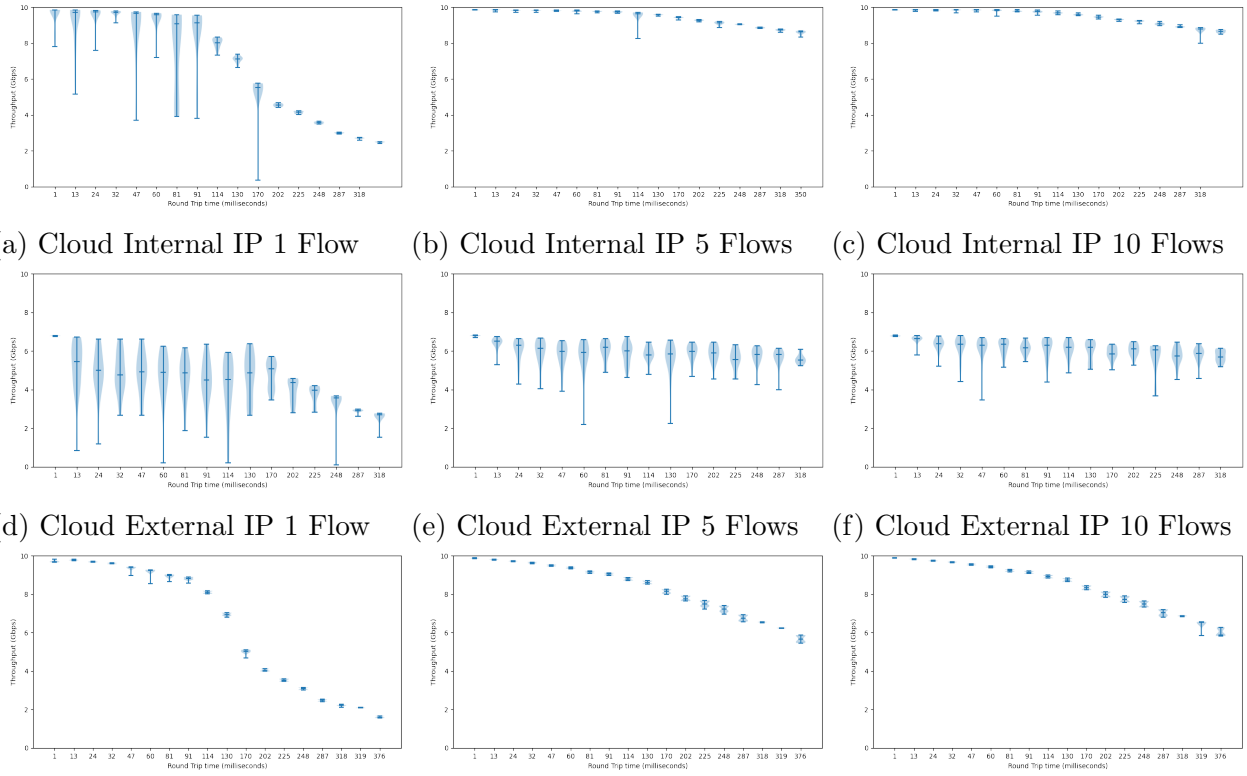


(b) servers, switches and hardware connection emulators

Figure 5.4: Physical testbed for target measurements.

the throughput profiles for the emulated network with the same RTTs as the Google Cloud connections with no loss. The main difference that we see here is that there is much less variance in the emulated network. This is to be expected, as it is operating without cross traffic and dedicated to only these network tests. If we look at the general shapes of these throughput profiles, despite the different circumstances of each, the throughput profiles

themselves have similar shapes. We will also confirm this when we discuss the C_{UC} for each of these throughput profiles.



(a) Cloud Internal IP 1 Flow (b) Cloud Internal IP 5 Flows (c) Cloud Internal IP 10 Flows

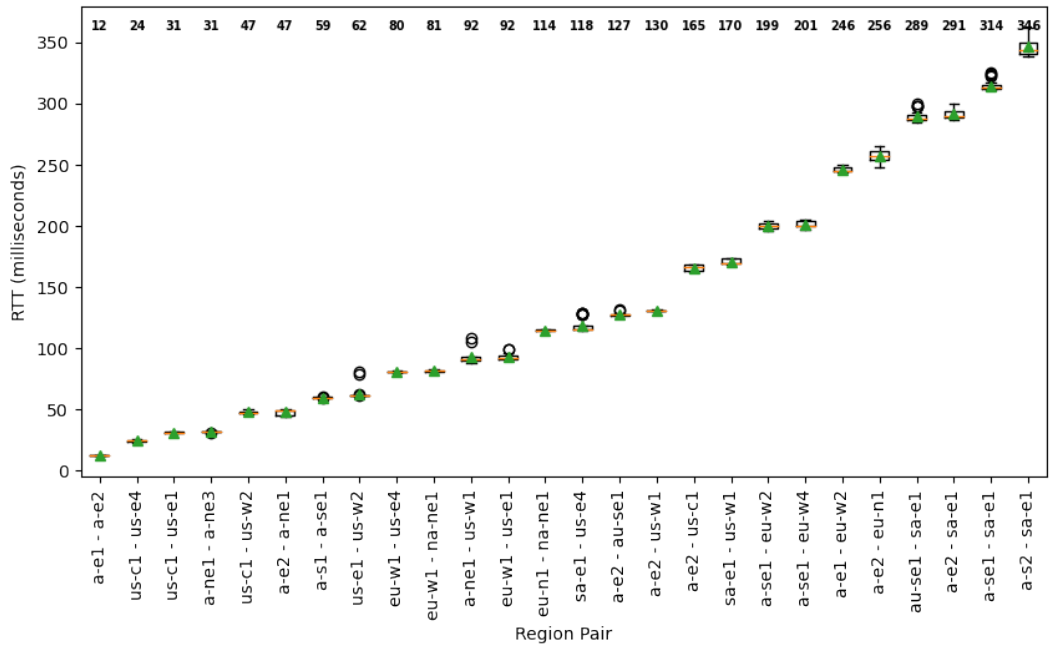
(d) Cloud External IP 1 Flow (e) Cloud External IP 5 Flows (f) Cloud External IP 10 Flows

(g) Emulated Network 1 Flow (h) Emulated Network 5 Flows (i) Emulated Network 10 Flows

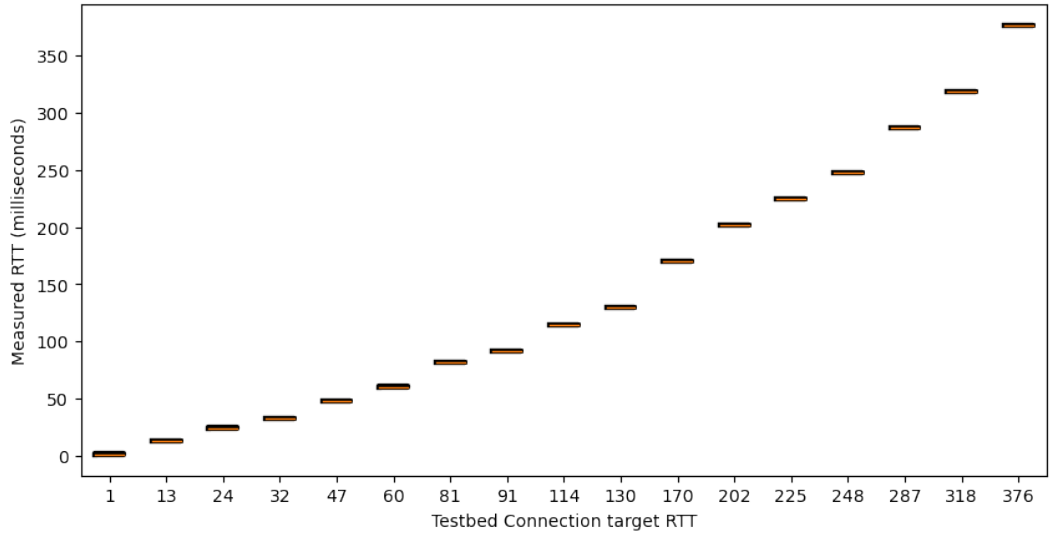
Figure 5.5: Throughput profiles of public cloud network using internal and external IPs for routing and their corresponding emulated testbed measurements for CUBIC.

5.4.1. Google Cloud Connections

In our tests on Google Cloud, we look at the throughput for two different available addressing options for inter-VM communication, Internal (Private) and External (Public) IP addresses. It should be noted that there is a lower bandwidth cap of 7 Gbps when using external IPs. There is more variance with the external IPs, especially with a single flow. There are multiple reasons why this could be the case. It could be a consequence of how the 7 Gbps cap is applied to the VM. It could also be because traffic is being routed over a different, more congested network path.



(a) Google Cloud connections



(b) testbed connections

Figure 5.6: RTT measurements in milliseconds.

In Fig. 5.6, we show the RTTs for the connections we test in Google Cloud and our testbed. The latency for both of these are very stable, and thus should not affect the results of our throughput tests, as might be the case if a connection took a significantly longer or shorter route during a particular test.

5.4.1.1. Outliers and Variance

On a few of the Google Cloud connections there is also a high level of variance between throughput measurements. One example is the Google Cloud throughput profile at a 91 ms RTT for 1 flow shown in Fig. 5.5a. Here, this particular connection that we chose underperforms more often than others. This could be for a number of reasons such as it being a particularly congested link.

There are also several outliers that appear significantly below the bulk of the samples. This again, is a consequence of operating on a shared network while trying to achieve the maximum capacity for throughput. These profiles are the aggregation of several measurements taken over the course of multiple weeks at different times of day and this shows that the state of the network is not always the same.

5.4.2. Testbed Network Profiles

For our emulated testbed throughput measurements there was very little variance and few outliers when using a 0% loss rate. This is to be expected because it is not a shared environment and there is no cross traffic. These tests should represent near the maximum achievable throughput on a 10 Gigabit network using these specific network settings. As we can see the emulated profiles are very close to the cloud throughput profiles with Internal IPs for 1, 5, and 10 parallel flows.

5.4.3. Parallel Flows

In Fig. 5.5, the charts on the left (a,d,g) show TCP CUBIC with a single flow, the charts in the middle use 5 parallel flows, and the charts on the right use 10 parallel flows. In all of our tests, using multiple parallel flows resulted in higher achieved throughput with less

variance than using just a single flow. However, there is an upper limit on the improvement seen, governed by the maximum available bandwidth of the link.

We can see that there is a significant different between a single flow and 5 flows, but relatively little different in the shape of the curve between 5 flows and 10 flows. This tells us that for these machines and connections, 5 flows is probably enough to fully saturate the link. Using more flows than needed will cause unnecessary overhead on the host in terms of CPU and memory usage. This is why it is imperative that we find the optimal number of parallel flows to use (highest throughput, lowest overhead) when performing large data transfers.

We found using multiple flows is beneficial for aggregate throughput regardless of the congestion control algorithm being used and is especially useful on high latency links. This is apparent in the shape of the throughput profiles when we are using a single flow vs multiple parallel flows. The profile is more convex with a single flow and more concave with multiple flows.

This can possibly be explained by the dynamics of TCP at high latencies and random losses. Loss-based congestion control algorithms interpret any loss as congestion and will decrease the congestion window accordingly. When the loss is not actually caused by congestion and was instead random, this can lead to a situation where we are not achieving our maximum throughput because the congestion window was reduced unnecessarily. For some TCP variants, this can affect connections with large RTTs with greater severity, as some congestion control algorithms operating on connections with large RTTs can have slower recovery time compared to small RTT connections because they increase window based on RTT or when ACKs are received. Unless they do something to compensate for this, like HTCP which adjusts the increase size based on RTT, recovery times can suffer.

We can correct this somewhat by using multiple parallel flows. Functionally, using parallel flows is equivalent to have a large 'virtual' maximum segment size (MSS) and can assist in speeding up recovery times for congestion control algorithms and gives us greater aggregate throughput [106] [107].

5.4.4. Congestion Control

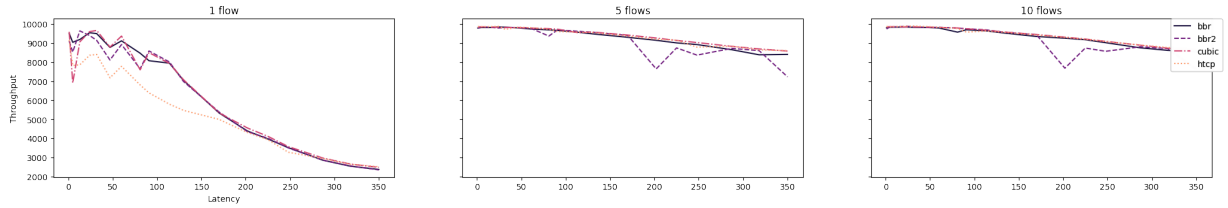


Figure 5.7: Throughput profiles for measurements on public internal cloud network for CUBIC, HTCP and BBR.

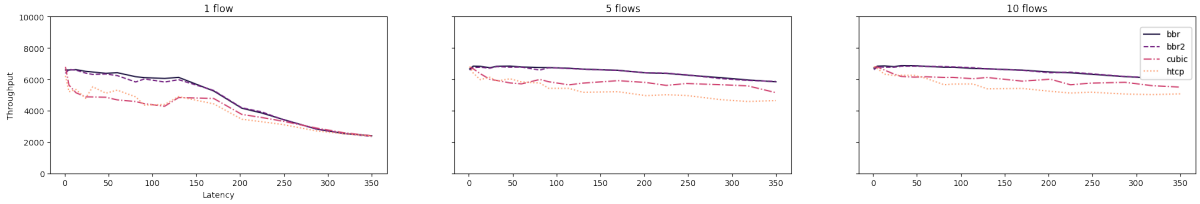


Figure 5.8: Throughput profiles for measurements on public external cloud network for CUBIC, HTCP, and BBR.

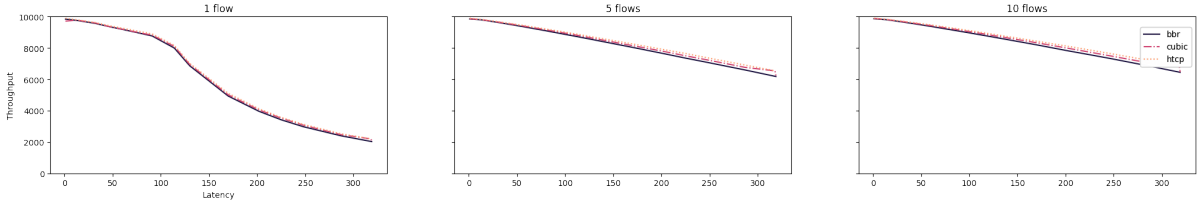


Figure 5.9: Throughput profiles for measurements on testbed for CUBIC, HTCP, and BBR with no error

As previously discussed, our tests use several different TCP congestion control algorithms to compare their effectiveness under different real and simulated circumstances. These include CUBIC, HTCP, BBR, and BBRv2 alpha. Each of these were designed with the purpose of improving performance for long, high bandwidth links and they all handle this task in a

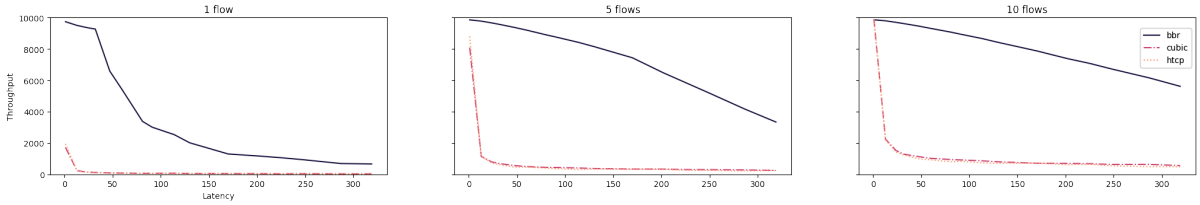


Figure 5.10: Throughput profiles for measurements on testbed for CUBIC, HTCP, and BBR with 1/1000 Error Rate

different manner. A more thorough description of each of these algorithms can be found in Chapter 3.1.3.

5.4.4.1. Comparison of Congestion Control Algorithms

In Fig. 5.7 we show the relative performance of these congestion control algorithms for Google Cloud using Internal IP addresses for 1, 5, and 10 parallel flows. We can compare these to the similar results in Fig. 5.8, Fig. 5.9, and Fig. 5.10 which show Google Cloud throughput using external IP addresses, the emulated network with no introduced loss, and the emulated network with 1 in 1000 error, respectively. In a production cloud environment with competing traffic and congestion we can see that generally BBR and BBRv2 perform better than the competition in terms of achieved throughput for a single flow with a low to medium RTT. This difference is diminished for higher RTTs. When using parallel flows, the TCP variants perform similarly, with the exception of BBRv2, which on some links shows somewhat lower throughput than the others. For external IPs on the cloud network, the BBR and BBRv2 show higher throughput than other TCP variants in most situations.

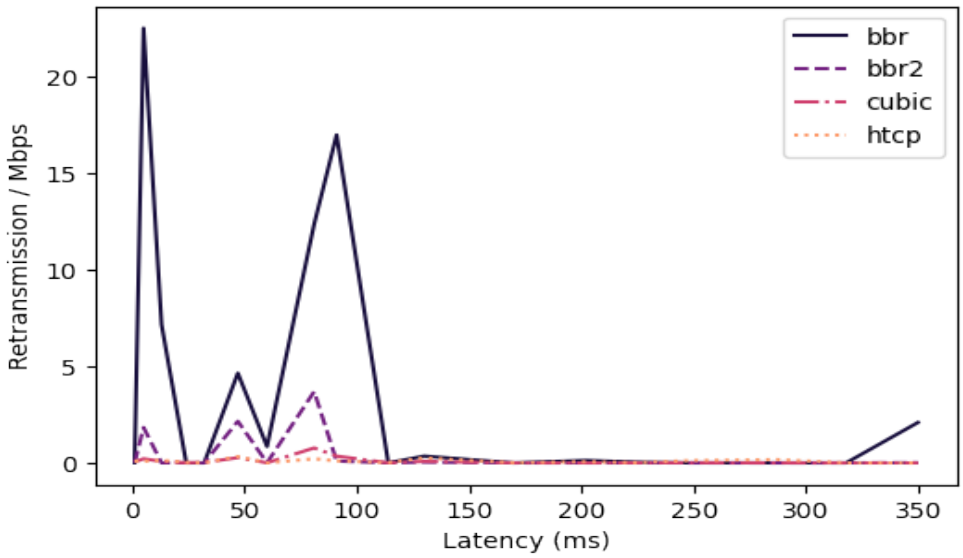
On our emulated testbed, there is almost no difference in performance between the congestion control algorithms when they are operating in a network with no loss and no cross traffic. This is the ideal situation and the curves we see here demonstrate the upper limits of throughput under these circumstances. The single flow shows a convex curve and both 5 and 10 parallel flows show concave curves.

When we introduce loss into the scenario as with Fig. 5.10, we can see that the curves become much more concave. HTCP and CUBIC are most effected by loss, only achieving close to 10 Gbps of throughput with 10 flows at 1 ms RTT. At higher latencies, throughput quickly drops to below 1 Gbps for 10 flows and below 100 Mbps for a single flow.

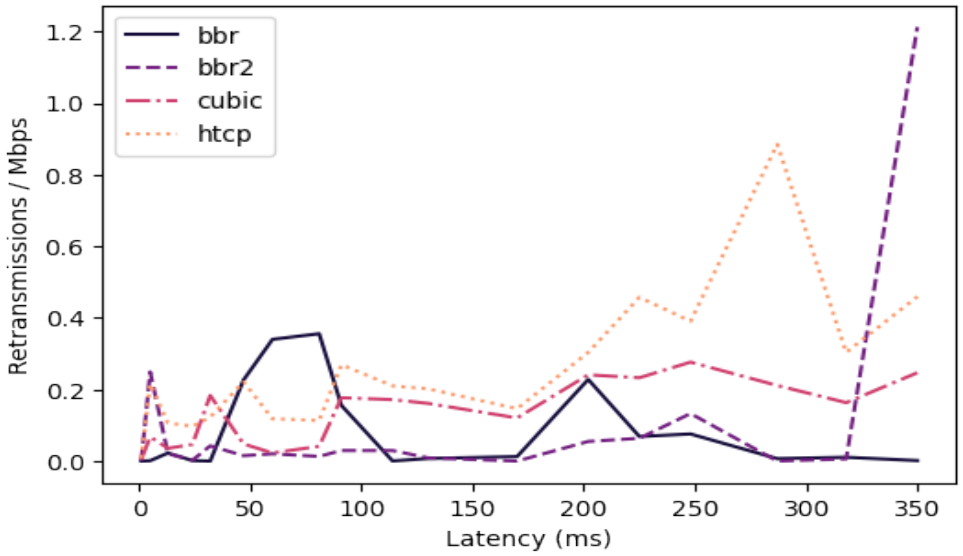
The only congestion control algorithm that holds up to these random network losses is BBR. With a single flow, the throughput profile is more concave than with no losses, but with 10 flows throughput is only decreased by around 1 Gbps even at an RTT of 350 ms. Again, this is largely because BBR does not base its congestion control calculations on loss. So when HTCP and CUBIC encounter these random instances of loss and decrease their congestion windows accordingly, BBR doesn't change and will only have to retransmit 1 in 1000 packets.

5.4.5. Retransmissions

In Fig. 5.11, we show the ratio of the average number of retransmissions for a test to the throughput in Mbps for that test. With 1 flow, BBR has an order of magnitude higher retransmission rate than the other congestion control algorithms for some RTTs. In particular, with an RTT of 5 ms, BBR experiences an average of retransmission ratio of 23 retransmissions per Mbps of throughput. For a 30 second test, this is an average of around 200,000 retransmissions. However, if when we look at the data, most of the tests for BBR with 5 ms RTT have 0 retransmissions, but 2 out of 10 tests have over 1,000,000 retransmissions. In these instances, throughput also drops dramatically from around 9,500 Mbps to 6,500 Mbps. This seems to be an instance of BBR overestimating the bottleneck bandwidth available. This is a known issue that sometimes occurs. Since BBR does not use loss/retransmissions as a control signal, an overestimation of the available bottleneck bandwidth can result in a much higher number of retransmissions as compared to a loss-based congestion control algorithms. This high number of retransmissions may have undesirable



(a) TCP Retransmissions vs Latency (Internal IPs, 1 Flow)



(b) TCP Retransmissions vs Latency (Internal IPs, 10 Flow)

Figure 5.11: Public cloud network with TCP retransmissions for each TCP variant

effects and cause unnecessary congestion for shared links. On this same chart, we can see that BBRv2 suffers from the same issue to a much reduced extent since BBRv2 uses packet loss as part of its calculation for congestion and packet pacing.

With 10 parallel flows, we actually see fewer aggregate retransmissions for BBR and most of the other variants are largely the same. Take note of the drastically different scales

for Fig. 5.11a and Fig. 5.11b. Here, BBRv2 experiences its highest retransmission ratio of 1.2 at an RTT of 350ms. Looking at the individual tests, we can see a high of 38,813 retransmissions in a test with a throughput of 7602 Mbps. This is about the same as the average throughput that we see at this RTT for BBRv2 (7639 Mbps).

BBRv2 also has much higher retransmisions than HTCP and CUBIC in some cases, but also has fewer than the original version of BBR in most cases. It does this while still maintaining the high throughput values with lower variance than BBR. BBRv2 uses packet loss as part of its calculation for congestion and packet pacing.

5.4.6. Emulated Loss

On our emulated test network, we are able to precisely test the capability of our network configurations when exposed to network loss. We used an error generator to create instances of loss in the transfers. We ran these tests using different error distributions and at several different rates of errors for each distribution. We tried Gaussian, uniform, Poisson, and periodic distributions and also having no introduced errors. Our measurements showed very little difference in effect between different error distributions. However, the error rate has a significant effect on throughput and this is effect is the same for similar error rates no matter which distribution we are using.

5.4.6.1. Loss rate

The loss rates used range from 0 induced loss to every 1/100 packets. A visualization of these results can be seen in Fig. 5.12.

Generally, higher rates of loss make the throughput profiles for a specific network configuration more convex. This can be seen most dramatically when using a single flow. When the loss rate is high, 1/1000, throughput drops precipitously even at low latencies. This

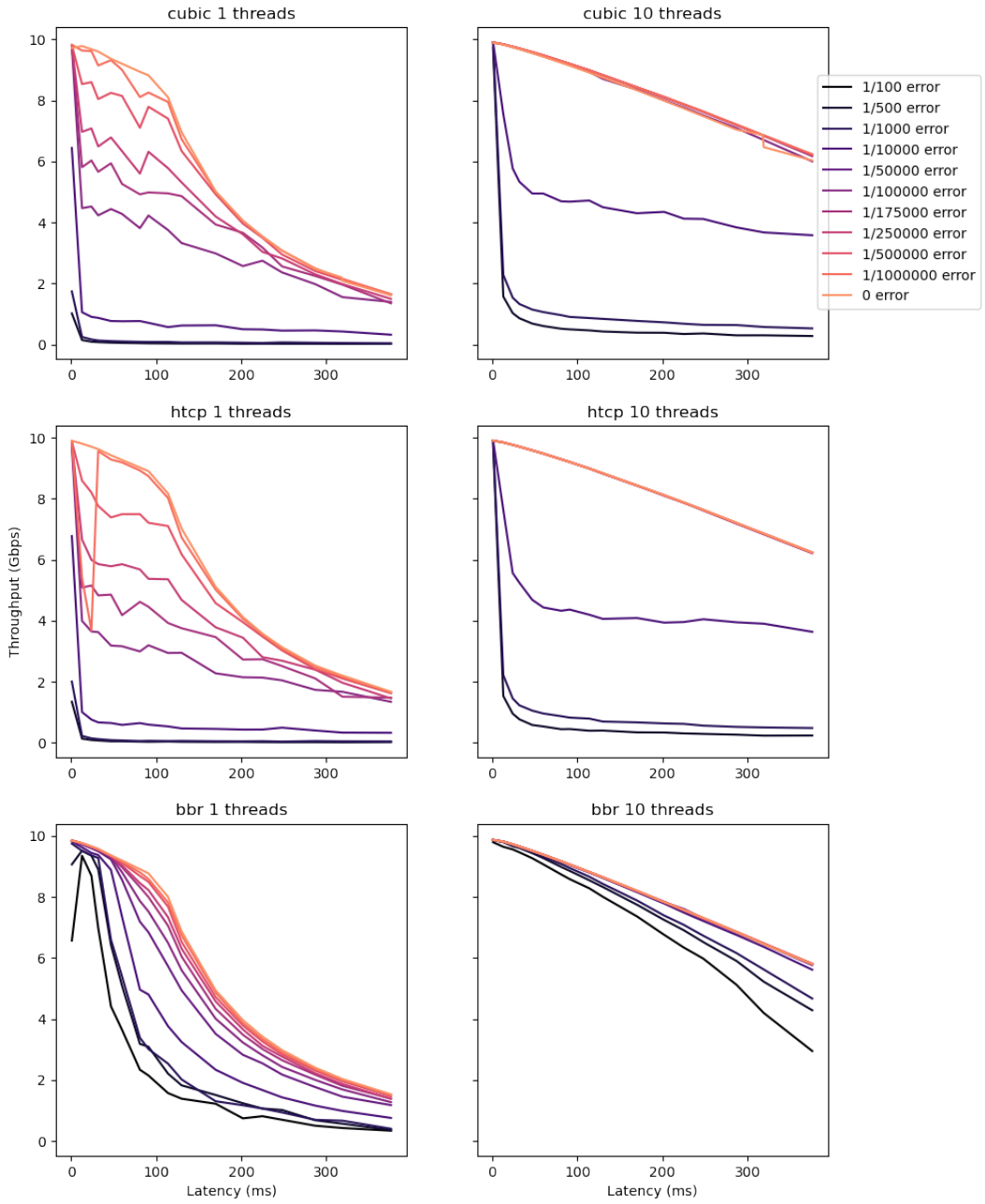


Figure 5.12: Effect of loss rate on throughput.

effect is offset somewhat by the addition of additional sending threads, which each have the same loss rate.

The behavior under loss also depends greatly upon the congestion control algorithm being used. Again, we see a marked improvement when using BBR. While throughput for

HTCP and CUBIC under high loss conditions suffers at low latencies, BBR manages to degrade more gracefully. Even with 0.01% loss, the throughput profile remains concave. If we compare HTCP, CUBIC, and BBR with 10 parallel threads at 350 ms RTT and 0.002% loss, the throughput for HTCP is 0.238 Gbps, CUBIC is 0.280 Gbps, and BBR is 4.283 Gbps. This is over a 1000% increase for high latency, high loss throughput.

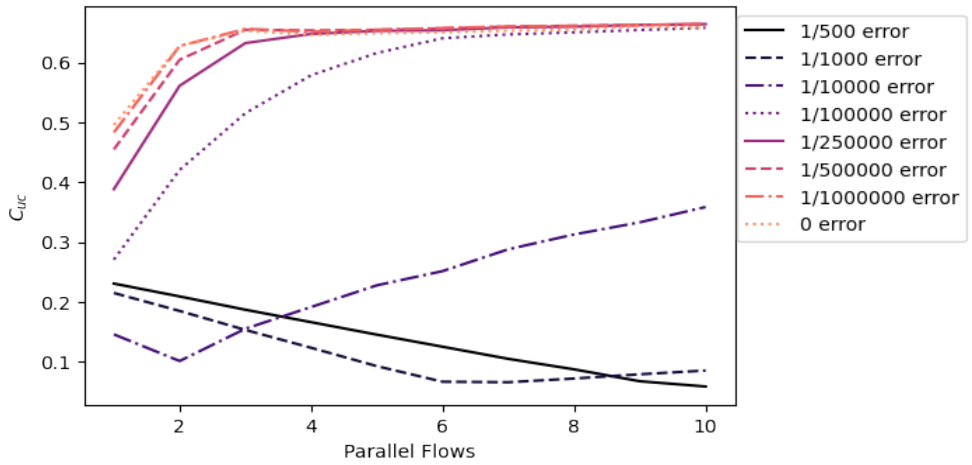
5.4.7. Coefficients: Ideal and Loss Conditions

We calculated the utilization-concavity coefficient, C_{UC}, described in Section 5.2.1 for each of our throughput profiles both on Google Cloud and on our emulated network. This will enable us to more easily compare between the throughput performance of these networks under a wide variety of circumstances.

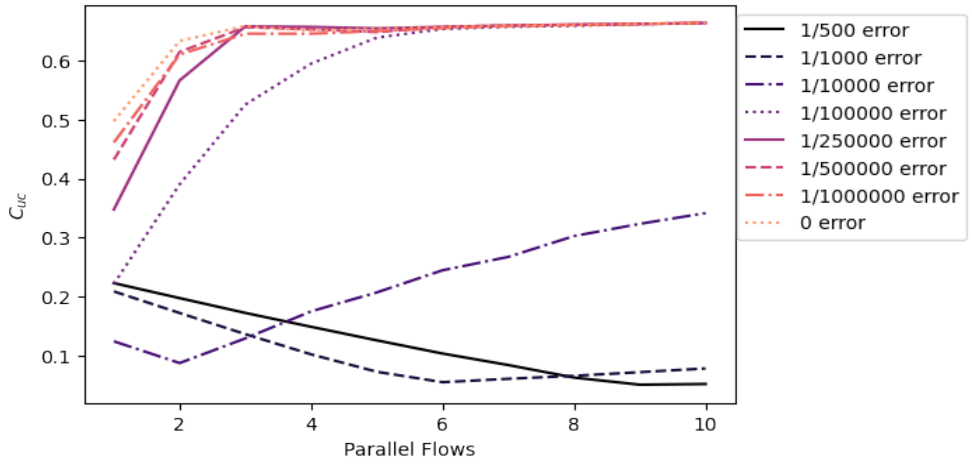
In Fig. 5.13, the C_{UC} is shown for different loss rates and numbers of streams for CUBIC, HTCP, and BBR in our emulated 10-Gigabit Ethernet (10 GigE) network. As we have seen in the previous throughput profiles, all of these congestion control algorithms perform well when there is little or no loss on the network and reach an optimal level around 3-4 parallel flows. For 5 flows and beyond, we see no additional benefit for the overhead of adding additional flows.

With CUBIC and HTCP in Fig. 5.13a and Fig. 5.13b, when there is 1 in 10,000 loss, the C_{UC} is low at 1 and 2 parallel flows, meaning a very convex throughput profile and low throughput and utilization overall. This is partially alleviated by adding additional flows and with 10 flows we see a C_{UC} of 0.3. When the level of loss is greater than this, CUBIC and HTCP are unable to recover even with additional flows. Throughput quickly drops to near zero when the RTT is above 1ms and this is reflected with a C_{UC} of less than 0.1.

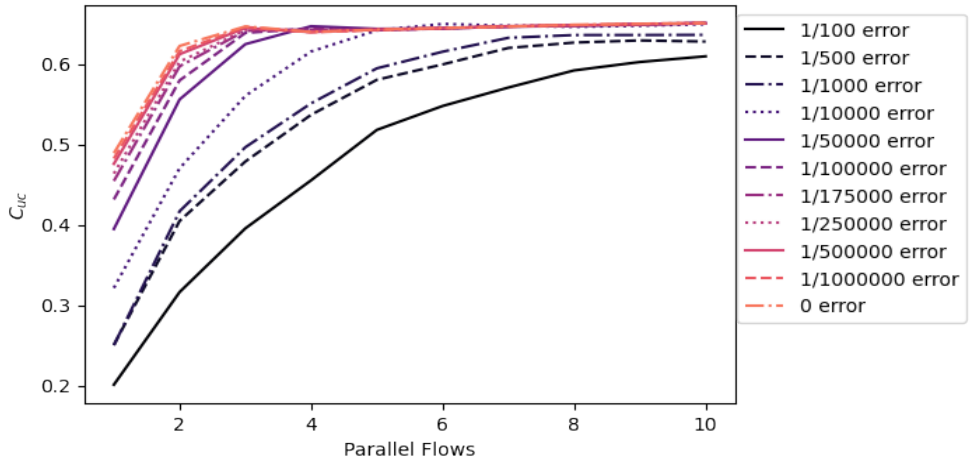
In Fig. 5.13c, we show that BBR maintains a relatively high C_{UC} despite high loss rate. The additional of more flows also seems to help more significantly than with either CUBIC



(a) C_{UC} vs parallel flows in emulated network with CUBIC



(b) C_{UC} vs parallel flows in emulated network with HTCP



(c) C_{UC} vs parallel flows in emulated network with BBR

Figure 5.13: Emulated network C_{UC} vs parallel flows

or HTCP. For each of those congestion control algorithms the C_{UC} with a loss rate of 1/500 and 10 parallel threads is less than 0.1. For BBR in the same situation, the C_{UC} is 0.61.

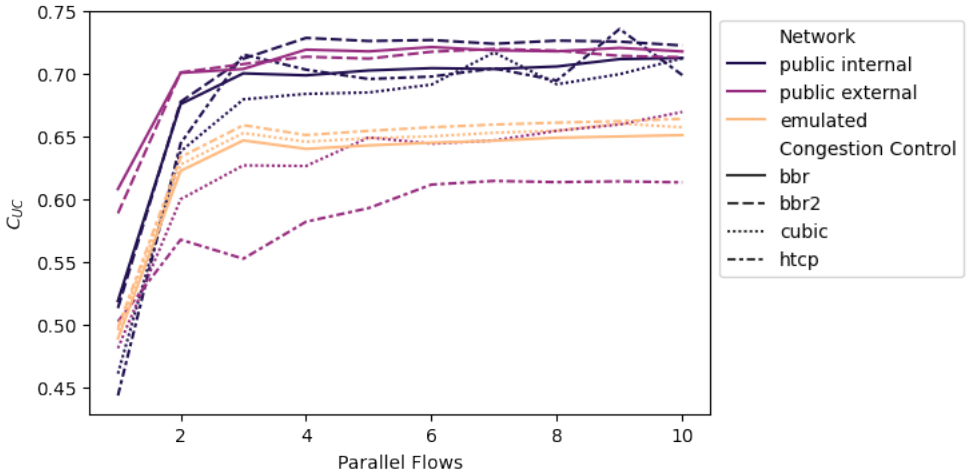


Figure 5.14: Comparison of C_{UC} of different TCP variants on emulated (with no error) and public cloud network

In Fig. 5.14, we show the C_{UC} for both the emulated connections with no error and the Google Cloud connections both using external and internal IP addressing. For external IP addressing, the C_{UC} is calculated with a maximum bandwidth of 7 Gbps, because the lower bandwidth cap does apply.

Interestingly, the public cloud connections actually have somewhat higher C_{UC} values compared to their emulated counterparts. We can confirm this in Fig. 5.5. The throughput measurements for the public cloud connections stay closer to their 10 Gbps maximum than the emulated networks at high RTTs. There are several reasons that this could be the case. One is that many of the links on the cloud network are likely higher bandwidth than what we are allocated, so we may be able to send at a faster rate for this artificial bandwidth cap than we could for a physical bandwidth cap. It could also be because they have slightly different kernel versions.

5.5. Conclusions and Future Work

Cloud infrastructures support dynamic provisioning of transport networks, which are possible alternatives to dedicated networks in a broad class of scenarios. We studied throughput performance of such networks over a public cloud environment using an emulated, dedicated testbed network with a similar range of RTTs and endpoint settings. We found that measurements and analytics of these two networks to be comparable in terms of throughput profiles and dynamics, despite latter being dedicated and emulated. The cloud network showed more variance in throughput values, most likely attributable to cross traffic on busy connections and hosts, which is not experienced in the emulated network. For all these networks, we found the most reliable way to increase throughput is to use multiple parallel flows. We also found that generally BBR had the best performance on this type of network across all RTTs. On the emulated network we also tested different levels of induced packet losses. We found that a high error rate has significant effect on throughput and this is only somewhat mitigated by using more parallel flows. Thus, the cloud networks are a more flexible, cost-effective alternative to dedicated network infrastructures in cases where their footprint is conducive.

In continuation of this work, we plan to use random loss distributions in the testbed network to see if a loss distribution other than periodic will change the throughput profiles for any TCP variant. Future research directions include a more detailed study of the cases where cloud connections have lower throughput and higher retransmissions; the possible causes include how VMs are orchestrated on cloud sites that typically serve as data centers, including their migrations and overall allocation. It would be of future interest to develop analytical models that explicitly account for the details of cloud networks, specifically, the under-performance of some connections.

We compared the throughput performance from VM to VM connections across Google Cloud to connections on a testbed with emulated latency and loss. We ran bulk-throughput

benchmarks across a wide range of RTTs on both these platforms and repeated the measurements using different congestion control algorithms and numbers of parallel TCP flows. We examined the differences that these parameters had on the measured throughput of the network and compared these configurations using their throughput profiles and their related utilization-concavity coefficients (C_{UC}. We found the the throughput profiles for the cloud network and the emulated network to be overall similar in shape and reaction to parameter changes.

They are similar enough that it could be possible to determine how a cloud network will react to certain parameter changes and network settings by first emulating that network. This could potentially save money as running tests on an emulated system to determine its characteristics is cheaper than running those tests on a production network in the cloud. Then we can use emulated to estimate physical. If we want to more closely match the emulated network to the physical network, we can pick a similar loss rate.

Our study also shows that BBR tends to respond much better to network losses than CUBIC or HTCP.

Chapter 6

Performance Forecasting and Anomaly Detection in Cloud Networks

6.1. Introduction

Now that we have extensively examined how to collect accurate network measurements in Chapter 3 and how to batch schedule a large amount of tests in Chapter 4.3, we can use these continuous and automated measurements to help understand how our network functions and the performance it will be capable of both now and in the future. As we have previously discussed, cloud and cloud-connected networks are continuing to expand to meet the growing demand as organizations offload the burden of managing physical infrastructure to various cloud providers. As this happens, it becomes essential that these organizations understand the network dynamics of their cloud infrastructure and are able to estimate or forecast what type of performance they will be able to achieve at any arbitrary time point in the future. Equally important is the ability of organizations to identify when their network is performing outside of its usual bounds. In this chapter, we will explore how we can use statistical and machine learning models to forecast future throughput and detect anomalous performance.

6.1.1. Throughput Prediction

Obviously the farther into the future we can forecast the better, but here there is often a trade-off with accuracy. The further our prediction is from the present, likely the less accurate it will be. This being the case, here we will focus on predicting 1 time-step into the future. For us, that on the granularity of days, as the dataset we are using has measurements taken every day. The models and ideas can easily be extended for multi-day predictions or

applied to a different timescale, such as minutes or seconds, provided they are retrained with appropriate datasets.

Forecasts such as these are useful in a diverse set of scenarios including, but not limited to, assisting with cloud orchestration of computations across highly distributed resources, high performance file transfers, resource allocation and management, and helping diagnose network issues such as performance bottlenecks. Accurate estimates can also help with cloud routing decisions [108] and be used to improve the performance of applications that use a datastream by dynamically adjusting the bitrate of the stream to better suit the current and predicted throughput.

When operating in a cloud environment, unless you have opted to get dedicated machines and network links which are available at a premium, the network connections and even the physical NIC you are using will be multi-tenant and have some amount of competing traffic. This could affect the throughput you are able to achieve across a link and can lead to lower than expected throughput and increased variability in network measurements. There are also situations or times where taking an active measurement of the network is not always the best option, such as when there is very limited bandwidth. Being able to rely on past data to estimate the throughput in the immediate future can help remedy this situation and reduce the strain on an already stressed network.

6.1.2. Performance Anomaly Detection

Another important aspect of cloud network analytics is the ability to detect anomalies and outliers. Both cloud providers and customers need to detect anomalies early enough that there are not any sustained service interruptions. For this reason, real-time, robust anomaly detection is essential. This depends upon having consistent historical data to train from and a model that will report anomalies while limiting false positives.

There are several different methods that can be used for anomaly detection including clustering and classification. In both of these we essentially try to create groupings that represent our normal data and then see how well each new data point fits into the groups. If it does not, then it can be considered anomalous. Another method of anomaly detection that can be applied to time series data is called predictability modeling. For this, we create a model that uses historic data to try to predict the next result. We use the error between the predicted result and the actual result to determine if the data point is possibly anomalous. This relies on the assumption that normal data points are more easily predictable than anomalies. In this way, we can easily adapt the prediction models that we create to anomaly detection models. There are a few differences in training and data preparation that will be discussed, but the architecture of the models stays largely the same.

6.1.3. Motivations

If we have a complete view of all aspects of a network, we could use all of the routing, traffic, and congestion information on all the routers to determine the throughput we are likely to be able to achieve on each specific route, such as with the control layer in a Software Defined Network [9]. However, if we have a more limited view of the network, it can be more difficult to predict throughput. Here, we have a scenario where we only have access to the information that the two endpoints of a network benchmark have access to. This is the typical situation when operating as a customer in a cloud environment.

Specifically, our dataset for these experiments consists of TCP memory transfer measurements using iPerf and Netperf, corresponding latency measurements, and metadata about the endpoints. This metadata includes the TCP window size, Maximum Transmission Unit (MTU), congestion control algorithm or TCP variant used, the TCP stack, kernel version, and any artificial limiters on the bandwidth enforced by the cloud provider that we know about through their public documentation. We have the ability to change the TCP send

and receive buffers for all endpoints as well as the congestion control algorithm. The MTU is set to 1460 for all of these experiments for each endpoint and in the virtual network that we have control over, however we do not know what the MTU is set at for any intermediate hardware. We have little to no information about the state of any network infrastructure such as gateways, switches, or routers, and we do not know and cannot control the amount of cross traffic on the network or its behavior. We can also gather packet loss and retransmission statistics, as those have an effect on throughput and can be indicative of congestion or problems across a network link.

We use this dataset to compare different methods to accurately predict throughput for connections in this network infrastructure that do not explicitly model the time series of individual pair-wise connections based on past throughput and latency measurements and associated metadata. We study classical, machine learning and neural network models to predict TCP throughput on our dataset of which the LSTNet [71] architecture achieves overall lowest error rate. These models, although trained specifically on these datasets, should be able to be retrained with data from other networks, or generalized to fit a wider scope.

In the remainder of this chapter, we first describe our data collection method and details of datasets in Section 6.2. Next, we describe our approach to this problem and justification for the models chosen to test in Section 6.3. Then we present the results of predictions using several models and compare their performance in Section 6.4. Finally, we outline out future research directions direction in Section 6.6. For a discussion on similar research in time series analysis and forecasting, refer to Chapter 2, Sections 2.3 and 2.5.

6.2. Data Collection and Analysis

The dataset that we use for training and evaluation in this chapter and throughout much of this thesis originates from our cloud benchmarking project with Google Cloud. As

part of our research for that project, we run a large set of mainly network benchmarks between virtual machines in different cloud regions and track their performance over time. The dataset used in this chapter is a subset of those measurements consisting of inter-region throughput an latency measurements on a specific machine type collected between June 2019 and August 2020. In addition to throughput and latency, we also included in the dataset the OS version, kernel version, cpu specs, rmem/wmem max, TCP send/receive buffer max, and cpu usage statistics. All of the endpoints have the same cpu spec, memory spec, and operating system (Ubuntu 18.04). If these specs were more variable from test to test, these parameters would likely also be included as features in our models.

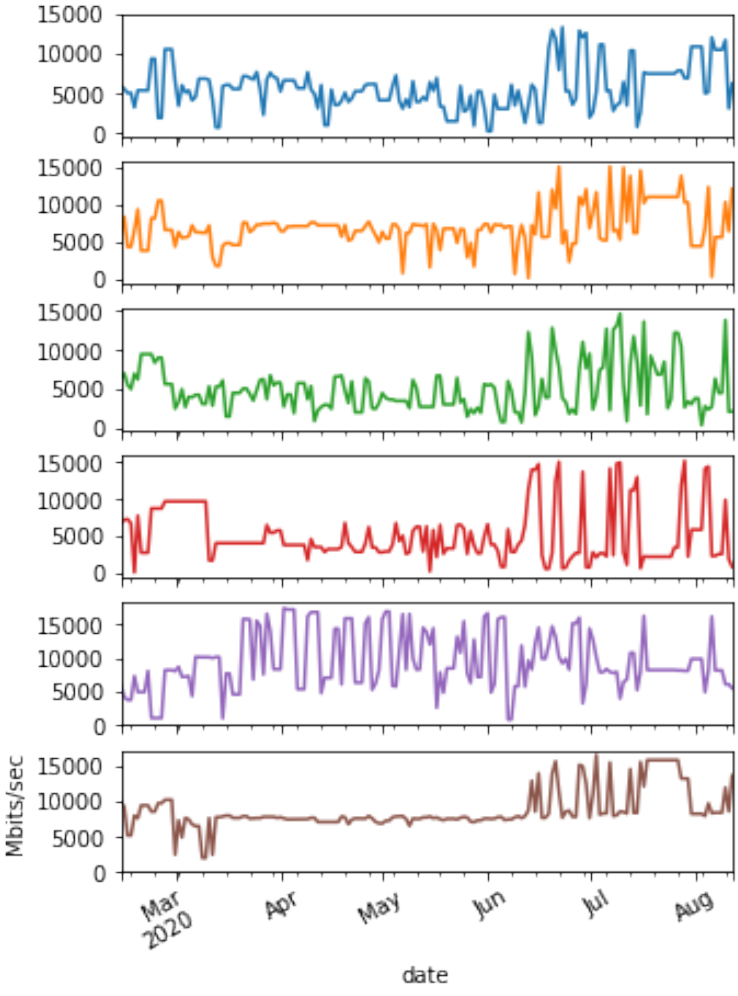


Figure 6.1: Throughput measurements for randomly selected connections in our dataset from mid February to mid August 2020.

The scope of these measurements are inter-region measurements between all unique pairs of cloud regions in Google Cloud as well as intra-region measurements from VMs situated in the same region. In this context, a region usually refers to a cloud provider's datacenter or grouping of datacenters situated in a specific geographic region. Each provider has slightly different terminology, but here we use 'region' or 'cloud region' to refer to a datacenter for a cloud provider. These regions are further divided into 'zones' which constitute individual failure domains within a region. Google Cloud specifically has 28 regions at the time of writing, representing datacenters in distributed locations from Virginia to Tokyo. A specific zone or region is a required argument when creating virtual machines.

By creating pairs of virtual machines in different zones (or the same zone) and running experiments between them we can benchmark the network performance of each link. Because we are benchmarking network performance between virtual machines in many cloud zones, we end up with 136 distinct zone pairs. For the dataset used in this chapter, we ran daily tests between all unique pairs of regions for 17 regions, that gives us 136 region pairs that we have data from. If we include the intra-region tests that brings it to 153 unique routes that we took daily measurements for over the course of 6 months. One could also consider the daily measurements for each route to represent its own time series, which gives us 153 distinct time series over the test period. Not all of these time series are of the same length, as some virtual machine pairs were added to the test setup later than others.

6.2.1. Analysis of Data Features

The first step for us to create an accurate prediction model for network throughput was to figure out which of our gathered metrics correlated most strongly with TCP throughput. Here, when we mention TCP throughput, we are specifically referring to single-stream and multi-stream throughput gathered via iPerf 2.0.13, which we discussed extensively in subsection 2.1.1.1. We also consider multi-stream and single stream throughput separately.

Our multi-stream throughput tests use 32 parallel sending threads to try to maximize total throughput. The throughput profiles of multi-stream and single stream Iperf are different enough that separate models for predicting each is helpful. The correlation of Iperf single stream throughput 1 time step into the future to each of these variables is detailed in Table 6.1. In other words, for throughput, latency, and max TCP buffer size at time step t, we want to find correlations with throughput at time step $t + 1$.

Table 6.1: Correlation of variables at 1 time step ahead single stream Iperf throughput

Correlation with t+1 single stream Iperf throughput	
Variables	**Correlation**
Iperf_throughput (1 stream) (t+1)	1.000
ping_average_latency	-0.304
ping_min_latency	-0.304
ping_max_latency	-0.297
TCP_max_receive_buffer	-0.164
Iperf_throughput (1 stream)	0.677

For our throughput prediction models, in addition to historical throughput, we have chosen the following metrics: ping average latency (RTT), and the TCP max send and receive buffer size. For time series prediction this means that the inputs to our model are the past n time steps of throughput, latency, and TCP buffer size data.

Table 6.1 also includes ping min latency and ping max latency that are fairly well correlated with 1 step ahead throughput. However, we found including them brings little to no value to our prediction models as they are much more closely correlated to ping average latency, which is already in our models.

We found average latency to have a relatively high degree of negative correlation with single stream throughput; higher latency means lower throughput. This is logically consistent with our own experience with throughput testing as well as many of the formula based

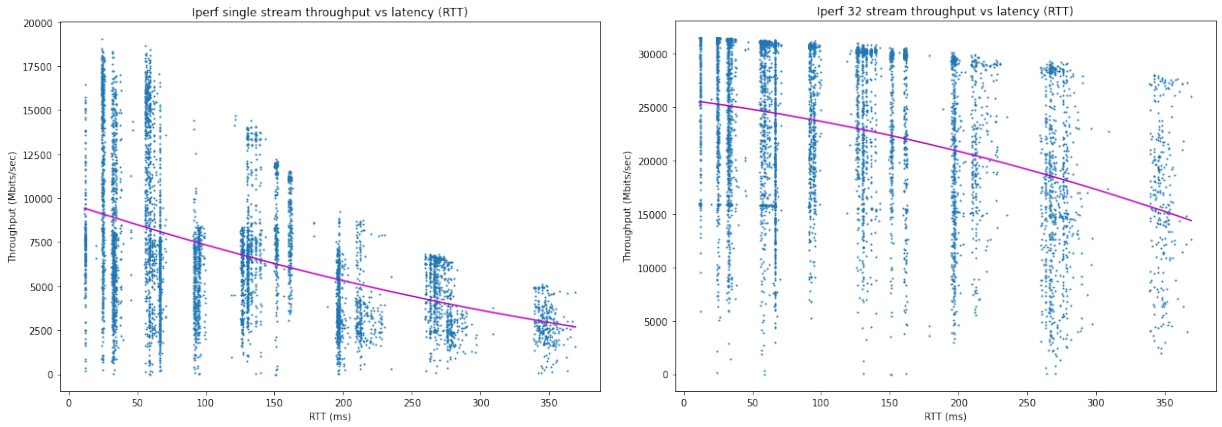


Figure 6.2: Scatter plot of throughput and latency from all points in our dataset. A second degree polynomial line of best fit shows a slightly convex curve.

models for throughput prediction [58]. Longer latencies reduce throughput especially for single stream workloads. The effect of latency on throughput can be reduced by employing multiple streams. We can see both of these effects in Figure 6.2. These charts plot latency vs single stream and latency vs 32 stream throughput. As can be seen in the first chart, single stream throughput and latency form a convex curve, as throughput decreases with higher latencies. Whereas 32 stream throughput vs latency forms a concave curve, as most throughput measurements are limited by an artificial cap at 32 Gbps. In these charts you can see that throughput also has a high variance, even at similar latencies. This is probably due to cross traffic.

Some of our tests used a max TCP buffer size that is larger than the default, so we found it necessary to train on this variable as well. Generally, larger TCP buffer sizes can help achieve higher throughput especially when dealing with higher latencies [109]. Because of this, tests on virtual machine pairs with higher latencies are run using appropriately higher TCP send/receive buffer sizes. This also means some of our connections with very low latencies are running with a lower (default) TCP buffer size while still achieving a greater throughput. This could lead to a situation where a spurious correlation is found between buffer size and throughput, in which a model learns that lower buffer sizes lead to higher

throughput values because that is what our raw data shows. Including the max TCP receive buffer size can help the model account for two measurements with similar latency values but significantly different throughput values. In the future, it would help to run tests with a wider variety of max TCP send/receive buffer sizes.

6.3. Prediction Models

In order to achieve the highest accuracy possible for our prediction, we trained and evaluated eight different models on our dataset. The general architectures for each of these models are listed in Table 6.2. For each model, we tried to optimize the architecture as much as possible using a grid search for their respective hyper-parameters. We tested both classic time series forecasting algorithms such as ARIMA, regressions, as well as multiple neural network architectures that have been used for time series prediction. For most of our models, we used a 10 day window size for our input to forecast 1 day ahead.

For ARIMA and VARIMAX, we modeled and evaluated each of the time series separately and chose the optimally performing p,q for each time series. The error estimation in Table 6.3 is an average of the resulting errors for running ARIMA on each separate time series in our dataset after have done a grid search for each time series to find its optimal parameters.

While we think that formula based methods are useful, they are largely suitable for the prediction of throughput on very short time scales (milliseconds to seconds) and are based on the patterns and behavior of TCP congestion control algorithms like Reno and Cubic [22]. So, overall, we have found these to be not particularly applicable to our use case of day to day throughput prediction, but could be very informative when creating a model for short term prediction.

We separately trained and evaluated these models for predicting both single stream throughput and 32 stream throughput using the features we previously determined. We also trained and evaluated these models to predict throughput based on only historical

Table 6.2: Table Detailing the architectures of each tested model

Model Architectures	
Model	Architecture
ARIMA	AR window = 10 MA window = 10
MLP	Dense Layer Dense Layer with dropout Dense layer
Simple LSTM	LSTM Layer with recurrent dropout Dense layer
Simple GRU	GRU Layer with recurrent dropout Dense layer
CNN	Conv1D MaxPool1D Conv1D MaxPool1D Conv1D MaxPool1D Dense Layer (output)
CNN + GRU	Conv2D Dropout Reshape GRU Dropout Flatten Dense
LSTNet	**Branch 1** Input Layer Reshape Layer Dense Layer Conv2D Layer Dropout Layer Reshape Layer GRU Layer **Branch 2 (in parallel)** pre_AR Layer Dropout Layer Dense Layer Dense Layer **(Combine Branch1 and Branch2)** Post_AR Layer Add Layer

throughput data to serve as a comparison and see if our additional features actually helped improve the accuracy of the models. Each of these models was trained using the first 60% of the dataset. An additional 20% of the dataset was reserved for validation and hyper-parameter tuning and the final 20% is used as our test dataset to evaluate the performance of the models.

Throughput is not only limited by distance, congestion, and other real factors. On many networks there are artificial limits on bandwidth, often depending upon the level of service enrolled. This can lead to lower actual throughput than predicted if this limit is not accounted for. Artificial limits were active for the cloud provider on which we ran our tests and created our dataset. This means that our models are trained to predict throughput of machines on this network, but likely would need to be retrained for a network with different characteristics. Alternatively, this could be worked into the model by adding maximum available bandwidth as a feature.

6.3.1. Error Metric To measure the performance of the models against our dataset we have chosen to use the Mean Absolute Percentage Error (MAPE). MAPE measures how far off predictions are from the true value on average as a percentage. It is formulated as:

$$\text{MAPE} = \frac{100}{n} * \sum(\frac{|y - \hat{y}|}{|y|}) \tag{6.1}$$

Other common error metrics we have seen in similar work include Mean Absolute Error (MAE), Mean squared error (MSE), and Root mean squared error (RMSE). We chose MAPE because throughput across various paths in our dataset exhibit a very wide range of possible values: from 10 Mbits/sec to around 32 Gbits/sec. MAPE should allow us to judge the accuracy of the models we are testing without a small number of relatively large errors on high throughput routes significantly affecting our overall error score, as they might with

other error metrics. However, if we were more concerned about errors for larger throughput links than smaller throughput links, then RMSE might be a better alternative.

6.4. Throughput Prediction Results

In this section, we share our results from the models that we used to predict on our test dataset. The test dataset is made up of the last 20% of data from our total dataset. The mean absolute percentage error (MAPE) as well as the Correlation Coefficient (CORR) of each model on our test dataset is shown in Tables 6.3, 6.5, and 6.7,. This table shows both: a) results when predicting single stream throughput and 32 stream throughput based on multivariate historical data (throughput, latency, and TCP max send/receive buffer) as well as: b) results for single stream throughput prediction based on only historical throughput in order to compare between univariate and multivariate models.

The lowest error that we were able to achieve for single stream 1-step prediction was with LSTNet, a hybrid neural network architecture, followed by a CNN + GRU architecture. For multi-stream throughput, the results of most of the neural network architectures were much closer.

All of the neural network architectures were shown to be able to perform better than ARIMA for this use case. This points to there being non-linearities in our dataset that a linear model like ARIMA cannot adequately model. ARIMA had both the lowest correlation and highest MAPE by a significant margin for predicting both single stream throughput and multi-stream throughput.

All of the univariate models had higher MAPEs than their multivariate counterparts, leading us to draw the conclusion that the addition of latency and TCP max send/receive buffer size did generally help improve the accuracy of these models.

Table 6.3: Result Error and Correlation metrics for all models for multivariate iPerf 1 stream throughput

predicted variable	**multivar iPerf 1 stream throughput**	
Model	**CORR**	**MAPE**
ARIMA	0.8965	520.4332%
MLP	0.9341	34.0527%
LSTM	0.9339	32.9195%
GRU	0.9340	39.0893%
CNN	0.9239	39.9573%
CNN + GRU	0.9345	34.4534%
LSTNet	0.9267	**30.6659**%

Table 6.5: Result Error and Correlation metrics for all models for multivariate iPerf 32 stream throughput

predicted variable	**multivar iPerf 32 stream throughput**	
Model	**CORR**	**MAPE**
ARIMA	0.7894	119.1619%
MLP	0.8653	30.3297%
LSTM	0.8780	32.5345%
GRU	0.8790	31.1101%
CNN	0.8709	32.0909%
CNN + GRU	0.8750	31.9466%
LSTNet	0.8604	**29.3034**%

Table 6.7: Result Error and Correlation metrics for all models for univariate iPerf 1 stream throughput

predicted variable	**univariate iPerf 1 stream throughput**	
Model	**CORR**	**MAPE**
ARIMA	0.8890	550.2305%
MLP	0.9335	50.6598%
LSTM	0.9330	73.5089%
GRU	0.9341	41.8926%
CNN	0.8955	40.7733%
CNN + GRU	0.9344	**36.9240**%
LSTNet	0.9292	39.6644%

All models showed a lower MAPE when predicting iPerf 32 stream throughput, but also a lower correlation. The lower MAPE values show that predicting multi-stream throughput is more accurate. The lower correlation for multi-stream could possibly be explained by the 32 Gbps throughput ceiling imposed by our infrastructure, as well as the relatively high variance we see with this high bandwidth, as discussed in Section III and can be seen in Figure 6.2.

The performance of these models shows that throughput, even at a daily level, is predictable with the right model. In Figure 6.3, we can see the 1-step predicted throughput vs actual throughput (single stream) for each of the models we tested. The actual throughput shows a significant amount of variance, but every model is able to approximate the overall trend in throughput. The fact that our error decreased with more complex architectures shows that there is potential to create or find a model that predicts this with even greater accuracy. However, we realize that there is likely an upper limit on the accuracy we are able to achieve given the limited view of the network we have, the time scale we are working with, and some amount of inherent unpredictability or randomness in network throughput.

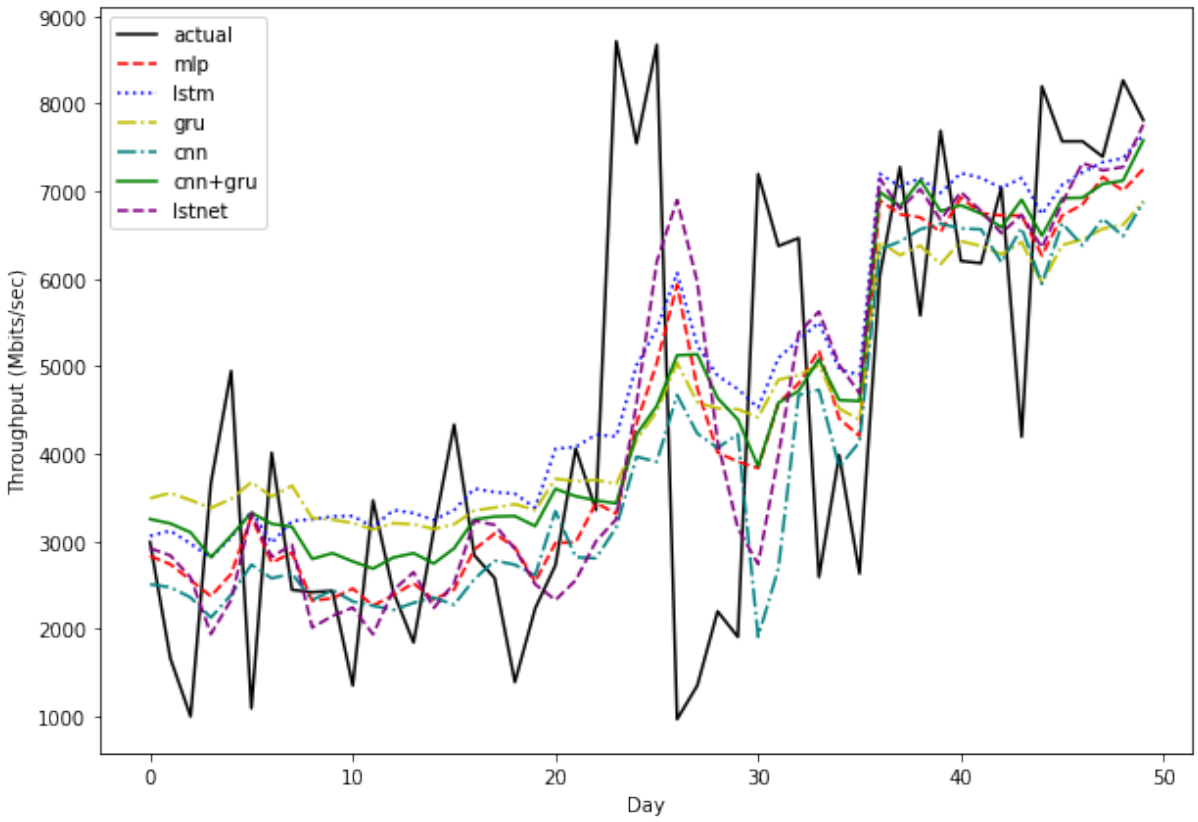


Figure 6.3: Comparison of predicted vs actual throughput for various models on a 50 day sample of test data

Even with an average of 30% error on the best performing models, these predictions can still be useful for traffic engineering, scheduling large transfers, or diagnosing network problems.

6.5. Anomaly Detection

Now that we've shown these models predictive capabilities, let us discuss how they can applied to the task of anomaly detection.

Currently for our ongoing cloud benchmarking efforts, we have implemented a simple anomaly detection based system on whether a value is significantly above or below the moving average of the last 10 daily measurements. The moving average is calculated and stored in a table in a nightly scheduled SQL query. A moving average kept for each unique

connection that we run benchmarks for. So tests from region us-east-1 to region us-central-1 will have their own moving average separate from tests between another pair of regions. Then when we run new tests, for every new data point entered into the database, we compare that point to the stored moving average using a Google Cloud Function. If a new data point is outside of the threshold, an alert is generated. These alerts are quite versatile and can be pushed to email, SMS, productivity tools such as Slack, etc. While our current system does work well for large, single point anomalies, we would like to create a more comprehensive system that functions for smaller deviations and increased instances of variance that we would still consider anomalous.

Our plan for implementing an improved anomaly detection system for cloud performance is to explore several general model architectures including both ones we have covered here and some novel deep learning architectures that we have not seen applied to this problem. We will take our cloud network performance data set, which tends to have a high amount of throughput variability, and divide it into testing and training time series sets much like we did with throughput prediction. Since it is a sizeable data set, it will primarily limit us to using unsupervised machine learning algorithms, as the anomalies present are not currently labeled.

6.5.1. Data Preparation

To be able to evaluate how well our anomaly detection models are performing, we must define what an anomaly is for our dataset. This is a major challenge in developing unsupervised anomaly detection algorithms that are not using a pre-labeled dataset [110]. Though it is possible to train with unlabeled data, we are not yet at a point where we are able to evaluate models without any labeled data. Many studies that have this issue simply apply a threshold to their data and label everything outside of that threshold as an anomaly. This could be based on the mean of the dataset as a whole, the mean of specific categories, or the

rolling mean of sequential data [111] [84]. We have chosen to take a similar approach. As previously discussed, our data is essentially many individual time series, each representing a network path between distinct regions or data centers. To label anomalies in our test data set, we find the mean values for each of these time series and define a threshold, t, as the mean, μ, plus or minus a constant, c multiplied by the standard deviation, σ, for that series. We also check these manually and added in additional points we thought were anomalous. Fortunately, we are dealing with performance data where identifying anomalies can be more obvious than in other types of data. If the throughput or latency are significantly higher or lower than the other recent data points in the series, this is an anomaly.

$$t = \mu \pm c * \sigma \tag{6.2}$$

Because we are using unsupervised predictive models, we do not have to label the training data, only the testing data. For training, we rely on the fact that there is much more normal data than abnormal data. This should result in our models being able to predict the normal data fairly accurately and not being able to predict the abnormal data with much accuracy [112].

For data features, we will use the same ones as for the prediction models already discussed: historical latency, throughput, and buffer sizes. Performance for these types of models tends to be better with low dimensional data [113]. We attempted to add additional features such as kernel version, time and day of the test encoded with a cyclical encoding, and information about number of errors and retransmissions. None of this improved the accuracy of the predictions or anomaly detecting power of any of the networks, and in many cases was actively detrimental.

6.5.2. Training

These algorithms were all trained in the same manner as previously, attempting to maximize predictive performance. To detect anomalies from our prediction models, we take the difference between the prediction and the actual measured value and compare that to our average prediction error. If the prediction error is above mean prediction error plus some threshold, we label this as a detected anomaly. This should allow us to compare the anomaly detection performance of different prediction models.

6.5.3. Anomaly Detection Results

We trained several different architectures for prediction and tested their ability to also detect anomalies. The results are listed in Table 6.9. Here, we show the precision, P, recall, R, and the F1-score of each model. These are calculated as follows from the true positive, false positive, and false negative rates.

$$P = \frac{TP}{TP + FP} \tag{6.3}$$

$$R = \frac{TP}{TP + FN} \tag{6.4}$$

$$F1 = \frac{2(P * R)}{P + R} \tag{6.5}$$

The best possible F1 score is 1.0, meaning that we identified all anomalies perfectly and had no false positives. For each of the models that we tested, we searched for the optimal anomaly detection threshold that would result in the best F1 score.

Table 6.9: Anomaly detection F1, Precision, and Recall scores for iPerf 32 stream throughput anomalies

.	**iPerf 32 stream throughput anomaly**		
Model	**F1**	**P**	**R**
SMA	0.604	0.515	0.73
MLP	0.689	0.59	0.81
LSTM	0.680	0.58	0.82
GRU	0.664	0.55	0.83
CNN	0.698	0.61	0.82
CNN + GRU	0.675	0.57	0.82
LSTNet	0.521	0.43	0.66
Transformer	0.623	0.58	0.66

From these models, the CNN performed the best with the least false positives and false negatives. However, the F1 score for most of the models was quite close and none of the models had outstanding results. Most of the machine learning models performed moderately better than a simple moving average, with a large trade off in terms of efficiency and resource utilization. The main contributor for the relatively modest F1 scores was high rate of false positives. This is related to the data. There are about 10 links that have a high amount of variance between measurements. This leads to a situation where these are not marked as anomalies, because it is normal for these links to have high variance, but for the same reason it also makes the values difficult to predict. The large prediction error marks it as an anomalous performance event. All of this combined yields a large amount of false positives for a subset of the links.

6.6. Conclusions

In this chapter, we set out to predict day to day throughput between endpoints in a cloud environment using data gathered over a 6 month period. We trained and tested several neural network based prediction models to predict throughput based on historical

throughput, latency, and TCP receive buffer size and compared them to the results from a classic ARIMA model. Overall, the neural network architectures performed much better than ARIMA and a hybrid model, LSTNet, showed the least prediction error. We also found multi-stream throughput to be generally more predictable than single stream throughput.

We then used these neural network prediction models to perform anomaly detection. We tested these against a simple moving average model. The neural networks outperformed the moving average. None of the models showed an F1 score better than 0.7. This is probably attributable to multiple factors. One such factor was a portion of our network links that had much higher variance than others, which led to a high number of false positives.

We see our results here as a good baseline to build on. In the future, we would like to apply these models to data on smaller time scales and compare the predictions against the results from several formula based throughput prediction models. We would also like to gather and analyse throughput data for network connections with a cap greater than 32 Gbps to compare them. We also plan to use our experience here to improve and develop new models to achieve greater accuracy at this task.

Chapter 7

Conclusion

In the previous chapters of this thesis proposal we have discussed the work we have done in cloud network performance analytics and benchmarking. In Chapter 2, we thoroughly explored related work in this field. In Chapter 3, we defined best practices for network testing and benchmarking including the use of tools such as Ping, iPerf, and Netperf. We showed settings and configurations to use for optimal throughput and latency results for cloud benchmarking and we discussed the use of the cloud benchmark automation tool, PerfKit Benchmarker (PKB), to automate the creation and provisioning of cloud infrastructure for benchmarks. We explained how each tool can be used to measure aspects of network performance and the advantages and disadvantages of using each tool.

The various network configuration options can be opaque in how they function, affect performance, and interact with other configuration options. Our goal in this chapter was to provide some clarity about how different variables affect throughput and latency. There are usually some simple changes we can make to improve our achieved throughput, such as increasing the maximum TCP send and receive buffers or choosing the appropriate TCP congestion control algorithm. Latency is a function of physical distance and network congestion, so if we cannot use machines that are physically closer together, we would have to try to decrease network congestion, which can also be done by all of machines on the network using appropriate TCP congestion control algorithms that are relatively fair in regards to the percentage of available bandwidth they consume.

Chapter 4 covers how take measurements efficiently, both in terms of single benchmarks as well as scheduling large numbers of benchmarks. We show how we can use relatively simple

statistics calculations to minimize the amount of time we need to run throughput tests for while maintaining accurate results. When running a network throughput benchmark, we gather interim results at a specific interval. Each of these interim results shows the network throughput in bits per second over that specific interval. If we use each of these interim results as a measurement sample, we can use these to calculate the mean throughput value within a confidence interval. This confidence interval will have a width ($\pm$ the mean). By specifying the desired width of the confidence interval, we can stop the measurement process after the desired confidence interval width is achieved. This removes guesswork from our benchmarking efforts and allows us to spend less time and money running tests and also alerts us to instances where our test durations may not be long enough to achieve good results due to high throughput variance. This can also be applied to other benchmarks such as latency, packets-per-second, and none network benchmarks that have a similar structure.

We then proposed a novel scheduling solution for large numbers of cloud network benchmarks that uses a graph representation of resources and the maximum matching for efficient batch scheduling. This algorithm represent virtual machines are nodes on a graph and benchmarks as edges between nodes. Multiple edges and self edges are possible when more than one benchmark needs to be executed between two virtual machines and when a benchmark only needs a single machine. From this graph of machines and benchmarks, we can greedily schedule batches of benchmarks. To schedule batches, we use the maximum matching, the largest set of edges in the graph which share no nodes. For us this means the largest set of benchmarks where their required machines do not overlap.

Using this solution, we are able to run a large number of cloud benchmarks on a daily basis while remaining within the various regional and cloud-wide quotas we are subject to. It minimizes the time spent benchmarking, which also reduces costs. This batch scheduling enables the data collection necessary for the analysis and model training we perform in subsequent chapters.

Then in Chapter 5, we presented a thorough analysis of cloud network dynamics and compared them with the dynamics for an emulated dedicated network with similar host and network configurations. For each of these networks we measured throughput across a wide range of RTTs. We compared the throughput profiles of these networks as well as their concavity-utilization coefficients and conclude that they are quite comparable at most RTTs, except for a few connections where the cloud network performed significantly worse at the same RTT as the emulated and other cloud connections. This is attributable to increased congestion and retransmissions along these particular connections.

We also ran these test using several different TCP congestion control algorithms and on the emulated network we used several different loss rates, from 0 loss to 1 in 500 packets loss. From our congestion control tests, at low or zero loss CUBIC, HTCP, and BBR perform largely similarly. Their differences become more apparent when faced with random losses (losses not caused by congestion). When there is a high loss rate, CUBIC and HTCP are able to achieve significantly lower throughput than BBR, due to how each algorithm handles loss. We also saw that even with induced loss and high latency, the best way to increase achieved throughput in those cases is to use multiple parallel data streams. This method is more effective with high latency, and when faced with high loss, it is most effective when using BBR.

In Chapter 6, we used the data and network knowledge we gained from the previous chapters to attempt to forecast future throughput and detect throughput anomalies. We trained and tested several traditional and deep learning models to forecast cloud throughput. To do this, we used a data set of daily cloud performance data gathered over the course of a year. This data set has measurements from all cloud regions to every other cloud region in Google Cloud's network taken on a daily basis. We treated each of these region-pairs as a time series. To train and test the network we split each of these time series into a sliding of 10 days of data aiming predict the 11th day. Because this is time series data, it makes

it more difficult to do k-fold cross validation because ideally we don't want to train on data that happens after data in our test dataset. We split the dataset at a certain date and made everything after that date the testing set, and everything before that date the training set. We found an effective deep learning model for prediction and concluded that all of the deep learning models we tested performed much better than the more traditional time series prediction models we tested for this task, Here our top performing model was LST-NET, a hybrid CNN LSTM model.

We then trained and tested these models for anomaly detection as well. This task was more difficult, as we lacked a labeled training dataset and the variance of the throughput for some network links was quite high. Overall we only achieved a top F1 score of 0.69 due to a high number of false positives from the high variance links.

Overall, in this work we have presented a thorough work of how to optimize cloud network performance, how to measure that network performance efficiently and at scale, and what we can do with that collected data. Here, our work has results in substantial real-world cost and time savings. These continued measurements also allow us to gain useful insights into the performance characteristics of our network. We can use the data to forecast future performance, allowing us to choose an optimal time to execute network intensive tasks. We can also detect performance anomalies, alerting us to events when the network is not performing as it should and allowing us to make necessary adjustments.

7.1. Future Work

There are several avenues to continue this work. The first is expansion of data gathering. In this work, the public cloud provider we gathered all of our performance data from was Google Cloud. It could be interesting to compare between the networks of multiple cloud providers. We could also expand these comparisons to larger cloud network links, which currently go up to 100 Gbps in capacity. Currently our long term measurements are on a

daily time scale. Network conditions can change much more rapidly than that. It would be interesting to collect network performance data on a much smaller time scale, such as hourly or even every minute. This data could be used to explore how performance changes across an hour, or day. It could help us develop models that better learn how network performance should look during the cycles of any given hour or day.

Additionally, we would like to explore the effect of cross traffic on performance in cloud networks. This would be difficult on a production network, because it is difficult or impossible for a user to determine exactly where VMs will be allocated. This will also make it difficult for us to us VMs to effect each other's communications with competing traffic. If we are able to get multiple VMs allocated on the same rack with a cluster-style placement policy, which is explicitly not guaranteed, we could possibly use the traffic of multiple VMs to effect the network performance of a target VM located in the same rack. A simple way to do this would be to emulate everything on our own machines, taking the guesswork out of relying on cloud VM placement.

In continuation of our work on efficient benchmarking, we believe that we could further improve the efficiency gains of our approach. This could possibly be done by using Bayesian credible intervals instead of confidence intervals. They are similar concepts, but Bayesian credible intervals also take a best guess as a parameter. By passing the algorithm a likely value for the mean, this could allow the interval to converge to the target width even more quickly. For network throughput, this likely mean value could either be an educated guess based off of latency, known bandwidth cap, and send/receive buffer sizes, or it could be based off of historical mean values for the same network link.

We would also like to expand our work in performance forecasting an anomaly detection. If we were able to take measurements on a smaller time scale, it could be very interesting to see how these algorithms perform on a minute by minute or hour by hour time scale. This would also allow us to train to recognize hourly and daily cycles, something we could

only do to a very limited extent with our daily performance data. This could lead to both better prediction and anomaly detection algorithms for network performance than what we currently have. Additionally, there are a constantly evolving set of best practices when it comes to time series prediction models. Though we have used state of the art models here, in a few years time, it is likely that new models will be able to perform this task with improved results. Additionally, there are additional avenues of anomaly detection that we would also like to employ against this problem, such as positive unlabeled learning.

Appendix A

Source Code

This appendix is intended to point the reader to the source code for each major piece of work from this thesis. We have linked to the repositories of each, all of which are hosted on Github.com.

In Chapter 4 we discussed how to use confidence intervals to take efficient network measurements. Our implementation of this can be found at: https://github.com/SMU-ATT-Center-for-Virtualization/iperf_early_stopping

From Chapter 4.3, we present our implementation of our cloud benchmark batch scheduling solution, **pkb_scheduler**. https://github.com/SMU-ATT-Center-for-Virtualization/pkb_scheduler.

From Chapter 5, our network analysis and comparison can be found at https://github.com/SMU-ATT-Center-for-Virtualization/public˙private˙network˙analysis

In Chapter 6 we discussed how we can use collected network data to do throughput prediction and anomaly detection. Our throughput prediction analysis and models can be found at https://github.com/dPhanekham/throughput_prediction_public_cloud. Our anomaly detection models and analysis can be found at https://github.com/SMU-ATT-Center-for-Virtualization/network_performance_anomaly_detect.

If you are looking for the underlying datasets used for network analysis or model training, these are too large to be hosted on Github.com and can be obtained from contacting Derek Phanekham at dphanekham@gmail.com.

Acronyms

CIDR Classless inter-domain routing. 46

FLOPS Floating Point Operations per Second. 8

FTP File Transfer Protocol. 9

ICMP Internet Control Message Protocol. 40, 146

MTU Maximum Transmission Unit. 11, 31, 99, 119

NIC Network Interface Controller. 118

UDP User Datagram Protocol. 41, 42

GLOSSARY

Bandwidth A measurement of the maximum capacity of a network or link (bits/sec). 4

Bulk Data Transfer This refers to the process of moving a large quantity of data across a network. Depending on the size of the data, this can use a specialized application such as Globus, or something like FTP. Large data transfers can require scheduling and can result in high bandwidth utilization for an extended period of time. 59

Cloud (Network) A cloud network refers the on-demand computing capability provided by a company with a data center or group of data centers. This usually operates on a infrastructure-as-a-service, or platform-as-a-service model. Cloud networks are useful in their ability to abstract infrastructure management away from the user and be incredibly scalable. 1, 32, 35, 59

GNU Linear Programming Kit (GLPK) An open source solver for large-scale linear programming and mixed integer programming problems [92]. 74

Internet Control Message Protocol (ICMP) A protocol commonly used to communicate errors and other operational information on a network. 12

iPerf A tool for measuring network performance including throughput, latency, and jitter. It has a wide variety of options to specify tests that fit the given requirements. xv, 12, 41, 77, 98, 100, 119, 122, 129, 130, 135

Latency (Network) A measurement of the amount of time it takes to send a message or packet and have it be received on the other end of the connection. This can refer to one-way latency or round-trip latency. One-way latency measures the amount of time it takes for a message to travel from sender to receiver. Round-trip latency measures the amount of time it takes for a message to travel from sender to receiver plus the amount of time it takes for a response to travel from receiver back to the sender. 3

Mean Absolute Error (MAE) The mean of the the absolute value of the residuals.

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|.$$ 127

Mean Absolute Percentage Error (MAPE) MAPE measures how far off predictions are from the true value on average as a percentage.

$$\text{MAPE} = \frac{100}{n} * \sum_{i=1}^{n}\left(\frac{|y-\hat{y}|}{|y|}\right)$$. 127, 128

Mean squared error (MSE) The mean of the the absolute value of the residuals.

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2.$$ 127

Netperf A very versatile tool for measuring network performance including throughput, latency, and jitter. It has a wide variety of options to specify tests that fit the given requirements. 12, 30, 41, 98, 119

PerfKit Benchmarker (PKB) An open source framework for running benchmarks on cloud infrastructure. 15, 36, 77, 137

Ping A standard utility used to measure network RTT latency with ICMP. This is also sometimes referred to as a ping-pong test because it measures to amount of times it takes to send a packet to a host (ping) and to receive a reply packet from that host (pong). 12, 30, 98

Root mean squared error (RMSE) The square root of the mean squared error

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}.$$ 127

Round Trip Time (RTT) The duration it takes for a packet to go from point A to point B, be acknowledged, and return to point A. 8, 12, 93, 97, 99, 108, 123, 139, 146

Service Level Agreement (SLA) A contract between a customer and a service provider (cloud, network, software, etc.) that details the services provided and the level of standards the provider is required to meet. 3, 59

Software Defined Networking (SDN) Software defined networking is a network architectural paradigm where the control plane of the network is separated from the data plane. This allows for more centralized orchestration of flows across the network. 2, 9

Throughput (Network) Sometimes also referred to as achievable throughput. This is the amount of data that can be transferred between two endpoints on a network in a given timeframe (bits/sec). 2–6, 9–12, 14, 17, 18, 21–24, 27, 31, 32, 40, 58–60, 62–65, 68–72, 74, 76, 89, 90, 93, 97, 117–125, 127, 128, 130, 132, 135, 137, 139

Time series Any series of data organized by time. 18

Time series forecasting prediction of future time series. 18

Transmission Control Protocol (TCP) One of the most widely used protocols for network communications. It is also commonly referenced as part of the TCP/IP protocol suite. TCP is connection oriented and uses a congestion avoidance algorithm to control its traffic. There are many different variants of TCP with different congestion avoidance algorithms such as CUBIC, RENO, and BBR. 4, 9–12, 15–17, 22, 24, 31–33, 41, 65, 68, 93, 97, 119, 121–125, 128, 136

Virtual Machine (VM) A virtual machine is essentially an emulated instance of a computer. They rely on virtualized layer or hypervisor running on a physical server. This allows a server to run multiple isolated 'guest' operating systems on a single physical system. There are several types of virtual machines and levels of virtualization from full virtualization to process virtualization. 2, 16, 37, 74, 77, 78, 80, 84, 85, 87, 91, 98, 99, 141

Virtual Private Cloud (VPC) A Virtual Private Cloud is a type of overlay network similar to a VPN. A VPC allows us to operate on a private network connecting multiple virtual machines possibly across multiple data centers. To the user it acts and appears very similarly to as if they were operating on a private LAN. 99

Virtual Private Network (VPN) A Virtual Private Network is an encrypted tunnel over a public network connecting two private networks (or a host to a network, or a host to another host). It makes the endpoint logically act as if they are sharing the same private network. 14

BIBLIOGRAPHY

[1] A. Aljohani, D. Holton and I. Awan, *Modeling and performance analysis of scalable web servers deployed on the cloud*, in *2013 Eighth International Conference on Broadband and Wireless Computing, Communication and Applications*, pp. 238–242, 2013. DOI. 2

[2] A. Jindal, V. Podolskiy and M. Gerndt, *Performance modeling for cloud microservice applications*, in *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, ICPE '19, pp. 25–32, Association for Computing Machinery, 2019. DOI. 2

[3] A. Uta and H. Obaseki, *A performance study of big data workloads in cloud datacenters with network variability*, in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE '18, pp. 113–118, Association for Computing Machinery, 2018. DOI. 2

[4] A. Iosup, N. Yigitbasi and D. Epema, *On the performance variability of production cloud services*, in *Proceedings of the 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGRID '11, pp. 104–113, IEEE Computer Society, 2011. DOI. 2

[5] C. Huang, J. Zhang, T. Huang and Y. Liu, *Providing guaranteed network performance across tenants: Advances, challenges and opportunities*, *China Communications* **18** (2021) 152–174. 2

[6] J. C. Mogul and L. Popa, *What we talk about when we talk about cloud network performance*, *ACM SIGCOMM Computer Communication Review* **42** (2012) 44–48. 2

[7] N. S. V. Rao, Q. Liu, Z. Liu, R. Kettimuthu and I. Foster, *Throughput analytics of data transfer infrastructures*, in *Testbeds and Research Infrastructures for the Development of Networks and Communities* (H. Gao, Y. Yin, X. Yang and H. Miao, eds.), Springer International Publishing, 2019. 2, 6, 17, 95

[8] S. Baucke, R. B. Ali, J. Kempf, R. Mishra, F. Ferioli and A. Carossino, *Cloud atlas: A software defined networking abstraction for cloud to WAN virtual networking*, in *2013 IEEE Sixth International Conference on Cloud Computing*, pp. 895–902, 2013. DOI. 2

[9] Z. Liu, Z. Wang, X. Yin, X. Shi, Y. Guo and Y. Tian, *Traffic matrix prediction based on deep learning for dynamic traffic engineering*, in *2019 IEEE Symposium on Computers and Communications (ISCC)*, pp. 1–7, 2019. DOI. 2, 119

[10] R. Aronoff, K. Mills and M. Wheatley, *Transport layer performance tools and measurement*, *IEEE Network* **1** (1987) 21–31. 8

[11] U. Krishnaswamy and I. Scherson, *A framework for computer performance evaluation using benchmark sets*, *IEEE Transactions on Computers* **49** (2000) 1325–1338. 8

[12] P. Gunningberg, M. Bjorkman, E. Nordmark, S. Pink, P. Sjodin and J.-E. Stromquist, *Application protocols and performance benchmarks*, *IEEE Communications Magazine* **27** 30–36. 9

[13] T. Shah and G. Hura, *Network simulator (NTSIM)-a benchmark prototype network architecture*, in *Fourth IEEE Region 10 International Conference TENCON*, pp. 1093–1098, 1989. DOI. 9

[14] S. Lee, J. Leaney, T. O'Neill and M. Hunter, *Performance benchmark of a parallel and distributed network simulator*, in *Workshop on Principles of Advanced and Distributed Simulation (PADS'05)*, pp. 101–108, 2005. DOI. 9

[15] C. Caini, R. Firrincieli and D. Lacamera, *SAT01-5: An emulation approach for the evaluation of enhanced transport protocols performance in satellite networks*, in *IEEE Globecom 2006*, pp. 1–5, 2006. DOI. 9

[16] B. Lantz, B. Heller and N. McKeown, *A network in a laptop: rapid prototyping for software-defined networks*, in *Proceedings of the Ninth ACM SIGCOMM Workshop on Hot Topics in Networks - Hotnets '10*, pp. 1–6, ACM Press, 2010. DOI. 9

[17] B. Huang, M. Bauer and M. Katchabaw, *Hpcbench - a linux-based network benchmark for high performance networks*, in *19th International Symposium on High Performance Computing Systems and Applications (HPCS'05)*, pp. 65–71, 2005. DOI. 10, 13

[18] S. S. Chaudhari and R. C. Biradar, *Survey of bandwidth estimation techniques in communication networks*, *Wireless Personal Communications* **83** (2015) 1425–1476. 10

[19] M. Jain and C. Dovrolis, *Ten fallacies and pitfalls on end-to-end available bandwidth estimation*, in *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement - IMC '04*, p. 272, ACM Press, 2004. DOI. 10

[20] N. V. Mnisi, O. J. Oyedapo and A. Kurien, *Active throughput estimation using RTT of differing ICMP packet sizes*, in *2008 Third International Conference on Broadband Communications, Information Technology & Biomedical Applications*, pp. 480–485, 2008. DOI. 10

[21] D. Kachan, E. Siemens and V. Shuvalov, *Available bandwidth measurement for 10 gbps networks*, in *2015 International Siberian Conference on Control and Communications (SIBCON)*, pp. 1–10, 2015. DOI. 10

[22] G. De Silva, M. C. Chan and W. T. Ooi, *Throughput estimation for short lived tcp cubic flows*, in *Proceedings of the 13th International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*, MOBIQUITOUS 2016, (New York, NY, USA), p. 227–236, Association for Computing Machinery, 2016. DOI. 11, 125

[23] "IPerf2." https://sourceforge.net/projects/iperf2/. 12

[24] "iPerf - the TCP, UDP and SCTP network bandwidth measurement tool." https://iperf.fr/. 12

[25] "HewlettPackard/netperf." https://github.com/HewlettPackard/netperf. 12

[26] G. Juckeland, M. Kluge, W. Nagel and S. Pfluger, *Performance analysis with BenchIT: portable, flexible, easy to use*, in *First International Conference on the Quantitative Evaluation of Systems, 2004. QEST 2004. Proceedings.*, pp. 320–321, 2004. DOI. 13

[27] "SPEC: Standard Performance Evaluation Corporation." https://www.spec.org. 13

[28] L. Klaver, T. van der Knaap, J. van der Geest, E. Harmsma, B. van der Waaij and P. Pileggi, *Towards independent run-time cloud monitoring*, in *Companion of the ACM/SPEC International Conference on Performance Engineering*, pp. 21–26, ACM, 2021. DOI. 14

[29] F. A. Arshad, G. Modelo-Howard and S. Bagchi, *To cloud or not to cloud: A study of trade-offs between in-house and outsourced virtual private network*, in *2012 20th IEEE International Conference on Network Protocols (ICNP)*, pp. 1–6, IEEE, 2012. DOI. 14

[30] L. Gillam, B. Li, J. O'Loughlin and A. P. S. Tomar, *Fair benchmarking for cloud computing systems*, *Journal of Cloud Computing: Advances, Systems and Applications* **2** 6. 14

[31] M. B. Chhetri, S. Chichin, Q. B. Vo and R. Kowalczyk, *Smart CloudBench – automated performance benchmarking of the cloud*, in *2013 IEEE Sixth International Conference on Cloud Computing*, pp. 414–421, 2013. DOI. 15

[32] J. Scheuner and P. Leitner, *Performance benchmarking of infrastructure-as-a-service (IaaS) clouds with cloud WorkBench*, in *Companion of the 2019 ACM/SPEC International Conference on Performance Engineering*, ICPE '19, pp. 53–56, Association for Computing Machinery, 2019. DOI. 15

[33] X. Liu, D. Fang and P. Xu, *Automated performance benchmarking platform of IaaS cloud*, in *2021 IEEE 20th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pp. 1402–1405, 2021. DOI. 15

[34] "PerfKit Benchmarker." https://googlecloudplatform.github.io/PerfKitBenchmarker, 2021. 15, 77

[35] Y. Wu, S. Kumar and S.-J. Park, *On transport protocol performance measurement over 10gbps high speed optical networks*, in *2009 Proceedings of 18th International Conference on Computer Communications and Networks*, pp. 1–6, 2009. DOI. 16

[36] A. Esterhuizen and A. E. Krzesinski, *TCP congestion control comparison*, *Southern Africa Telecommunication Networks and Applications Conference* (2012) 6. 16, 17

[37] T. Lukaseder, L. Bradatsch, B. Erb, R. W. Van Der Heijden and F. Kargl, *A comparison of TCP congestion control algorithms in 10g networks*, in *2016 IEEE 41st Conference on Local Computer Networks (LCN)*, pp. 706–714, 2016. DOI. 16, 17

[38] J. Crichigno, Z. Csibi, E. Bou-Harb and N. Ghani, *Impact of segment size and parallel streams on TCP BBR*, in *2018 41st International Conference on Telecommunications and Signal Processing (TSP)*, pp. 1–5, 2019. DOI. 16, 66

[39] B. Jaeger, D. Scholz, D. Raumer, F. Geyer and G. Carle, *Reproducible measurements of TCP BBR congestion control*, *Computer Communications* **144** (2019) 31–43. 16

[40] Y. Cao, A. Jain, K. Sharma, A. Balasubramanian and A. Gandhi, *When to use and when not to use BBR: An empirical analysis and evaluation study*, in *Proceedings of the Internet Measurement Conference*, pp. 130–136, ACM, 2013. DOI. 16, 35

[41] P. Ha, M. Vu, T.-A. Le and L. Xu, *TCP BBR in cloud networks: Challenges, analysis, and solutions*, in *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, pp. 943–953, 2021. DOI. 16

[42] A. A. Philip, R. Ware, R. Athapathu, J. Sherry and V. Sekar, *Revisiting TCP congestion control throughput models & fairness properties at scale*, in *Proceedings of the 21st ACM Internet Measurement Conference*, IMC '21, pp. 96–103, Association for Computing Machinery, 2021. DOI. 16

[43] E. F. Kfoury, J. Gomez, J. Crichigno and E. Bou-Harb, *An emulation-based evaluation of TCP BBRv2 alpha for wired broadband*, *Computer Communications* **161** (2021) 212–224. 16

[44] B. W. Settlemyer, N. S. V. Rao, S. W. Poole, S. W. Hodson, S. E. Hicks and P. M. Newman, *Experimental analysis of 10gbps transfers over physical and emulated dedicated connections*, in *2012 International Conference on Computing, Networking and Communications (ICNC)*, pp. 845–850, 2012. DOI. 16, 17

[45] M. Ahmad, A. B. Ngadi, A. Nawaz, U. Ahmad, T. Mustafa and A. Raza, *"a survey on tcp cubic variant regarding performance"*, in *2012 15th International Multitopic Conference (INMIC)*, pp. 409–412, 2012. DOI. 16

[46] B. Tierney, E. Dart, E. Kissel and E. Adhikarla, *Exploring the BBRv2 congestion control algorithm for use on data transfer nodes*, in *2021 IEEE Workshop on Innovating the Network for Data-Intensive Science (INDIS)*, pp. 23–33, 2021. DOI. 16

[47] Y.-J. Song, G.-H. Kim, I. Mahmud, W.-K. Seo and Y.-Z. Cho, *Understanding of BBRv2: Evaluation and comparison with BBRv1 congestion control algorithm*, *IEEE Access* **9** (2021) 37131–37145. 16

[48] R. P. Tahiliani, M. P. Tahiliani and K. C. Sekaran, *TCP variants for data center networks: A comparative study*, in *2012 International Symposium on Cloud and Services Computing*, pp. 57–62, 2021. DOI. 17

[49] A. Ganji, A. Singh and M. Shahzad, *Choosing TCP variants for cloud tenants – a measurement based approach*, in *2020 29th International Conference on Computer Communications and Networks (ICCCN)*, pp. 1–9, 2020. DOI. 17

[50] S. Jouet and D. P. Pezaros, *Measurement-based TCP parameter tuning in cloud data centers*, in *2013 21st IEEE International Conference on Network Protocols (ICNP)*, pp. 1–3, 2013. DOI. 17

[51] B. W. Settlemyer, J. D. Dobson, S. W. Hodson, J. A. Kuehn, S. W. Poole and T. M. Ruwart, *A technique for moving large data sets over high-performance long distance networks*, in *2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST)*, pp. 1–6, 2011. DOI. 17

[52] Y. Sugawara, T. Yoshino, H. Tezuka, M. Inaba and K. Hiraki, *Effect of packet shuffler on parallel TCP stream network*, in *Seventh International Conference on Networking (icn 2008)*, pp. 180–185, 2008. DOI. 17

[53] S. Ahmad and M. J. Arshad, *Enhancing fast TCP's performance using single TCP connection for parallel traffic flows to prevent head-of-line blocking*, *IEEE Access* **7** (2019) 148152–148162. 17

[54] Q. Liu and N. Rao, *On concavity and utilization analytics of wide-area network transport protocols*, in *2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pp. 430–438, 2018. DOI. 17

[55] H.-W. Kim, J.-H. Lee, Y.-H. Choi, Y.-U. Chung and H. Lee, *Dynamic bandwidth provisioning using ARIMA-based traffic forecasting for Mobile WiMAX*, *Computer Communications* **34** (2011) 99 – 106. 21, 23

[56] A. Biernacki, *Improving quality of adaptive video by traffic prediction with (f)arima models*, *Journal of Communications and Networks* **19** (2017) 521–530. 21, 23

[57] M. Mirza, J. Sommers, P. Barford and X. Zhu, *A machine learning approach to tcp throughput prediction*, *IEEE/ACM Trans. Netw.* **18** (Aug., 2010) 1026–1039. 22, 23

[58] J. Padhye, V. Firoiu, D. Towsley and J. Kurose, *Modeling tcp throughput: A simple model and its empirical validation*, *SIGCOMM Comput. Commun. Rev.* **28** (Oct., 1998) 303–314. 22, 124

[59] J.-H. Hwang and C. Yoo, *Formula-based TCP throughput prediction with available bandwidth*, *IEEE Communications Letters* **14** 363–365. 22

[60] N. Bui, F. Michelinakis and J. Widmer, *A model for throughput prediction for mobile users*, in *European Wireless 2014; 20th European Wireless Conference*, pp. 1–6, 2014. 22

[61] B. Wei, W. Kawakami, K. Kanai and J. Katto, *A history-based tcp throughput prediction incorporating communication quality features by support vector regression for mobile network*, in *2017 IEEE International Symposium on Multimedia (ISM)*, pp. 374–375, 2017. DOI. 23

[62] N. S. V. Rao, S. Sen, Z. Liu, R. Kettimuthu and I. Foster, *Learning concave-convex profiles of data transport over dedicated connections*, in *Machine Learning for Networking* (E. Renault, P. Muhlethaler and S. Bourmerdassi, eds.), Lecture Notes in Computer Science 11407, Springer, 2019. 24

[63] B. Wei, M. Okano, K. Kanai, W. Kawakami and J. Katto, *Throughput prediction using recurrent neural network model*, in *2018 IEEE 7th Global Conference on Consumer Electronics (GCCE)*, pp. 107–108, 2018. DOI. 25

[64] Y. Bengio, P. Simard and P. Frasconi, *Learning long-term dependencies with gradient descent is difficult*, *IEEE Transactions on Neural Networks* **5** (1994) 157–166. 25

[65] K. Cho, B. van Merriënboer, D. Bahdanau and Y. Bengio, *On the properties of neural machine translation: Encoder–decoder approaches*, in *Proceedings of SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation*, (Doha, Qatar), pp. 103–111, Association for Computational Linguistics, Oct., 2014. DOI. 25

[66] S. Hochreiter and J. Schmidhuber, *Long short-term memory*, *Neural Comput.* **9** (Nov., 1997) 1735–1780. 25

[67] J. Chung, Ç. Gülçehre, K. Cho and Y. Bengio, *Empirical evaluation of gated recurrent neural networks on sequence modeling*, *CoRR* **abs/1412.3555** (2014) , [1412.3555]. 25

[68] J. B. Yang, M. N. Nguyen, P. P. San, X. L. Li and S. Krishnaswamy, *Deep convolutional neural networks on multichannel time series for human activity recognition*, in *Proceedings of the 24th International Conference on Artificial Intelligence*, IJCAI'15, p. 3995–4001, AAAI Press, 2015. 25

[69] T.-Y. Kim and S.-B. Cho, *Predicting residential energy consumption using CNN-LSTM neural networks*, *Energy* **182** (2019) 72 – 81. 25

[70] Büyükşahin and Ertekin, *Improving forecasting accuracy of time series data using a new ARIMA-ANN hybrid method and empirical mode decomposition*, *Neurocomputing* **361** (2019) 151 – 163. 25

[71] G. Lai, W.-C. Chang, Y. Yang and H. Liu, *Modeling long- and short-term temporal patterns with deep neural networks*, in *The 41st International ACM SIGIR Conference on Research: Development in Information Retrieval*, SIGIR '18, (New York, NY, USA), p. 95–104, Association for Computing Machinery, 2018. DOI. 26, 120

[72] C. Gao, Y. Chen, Z. Wang, H. Xia and N. Lv, *Anomaly detection frameworks for outlier and pattern anomaly of time series in wireless sensor networks*, in *2020 International Conference on Networking and Network Applications (NaNA)*, pp. 229–232, 2020. DOI. 26

[73] F. Knorn and D. J. Leith, *Adaptive kalman filtering for anomaly detection in software appliances*, in *IEEE INFOCOM Workshops 2008*, pp. 1–6, 2008. DOI. 26

[74] M. A. August, M. H. Diallo, C. T. Graves, S. M. Slayback and D. Glasser, *AnomalyDetect: Anomaly detection for preserving availability of virtualized cloud services*, in *2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS*W)*, pp. 334–340, 2017. DOI. 26

[75] A. Andrade, C. Montez, R. Moraes, A. Pinto, F. Vasques and G. L. da Silva, *Outlier detection using k-means clustering and lightweight methods for wireless sensor networks*, in *IECON 2016 - 42nd Annual Conference of the IEEE Industrial Electronics Society*, pp. 4683–4688, 2016. DOI. 26

[76] L. Yang, Y. Lu, S. X. Yang, T. Guo and Z. Liang, *A secure clustering protocol with fuzzy trust evaluation and outlier detection for industrial wireless sensor networks*, . 26

[77] Z. Zhao, Y. Zhang, X. Zhu and J. Zuo, *Research on time series anomaly detection algorithm and application*, in *2019 IEEE 4th Advanced Information Technology, Electronic and Automation Control Conference (IAEAC)*, vol. 1, pp. 16–20, 2019. DOI. 26

[78] Y. Chen, J. Qian and V. Saligrama, *A new one-class SVM for anomaly detection*, in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 3567–3571, 2013. DOI. 27

[79] O. I. Provotar, Y. M. Linder and M. M. Veres, *Unsupervised anomaly detection in time series using LSTM-based autoencoders*, in *2019 IEEE International Conference on Advanced Trends in Information Theory (ATIT)*, pp. 513–517, 2019. DOI. 27

[80] X. Xu, H. Zhao, H. Liu and H. Sun, *LSTM-GAN-XGBOOST based anomaly detection algorithm for time series data*, in *2020 11th International Conference on Prognostics and System Health Management (PHM-2020 Jinan)*, pp. 334–339, 2020. DOI. 27

[81] Z. Chen, C. K. Yeo, B. S. Lee and C. T. Lau, *Autoencoder-based network anomaly detection*, in *2018 Wireless Telecommunications Symposium (WTS)*, pp. 1–5, 2018. DOI. 27

[82] F. Lüer, D. Mautz and C. Böhm, *Anomaly detection in time series using generative adversarial networks*, in *2019 International Conference on Data Mining Workshops (ICDMW)*, pp. 1047–1048, 2019. DOI. 27

[83] Y. Choi, H. Lim, H. Choi and I.-J. Kim, *GAN-based anomaly detection and localization of multivariate time series data for power plant*, in *2020 IEEE International Conference on Big Data and Smart Computing (BigComp)*, pp. 71–74, 2020. DOI. 27

[84] G. Qin, Y. Chen and Y.-X. Lin, *Anomaly detection using LSTM in IP networks*, in *2018 Sixth International Conference on Advanced Cloud and Big Data (CBD)*, pp. 334–337, 2018. DOI. 27, 133

[85] P. Gaikwad, A. Mandal, P. Ruth, G. Juve, D. Król and E. Deelman, *Anomaly detection for scientific workflow applications on networked clouds*, in *2016 International Conference on High Performance Computing Simulation (HPCS)*, pp. 645–652, 2016. DOI. 28

[86] D. Leith and R. Shorten, *H-TCP: TCP for high-speed and long-distance networks*, *PFLDnet* (2004) 16. 34, 99

[87] S. Ha, I. Rhee and L. Xu, *CUBIC: a new TCP-friendly high-speed TCP variant*, *ACM SIGOPS Operating Systems Review* **42** (2008) 64–74. 34, 99

[88] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh and V. Jacobson, *BBR: Congestion-based congestion control: Measuring bottleneck bandwidth and round-trip propagation time*, *Queue* **14** (2013) 20–53. 35, 65

[89] M. Hock, R. Bless and M. Zitterbart, *Experimental evaluation of BBR congestion control*, in *2017 IEEE 25th International Conference on Network Protocols (ICNP)*, pp. 1–10. DOI. 35

[90] M. Dekking, ed., *A Modern Introduction to Probability and Statistics: Understanding Why and How*. Springer texts in statistics. Springer, 2005. 60

[91] M. A. Alsaih, R. Latip, A. Abdullah, S. K. Subramaniam and K. Ali Alezabi, *Dynamic job scheduling strategy using jobs characteristics in cloud computing*, *Symmetry* **12** (2020) 1638. 74

[92] A. Makhorin, "Glpk (gnu linear programming kit)." http://www.gnu.org/software/glpk/glpk.html. 74, 145

[93] D. Shah, *Maximal matching scheduling is good enough*, in *GLOBECOM '03. IEEE Global Telecommunications Conference (IEEE Cat. No.03CH37489)*, vol. 6, pp. 3009–3013 vol.6, 2003. DOI. 74

[94] R. M. Karp, *Reducibility among combinatorial problems*, in *Complexity of Computer Computations: Proceedings of a symposium on the Complexity of Computer Computations* (R. E. Miller, J. W. Thatcher and J. D. Bohlinger, eds.), pp. 85–103, Springer US, 1972. DOI. 75

[95] H. W. Lenstra, *Integer programming with a fixed number of variables*, *Mathematics of Operations Research* **8** (1983) 538–548. 75

[96] G. B. Dantzig, *Discrete-Variable Extremum Problems*, *INFORMS: Operations Research* **5** (1957) 266–277. 80

[97] C. Berge, *Two theorems in graph theory*, *Proceedings of the National Academy of Sciences of the United States of America* **43** (1957) 842–844. 81

[98] A. A. Hagberg, D. A. Schult and P. J. Swart, *Exploring network structure, dynamics, and function using networkx*, in *Proceedings of the 7th Python in Science Conference* (G. Varoquaux, T. Vaught and J. Millman, eds.), (Pasadena, CA USA), pp. 11 – 15, 2008. 82

[99] Z. Galil, *Efficient algorithms for finding maximum matching in graphs*, *ACM Computing Surveys* **18** (1986) 23–38. 83

[100] I. Foster and C. Kesselman, *Globus: A metacomputing infrastructure toolkit*, *International Journal for Supercomputer Applications* **11** (1997) 115–128. 92

[101] N. S. V. Rao, Q. Liu, S. Sen, Z. Liu and R. Kettimuthu, *Measurements and analytics of wide-area file transfers over dedicated connections*, in *International Conference on Distributed Computing and Networking*, 2019. 93

[102] N. S. Rao, Q. Liu, S. Sen, D. Towlsey, G. Vardoyan, R. Kettimuthu et al., *TCP throughput profiles using measurements over dedicated connections*, in *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, pp. 193–204, ACM. DOI. 94

[103] M. Dalton, D. Schultz, J. Adriaens, A. Arefin, A. Gupta, B. Fahs et al., *Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization*, *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI '18)* (2018) 16. 99

[104] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh and V. Jacobson, *BBR: Congestion based congestion control*, *ACM Queue* **14** (Dec., 2016) . 99

[105] A. Mishra, X. Sun, A. Jain, S. Pande, R. Joshi and B. Leong, *The great internet TCP congestion control census*, *Proceedings of the ACM on Measurement and Analysis of Computing Systems* **3** (2019) 45:1–45:24. 99

[106] T. Hacker, B. Athey and B. Noble, *The end-to-end performance effects of parallel tcp sockets on a lossy wide-area network*, in *Proceedings 16th International Parallel and Distributed Processing Symposium*, pp. 10 pp–, 2002. DOI. 106

[107] J. Lee, D. Gunter, B. Tierney, B. Allcock, J. Bester, J. Bresnahan et al., *Applied techniques for high bandwidth data transfers across wide area networks*, *Lawrence Berkeley National Laboratory* (2001) . 106

[108] Q. He, C. Dovrolis and M. Ammar, *On the predictability of large transfer tcp throughput*, *SIGCOMM Comput. Commun. Rev.* **35** (Aug., 2005) 145–156. 118

[109] C. Villamizar and C. Song, *High performance tcp in ansnet*, *SIGCOMM Comput. Commun. Rev.* **24** (Oct., 1994) 45–60. 124

[110] N. Moustafa, J. Hu and J. Slay, *A holistic review of network anomaly detection systems: A comprehensive survey*, *Journal of Network and Computer Applications* **128** (2019) 33–55. 132

[111] E. H. Budiarto, A. Erna Permanasari and S. Fauziati, *Unsupervised anomaly detection using k-means, local outlier factor and one class SVM*, in *2019 5th International Conference on Science and Technology (ICST)*, vol. 1, pp. 1–5, 2019. DOI. 133

[112] G. Pang, C. Shen, L. Cao and A. V. D. Hengel, *Deep learning for anomaly detection: A review*, *ACM Computing Surveys* **54** (2021) 1–38. 133

[113] W. Lu, Y. Cheng, C. Xiao, S. Chang, S. Huang, B. Liang et al., *Unsupervised sequential outlier detection with deep architectures*, *IEEE Transactions on Image Processing* **26** (2017) 4321–4330. 133