2017

Improving Network Performance Through Endpoint Diagnosis And Multipath Communications

Behnaz Arzani Arzani
University of Pennsylvania, barzani@seas.upenn.edu

Improving Network Performance Through Endpoint Diagnosis And Multipath Communications

Abstract

Components of networks, and by extension the internet can fail. It is, therefore, important to find the points of failure and resolve existing issues as quickly as possible. Resolution, however, takes time and its important to maintain high quality of service (QoS) for existing clients while it is in progress. In this work, our goal is to provide clients with means of avoiding failures if/when possible to maintain high QoS while enabling them to assist in the diagnosis process to speed up the time to recovery.

Fixing failures relies on first detecting that there is one and then identifying where it occurred so as to be able to remedy it. We take a two-step approach in our solution. First, we identify the entity (Client, Server, Network) responsible for the failure. Next, if a failure is identified as network related additional algorithms are triggered to detect the device responsible.

To achieve the first step, we revisit the question: how much can you infer about a failure using TCP statistics collected at one of the endpoints in a connection? Using an agent that captures TCP statistics at one of the end points we devise a classification algorithm that identifies the root cause of failures. Using insights derived from this classification algorithm we identify dominant TCP metrics that indicate where/why problems occur. If/when a failure is identified as a network related problem, the second step is triggered, where the algorithm uses additional information that is collected from ``failed'' connections to identify the device which resulted in the failure.

Failures are also disruptive to user's performance. Resolution may take time. Therefore, it is important to be able to shield clients from their effects as much as possible.

One option for avoiding problems resulting from failures is to rely on multiple paths (they are unlikely to go bad at the same time). The use of multiple paths involves both selecting paths (routing) and using them effectively. The second part of this thesis explores the efficacy of multipath communication in such situations.

It is expected that multi-path communications have monetary implications for the ISP's and content providers. Our solution, therefore, aims to minimize such costs to the content providers while significantly improving user performance.

Degree Type

Dissertation

Degree Name

Doctor of Philosophy (PhD)

Graduate Group

Computer and Information Science

First Advisor

Boon Thau Loo

Second Advisor
Roch Guerin

Keywords
Data Center Networks, Multipath communications, Network Diagnosis, Networking, Transport Protocols

IMPROVING NETWORK PERFORMANCE THROUGH ENDPOINT DIAGNOSIS AND MULTIPATH COMMUNICATIONS

Behnaz Arzani

A DISSERTATION
in
Computer and Information Science
Presented to the Faculties of the University of Pennsylvania in Partial
Fulfillment of the Requirements for the Degree of Doctor of Philosophy

2017

Boon Thau Loo
Associate Professor,
Computer and Information Science
Supervisor of Dissertation

Roch Guerin
Professor,
Electrical and Systems Engineering
Co-Supervisor of Dissertation

Lyle Ungar
Professor, Computer and Information Science
Graduate Group Chairperson

Dissertation Committee

Sudipto Guha (Chair), Associate Professor, Computer and Information Science

Andreas Haeberlen, Associate Professor, Computer and Information Science

Zachary Ives, Professor, Computer and Information Science

Jennifer Rexford, Gordon Y. S. Wu Professor, Computer and Information Science
(Princeton University)

IMPROVING NETWORK PERFORMANCE THROUGH ENDPOINT DIAGNOSIS AND MULTIPATH COMMUNICATIONS

Dedicated to my mom, dad, and brother.

Acknowledgments

My six years at the University of Pennsylvania have been among the most exciting and eventful times in my life. This section is an attempt to express my gratitude to all those who made this thesis possible. The completion of this thesis is owed to many people. I would like to thank my mom and dad for your love and support all these years. Mom, you gave up your dreams for a Ph.D. (three times) to be there for us when we grew up, this is for you. My utmost gratitude goes to my two advisors Dr. Boon Thau Loo and Prof. Roch Guerin. To Boon for his constant support, for not only being the best advisor but the best friend one could ever ask for, and for making sure that I had everything I needed to complete this thesis. To Roch, for giving me the opportunity to come to Penn, for his constant support and useful feedback, and for being there with advice whenever I needed help.

I would like to thank the many professors who provided useful feedback and advice: Prof. Jennifer Rexford, Dr. Andreas Heaberlen, Dr. Sudipto Guha, Prof. Zack Ives, and Dr. Vincent Liu. I would like to thank my collaborators at Microsoft and Microsoft research for their collaboration, their useful insights, and for providing the resources needed to complete this thesis. My thanks go to Dr. Selim Ciraci and Dr. Geoff Outhred for giving me the opportunity to work with them, for inspiring me throughout my Ph.D., and for their friendship. My thanks to Dr. Jitu Padhye, Dr. Hongqiang Liu, and Dr. Yibo Zhu for always having my back. In addition to those mentioned above, I am grateful to so many amazing friends

who made my study at Penn enjoyable and for helping me get through some of the harder times: Omid Alipourfard, Shohreh Shaghaghian, Farzaneh Khajouie, Sara Rahiminejad, Neda Rohani, Hoda Heidari, Mohammad Hassan Lotfi, Mehrnoush Soroush, Emad Khazraee, Majid Irae, Monia Ghobadi, Alec Koppel, Santiago Segarra, Mohammad Javad Salehi, Ang Chen, and Luiz Chamon, Salar Moarref. This work is supported by Grants: CNS-1218066, CNS-0845552, ITR-1138996, CNS-0915982, and as well as AFOSR Young Investigator Award FA9550-12-1-0327, NSF CNS-1513679, DARPA/I2O HR0011-15-C-0098 and CNS-0845552.

ABSTRACT

IMPROVING NETWORK PERFORMANCE THROUGH ENDPOINT DIAGNOSIS AND MULTIPATH COMMUNICATIONS

Behnaz Arzani

Boon Thau Loo
Roch Guerin

Components of networks, and by extension the internet can fail. It is, therefore, important to find the points of failure and resolve existing issues as quickly as possible. Resolution, however, takes time and its important to maintain high quality of service (QoS) for existing clients while it is in progress. In this work, our goal is to provide clients with means of avoiding failures if/when possible to maintain high QoS while enabling them to assist in the diagnosis process to speed up the time to recovery.

Fixing failures relies on first detecting that there is one and then identifying where it occurred so as to be able to remedy it. We take a two-step approach in our solution. First, we identify the entity (Client, Server, Network) responsible for the failure. Next, if a failure is identified as network related additional algorithms are triggered to detect the device responsible.

To achieve the first step, we revisit the question: how much can you infer about a failure using TCP statistics collected at one of the endpoints in a connection? Using an agent that captures TCP statistics at one of the end points we devise a classification algorithm that identifies the root cause of failures. Using insights derived from this classification algorithm we identify dominant TCP metrics that indicate where/why problems occur. If/when a failure is identified as a network related problem, the second step is triggered, where the algorithm uses additional information that is collected from "failed" connections to identify the device which resulted in the failure.

Failures are also disruptive to user's performance. Resolution may take time. Therefore, it is important to be able to shield clients from their effects as much as possible. One option for avoiding problems resulting from failures is to rely on multiple paths (they are unlikely to go bad at the same time). The use of multiple paths involves both selecting paths (routing) and using them effectively. The second part of this thesis explores the efficacy of multipath communication in such situations. It is expected that multi-path communications have monetary implications for the ISP's and content providers. Our solution, therefore, aims to minimize such costs to the content providers while significantly improving user performance.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Today guaranteeing good performance to users of the Internet as well as various other networks such as datacenters is still a challenge [8, 42, 119, 65]. These networks are comprised of hundreds (if not tens of thousands) of routers and switches which deliver user traffic on a best effort basis.

Failures and performance problems in such settings are not uncommon. Sources of problems include (but are not limited to) congestion in the network, server overloads, link failures, and occasionally bad configurations. ISP's attempt to deal with problems such as congestion through traffic engineering [54, 15]. Datacenter operators have devised various diagnosis and debugging tools to minimize the mean time to recovery [119, 65, 145, 17, 5]. However, none of these solutions are completely effective (especially in providing real-time solutions). The traffic engineering problem for traditional network protocols is NP hard [54]. Furthermore, in both datacenters and the Internet, the network is typically comprised of a number of different corporations and organizations. These organizations do not always have an incentive to cooperate and to share information across their respective boundaries. As a simple example, while BGP allows for ASs to advertise their path preferences to their neighbors (through MED values), most ASs ignore such specifications when deciding on the best path for their traffic. This makes global traffic engineering difficult, if not impossible. In datacenters, the users belonging to one organization may access a different organizations service through the datacenter's network, rendering end to end monitoring and visibility difficult. On some occasions, these entities also provide competitive services to the users which limit their willingness to cooperate [101]. Therefore, tools and methodologies that depend on such information sharing typically are inadequate.

It is for these reasons that we believe it is important to empower the end users themselves to be able to circumvent congestion occurrences and failures when they occur, as well as to assist in the diagnosis process in the event of failures as they are able to indirectly observe the entire communication path. Doing so requires solving a number of different challenges. To enable users (clients) to assist in the diagnosis process, effective monitoring tools are required that enable them to reason about the network and the service without direct visibility into these systems. Furthermore, such monitoring tools would need to be extremely lightweight as they should coexist with the client without disrupting its "normal" behavior. Meanwhile, there are a number of different approaches that can be used to enable clients to maintain high quality of service (QoS) when failures occur. One example is the use of diversity coding [16] which augment the data transmitted from the end host with additional redundancy to enable recovery from potential losses. However, the ability of such approaches to cope with high loss rates is limited. A different approach would be to allow clients to make use of the path diversity inherently present in today's networks (both datacenters and the Internet) as such path are unlikely to fail at the exact same time. By doing so clients can route around such failures if and when they occur. Doing so requires clients to detect/quantify path quality and to quickly and effectively migrate traffic onto healthy path when failures occur. Furthermore, there are cost implications to the ISPs and content providers that need to be accounted for. For example, in this work, we will show that multipath clients tend to result in an increase in cost for content providers. However, with careful planning, such impacts can be largely controlled and mitigated.

Our contributions in this work are to address the afore mentioned challenges. Specifically:

- **Endpoint Diagnosis.** We attempt to allow clients to assist in the diagnosis process. We take a two-step approach to solving this problem: a) we identify the entity responsible for the failure through a system we call NetPoirot. NetPoirot answers the following question: how much can you infer about a failure in the data center using TCP statistics collected at one of the endpoints? Using an agent that captures TCP statistics it creates a classification algorithm that identifies the root cause of failures using this information at a single endpoint. Our evaluations on a real production datacenter show that it is highly accurate in identifying the entity responsible for the failure. b) In many cases, the entity identified by NetPoirot as the cause of the failure is the network. The network itself, however, is comprised of many devices and machines. Through a system named Vigil, we identify which of these devices was responsible for each failure instance. Vigil is a lightweight always on monitoring tool that is completely contained within the end hosts. During its one-month deployment in a tier-1 data center, it has detected every problem identified by other previously deployed monitoring services while also finding the sources of other problems. The system is effective in detecting drop rates as low as 0.05%.

- **Endpoint resilience through multipath communications.** We explore the efficacy of the Multipath TCP transport protocol in allowing clients to circumvent failures by going through multiple paths. Such solutions have been extensively studied in the context of datacenters and ISPs e.g., [9, 105]. However, their implications in many contexts, especially when used for applications such as video delivery, is not yet completely understood. We show in this work that the current state of the art multipath transport protocol [105] suffers from unintended consequences that could have severe implications those using it. We further illustrate why such multipath approaches typically result in an increase in peering cost when used for applications such as video delivery. We then present a new solution in this context which attempts to remove all such undesirable behaviors and that effectively improves user performance.

The main contribution of this work is, therefore, in extending current approaches for both diagnosis and resilience in a way that they can be used at the endpoints, without any support from the network operators, and without negatively affecting the other parties involved in the communication process.

Chapter 2

Finding The Entity Responsible For The Failure

2.1 Introduction

Components of networks, and by extension the internet can fail. We can deal with these failures in two ways. We can either try finding the points of failure and fixing them, or we try avoiding the failures and route around them. The first two chapters of this thesis are devoted to the first of these two solutions.

Fixing failures relies on first detecting that there is one and then identifying where it occurred so as to be able to remedy it. We take a two step approach in our solution. First, we identify the entity (Client, Server, Network) responsible for the failure. Next, if a failure is identified as network related additional algorithms are triggered to detect the device responsible. This approach allows for a targeted investigation whereby additional monitoring/diagnosis algorithms are only initiated when necessary, thus significantly reducing the diagnosis overhead. Furthermore, such a design significantly reduces the amount of cooperation required between the entities maintaining the client, server, and the network in the diagnosis process.

The first step is done through a system we refer to as NetPoirot. NetPoirot relies on TCP metrics captured from one of the endpoints involved in a TCP connection to determine which entity is responsible for the failure. Our evaluations of NetPoirot in a production data center shows that it is highly accurate (with Recall and Percision as high as 97%). Our results also indicate that 17% of the failures observed and diagnosed by NetPoirot were network related. This introduces the need for additional steps in identifying the device (in the network) that is responsible for such failures. We do this through a system we refer to as Vigil. This chapter is devoted to the description of NetPoirot. We defer the description of Vigil to the next chapter.

2.2 Motivation

In today's data centers, a common issue faced by operators is troubleshooting faults in complex services. It is often unclear whether the cause of performance bottlenecks lies in the underlying network, client, or service-level application code/-machine. Often times, the knee-jerk reaction is to first blame the network whenever performance issues surface.

The problem is exacerbated by two issues. First, the parties involved in diagnosing the errors (e.g. the service DevOps engineers, network operators) may operate as different entities either within or across companies and each party may lack easy access to the other's performance/debugging logs. This significantly increases debugging time and extends the mean time to recovery. Second, these failures are sometimes intermittent and non-deterministic, and hence difficult to reproduce without high fidelity, always on, monitoring probes in place throughout the entire infrastructure.

To highlight these challenges, we present a real-world example seen within a production cloud. The public cloud offering is used by over 1 billion customers, whose applications reside within VMs in its data centers. In one particular scenario, the VM triggers an operation within the hypervisor, that requires the hypervisor to send a request to a remote service. Whenever the request/response latency increases (due to remote service failure, overload, or network slowdowns), an error occurs in the hypervisor which in turn causes the VM to panic and reboot, hence disrupting normal operations. We refer to this as "Event X or EX".

EX occurs intermittently and only on a subset of nodes at any given time. However, its occurrence is sufficiently frequent to degrade user experience. It is also unclear when a reboot happens, whether the error is caused by a remote service issue (overload, server failure), or a network problem (e.g. packet drops or router misconfigurations). In one example, when EX occurred the remote service DevOps team was first contacted. They suspected that EX may be occurring due to high request/response latencies, and blamed the network, and passed on the diagnostics to the network engineers. The network engineers observed normal RTT times in TCP traces at the time of failure and suspected that the problem is probably due to slow server responses, and handed over the issue back to the server operations team. The iterations continued until the various teams involved pieced together the sequence of events that led to the service disruption. The entire process is time-consuming, expensive, and involves many engineers and developers spanning different organizations. To make matters worse, while the symptoms of EX may be the same, the actual cause may differ across occurrence, and fixing one occurrence of EX may not prevent others from happening in the future. Note that such lengthy debugging across multiple subsystems is an issue not unique to this production cloud, and has been reported elsewhere as well [132, 65].

While many network diagnosis tools have been developed in the past, all of them come short in some way when solving problems such as EX. Some require access to the entire system [138], others are too heavyweight and so cannot be used in an always on fashion [140, 143], and finally some require information that the service/network are not willing to share [17, 138]. Many of the proposed concepts in these tools have been adopted by the afore mentioned organization and their failure in assisting its engineers to diagnose EX is itself a testament to the persistent need for a more effective tool.

In the ideal case, one would like to quickly pinpoint the *most likely* source of the failure within the infrastructure. The team (client, network, or remote service) that owns the failure can then provide a timely response, rather than have the error be passed around within organizations.

To achieve this goal, we propose NetPoirot [1] that aims to perform such failure attribution quickly and accurately. NetPoirot requires only a TCP measurement agent to be deployed at each client VM in the cloud. The key insight of NetPoirot is that different types of failures, albeit not network related, will cause TCP to react differently. For example, a slow reading remote service results in exhaustion of the TCP receive window on the sender VM, which itself triggers TCP zero window probing. Packet drops on a router result in an increase in the number of duplicate ACKs. High CPU load on the client result in fewer transmission opportunities for the client application and thus fewer data sent by TCP. These differences are not always easy to define, given the high correlation between the various TCP metrics.

Specifically, NetPoirot makes the following contributions:

Lightweight continuous monitoring. We develop a TCP monitoring agent that runs on all client machines within our data centers. These machines request services within and across data centers under our administration. The agent captures various TCP related metrics periodically. It is implemented in Windows, although a similar tool in Linux can be supported. Unlike more heavyweight approaches

[1] Named after Agatha Christie's famous detective Hercule Poirot.

that require packet captures, NetPoirot 's agent is lightweight and non-intrusive, requiring only aggregate TCP statistics to be periodically collected and measured. At runtime, the agent requires only 132 numeric values be examined (and then discarded) by the client machine every 30 seconds.

Machine-learning based classification. Using data collected from our agents, we study the extent in which TCP statistics can be used to distinguish various types of failures in data centers using decision-tree based supervised learning algorithms. We then identify the key parameters in TCP that are most representative of each type of failure in data centers.

Our approach is a significant improvement over techniques such as SNAP [138]. SNAP requires *full* knowledge of the data center topology, machine configuration, and application to server mapping. Furthermore, it manually reasons about each type of failure and devises a rule for each problem accordingly. Such manual inspections are limited in scope and are prone to human error. NetPoirot uses machine learning instead, which allows us to uncover more complex insights. For example, our use of decision trees allows us to identify the dominant set of TCP metrics that help to classify each failure. Furthermore, NetPoirot does not require any knowledge of the topology, or client to server mappings.

NetPoirot implementation. We have designed and developed a proof-of-concept implementation of NetPoirot that combines the above two ideas and is used for diagnosis at runtime. NetPoirot consists of a training phase, where a variety of faults are injected onto training VMs, in order to produce a diagnosis function as output. This function is then distributed to all machines in the data center to be used at runtime to identify the source of failures. NetPoirot can be used by both *customers* and *data center operators* as it does not require any information from the network or service.

NetPoirot evaluation. We perform an extensive evaluation of NetPoirot to validate its effectiveness. We evaluate the worst case performance of NetPoirot using data collected over a 6 month period in a production data center that hosts over 170K web servers and transfers 10Tbps of traffic. We have induced 12 different common types of failures in the communication of two different applications running on 30 different machines and observed the changes in TCP statistics collected from these machines. NetPoirot performs coarse-grained blame allocation on this data with high accuracy (96% for some failure types). NetPoirot 's accuracy improves even further if its input information is augmented with additional information from the endpoint on which it resides. Our monitoring agent has currently been deployed on all the compute nodes within our data centers. We show that NetPoirot can accurately identify the entity responsible for a variety of failures.

The tradeoff that NetPoirot makes is that while it is able to determine the entity (network, client, remote service) most responsible for the failure, it does not pinpoint the exact physical device or the specific problem type (e.g. high CPU load on the server vs high I/O load on the server). We argue that this is a worthwhile tradeoff because identifying the entity is often times the operationally most time-consuming (and hence expensive) part of failure detection, e.g. EX failures take from 1 hour (high severity) to days to diagnose. In fact, the development of NetPoirot was commissioned as a direct result of this problem in the case of EX.

While we do not claim that we can diagnose *all* potential problems, our results indicate that it is possible to distinguish between a number of common network, server, and client failures with high accuracy. This is significant, as we do not need to use per-flow TCP metrics, nor do we need data from both endpoints. Since only aggregate metrics are collected, our methodology allows us to maintain some extent of customer privacy.

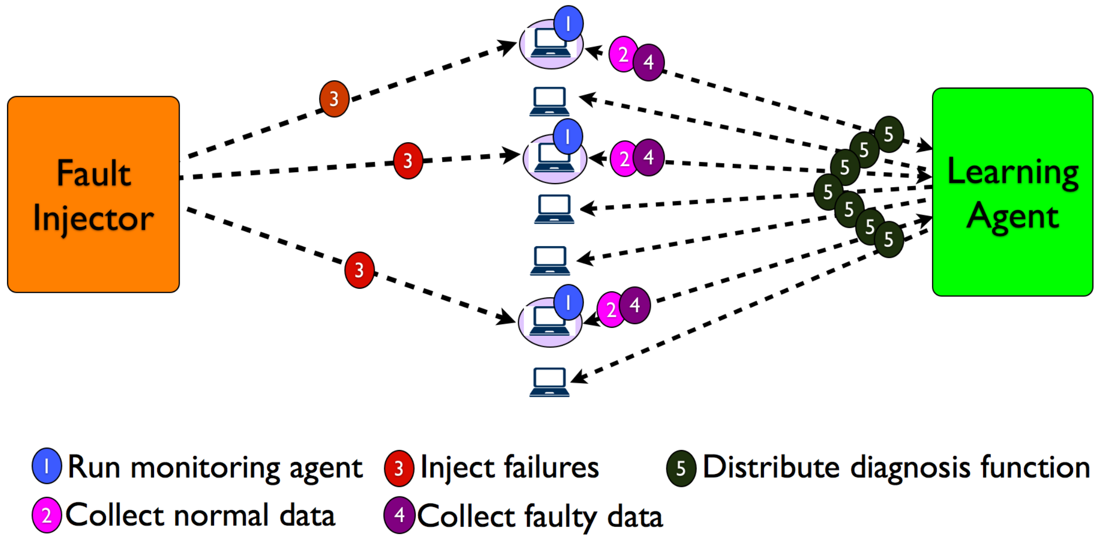


Figure 2.1: Overview of NetPoirot

2.3 Overview

We first begin with an overview of NetPoirot , which consists of two phases: *training* and *runtime*.

Training phase. Fig. 3.2 depicts the training phase. The training phase occurs in the production environment on a subset of machines, termed as the *training nodes*. During this phase, a TCP *monitoring agent* is installed on these nodes. Details on the monitoring agent are presented in Section 2.4. The agent can be either installed at the hypervisor or within individual VMs hosted by the training nodes. When there are no failures, agent statistics are periodically sent to the learning agent to capture the behavior of a non-fault scenario. From time to time, we use a *fault injector* to inject a variety of failures listed in Table 2.1. We define the “client” as the local machine that is communicating with a remote service, which we refer to as the “server”. Failures range from increased CPU load, increased memory load, increased storage load (by generating excess I/Os on either the client or server using

General label	**Failure**	**How it is induced**
Server	High CPU load on server	Application running kernel level CPU intensive operations
	Slow reading server	Modified the server application, added a random delay to its read operation
	High I/O on server	SQLIO [92]
	High memory load on server	Testlimit [93]
Client	High CPU load on client	Application running kernel level CPU intensive operations
	High I/O on client	SQLIO l [92]
	High memory load on client	Testlimit [93]
Network	Bandwidth throttling	Added rules in the A10's
	Sporadic packet drops	SoftPerfect [4]
	Packet reordering	SoftPerfect [4]
	Random connection drops	NEWT [94]
	High Latency	SoftPerfect [4]

Table 2.1: Table describing the types of faults we induced in our system for training.

SQLIO), and also various forms of network-related problems such as throttling or packet drops.

For each injected failure, the monitoring agent's TCP statistics are collected and then used by the learning agent to create a *diagnosis function*, trained using as the label, either the actual failure itself or its type (e.g. client, remote, network). The diagnosis function is the output of our supervised learning algorithm. The input to the function is the latest TCP statistics read from the end point, and the output is the entity most likely to be responsible for the failure. This function is broadcast to all VMs at the end of the training phase.

Runtime phase. At run time, all monitored nodes run the monitoring agent. Whenever failures are detected, each VM invokes the diagnosis function using digests collected locally and then generate as output the likely source of failure.

2.4 Description of NetPoirot

In this section, we describe the various components of NetPoirot . NetPoirot is comprised of a monitoring agent that collects data that is then input to a learning agent. The learning agent uses this information to create a diagnosis function that is used at runtime. The following describes the details of each of these components in more detail.

2.4.1 Monitoring Agent

Our TCP monitoring agent is installed at each machine's hypervisor or within individual VMs. The installation is limited to only client machines communicating with various remote services within/across data centers. For example, if the remote service is storage, this precludes the need to run the agent on storage servers. The agent collects TCP statistics for all connections seen on its monitored node. Given that our implementation is based on Windows, we will describe the agent based on Windows terminology. These statistics can also be collected in a Linux-based system.

The agent is implemented using Windows ETW events [91], a publish-subscribe messaging system in the Windows OS. A TCP ETW event is triggered every time a TCP related event, e.g. the arrival of a duplicate ACK occurs on any one of the connections currently active in the OS. The agent collects and aggregates events at the granularity of *epochs* so as to minimize bandwidth/storage overhead during training. Within every epoch, it receives ETW events, extracts relevant features and stores them in a hash table based on TCP's $5-$tuple. At the end of an epoch, the TCP metrics that depend on the transmission rate are normalized by the number of bytes posted by the application in that epoch. The normalized metrics are marked in Table 2.2. Each individual metric is then further aggregated by calculat-

metric	**statistics calculated**	**abbreviation**
Number of flows	$\mathcal{R}$	NumFlows
Maximum congestion window in δ	$\mathcal{S}$	MCWND
The change in congestion window in δ	$\mathcal{S}(*)$	DCWND
The last congestion window observed in δ	$\mathcal{S}(*)$	LCWND
The last advertised (remote)receive window observed in δ	$\mathcal{S}(*)$	LRWND
The change in (remote) receive window observed in δ	$\mathcal{S}$ (*)	DRWND
Maximum smooth RTT estimate observed in δ	$\mathcal{S}$ (*)	MRTT
Sum of the smooth RTT estimates observed in δ	$\mathcal{S}$ (*)	SumRTT
Number of smooth RTT estimates observed in δ	$\mathcal{S}$ (*)	NumRTT
Duration in which connection has been open	$\mathcal{S}$	Duration
Fraction of open connections	$\mathcal{R}$	FracOpen
Fraction of connection closed	$\mathcal{R}$	FracClosed
Fraction of connections newly opened	$\mathcal{R}$	FracNew
Number of duplicate ACKs	$\mathcal{S}$ (*)	DupAcks
Number of triple duplicate ACKs	$\mathcal{S}$ (*)	TDupAcks
Number of timeouts	$\mathcal{S}$ (*)	Timeouts
Number of resets	$\mathcal{S}$ (*)	RSTs
Time spent in zero window probing	$\mathcal{S}$	Probing
Error codes observed by the socket	$\mathcal{R}$	Error Code
Number of bytes posted by the application	$\mathcal{S}$	BPosted
Number of bytes sent by TCP	$\mathcal{S}$	BSent
Number of bytes received by TCP	$\mathcal{S}$ (*)	BReceived
Number of bytes delivered to the application	$\mathcal{S}$ (*)	BDelivered
Ratio of the number of bytes posted by the application to the number of bytes sent	$\mathcal{S}$	BPostedToBSent
Ratio of the number of bytes received by TCP to the number of bytes delivered	$\mathcal{S}$	BReceivedToBDelivered

Table 2.2: Features captured by the monitoring agent during each epoch. We use $\mathcal{R}$ to show that the raw value of a feature is captured and $\mathcal{S}$ to show that we capture the statistics of that feature. (*) indicates normalized metrics.

ing its mean, standard deviation, min, max, 10^{th}, 50^{th}, and 95^{th} percentile across all TCP connections going to the same destination IP/Port.

We assume that identical failures happen within a single epoch, e.g. if a connection experiences failure A, then all other connections between the same endpoints in the same epoch either experience no failure, or also experience failure A. Therefore, the epoch duration needs to be carefully tuned. Small epochs increase monitoring overhead, but large epochs run the risk that sporadic failures of different types will occur within one epoch, affecting the accuracy of the learning algorithm. We currently use an epoch of 30s. Fine tuning the epoch duration is part of our future work.

Table 2.2 shows the features maintained within an epoch by the monitoring agent. Our aggregation method reduces the amount of bandwidth required on the machines in the training stage[2] and has the added benefit of hiding the clients exact transmission patterns. Furthermore, when applications change their transmission pattern across connections in reaction to failures it allows for this change to be detected. In the other extreme, one may decide to use per connection statistics with more overhead but with the benefit of detecting why each individual connection has failed separately.

The agent imposes low runtime overheads. Based on our benchmarks, even in the absence of aggregation, when processing $500,000$ events per second, each agent uses 4% CPU load on an 8 core machine and less than 20 MB in memory.

2.4.2 Learning Agent

During the training phase, the learning agent takes as input TCP metrics gathered by monitoring agents on training nodes. At run time, it distributes the learned

[2] Without aggregation, the client needs to transmit $31n$ features every epoch to the learning agent where n is the number of connections during that epoch. With aggregation, this number is reduced to 130.

model to all clients to be used for diagnosis. The model has to quickly classify epochs with the appropriate labels to indicate whether it is a remote (Server), local (Client), or Network issue.

The learning agent uses decision trees as its classification model. In a decision tree, each internal node conducts a test on an attribute, each branch represents the outcome of the test, and the leaf nodes represent the *class labels*. The paths from root to leaf represents the classification rules.

In our setting, the internal nodes correspond to one of the aggregated TCP metrics being monitored. The learning phase determines the structure of the decision tree, in terms of the choice of attributes and the order in which they are used for testing along the path from the root to label (this ordering is determined by the information gain of features in the dataset). The specific nature of the test at each node, i.e. the inequality tests, is also determined in this phase.

As noted by prior work [36, 6, 49], the structure of decision trees allows for further understanding of the attributes that identify each failure. For this reason, we found decision trees more attractive to use than other machine learning approaches. We will elaborate further on this in Section 2.5.

Fig. 2.2 shows an example of a decision tree, that distinguishes packet reordering from normal data. Leaf colors in the figure represent the labels of the training data that ended up in those leaves. Most leaves are ”pure”, i.e. all the data in those nodes have the same label. leaf 2 shows an ”impure” leaf that has a mix of both labels. In such situations, the tree picks the majority label in the leaf as its diagnosis.

Based on the concept of decision trees, our learning agent requires three enhancements for improved stability and accuracy:

Random forests. Our learning agent uses an enhanced type of decision tree, known as *random forests* [26][3]. In random forests, multiple decision trees are generated

[3] Version 4.6-10 in R version 3.2.1.

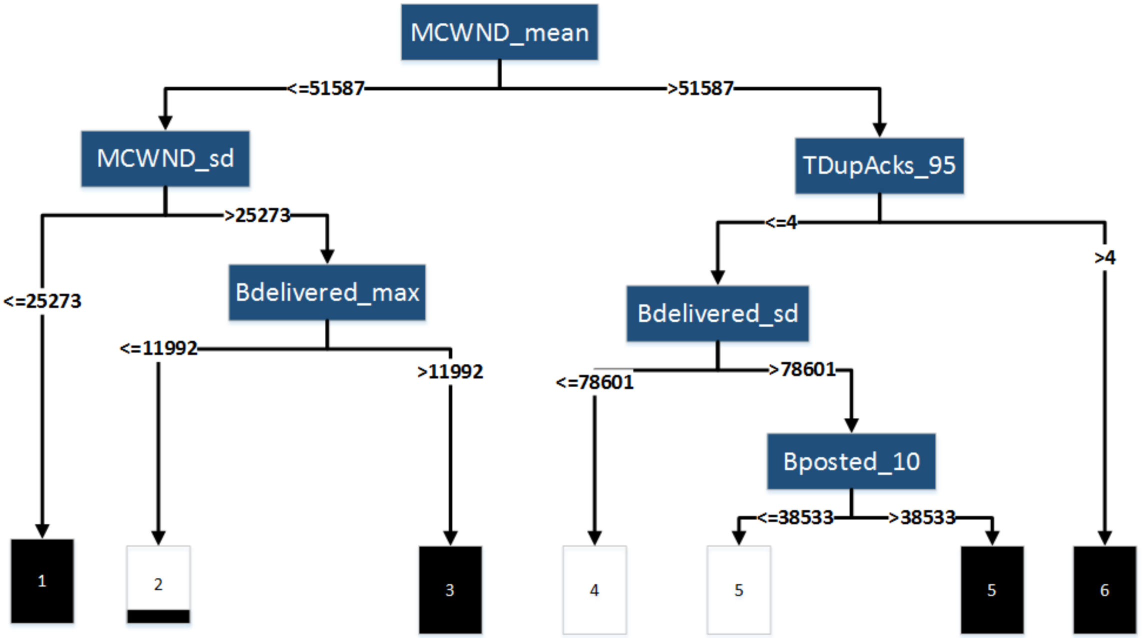


Figure 2.2: Example tree. The white/Black leave colors illustrate the labels of the training data that end up in that leaf.

from different subsets of the data, and the classification decision is majority-based, where a majority is defined based on a cutoff fraction specified by the user. For example, a cut-off of $(0.2, 0.8)$ indicates that for class 1 to be chosen as the label, at least, 20% of the trees in the forest need to output 1 as the label as well. Random forests improve stability and accuracy in the face of differences in machine characteristics and outliers.

Multi-round classification. To improve accuracy, we do *rounds* of classifications. First, the forest is trained to classify Network failures only. The Server and Client failures in the training set are labeled as non-faulty (Normal) in this phase. Next, the Network failure data is removed from the training set, and a new forest is trained to find Server failures with Client failures labeled as Normal. Finally, the Server data is also removed and a forest is trained to identify client-side failures. At run time, data is first passed through the first forest, if classified as Network, the process terminates. If it is classified as Normal, it is passed through the second forest. Again, if it is classified as Server failure the process terminates. If not, the data is passed through the third forest and is assigned a label of Normal/Client. In machine learning, such multi-round classifications are referred to as *tournaments*. In traditional tournaments, different decision trees are used in pair-wise competitions. Our tournament strategy is a modification of standard tournaments, as they did not work well in our setting.

Per-application training. Applications react to failures differently. One application may choose to open more TCP connections when its attempts on existing connections fail while others may keep retrying on the ones currently open. Some form of normalization, such as that we use for the monitoring agent, helps avoid dependence on the transmission rate of the client itself. However, it does not help avoid this particular problem given that the effects of application behavior go beyond the transmission rate but also influence the number of connections, their duration, etc. Indeed, these behaviors themselves improve NetPoirot 's accuracy as they provide more information about the failure. Hence, it is advised to train NetPoirot for each application separately. We argue that unless applications change drastically on a daily basis, there is sufficient time in between major application code releases and deploys for the model to be updated.

Two-phase tree construction with cross-validation. Each forest is constructed in two phases. First, given the training set, we determine basic parameters of the forest, e.g., its cutoff value and a minimum number of data points required in a leaf node. The latter is required to bound the tree sizes and to avoid overfitting. Once these parameters are determined, the training set is used to create the actual forest.

One of the pitfalls of any machine learning algorithm is the danger of *overfitting*, where the trained model is tailored to explicitly fit the subsequent testing set. This leads to poor future predictive performance. To avoid overfitting, we apply a standard machine learning technique, namely a modified variant of *cross validation*

(CV). In a nutshell, the first phase is accomplished using *N-fold* [28] CV which estimates error using subsets of the training data, while the second phase builds the model using all the training data.

In the classic form of N-fold CV, the training data is randomly divided into N subsets (folds). Iterating over all folds i, in each iteration i is omitted from the training set and the model is trained over the remaining $N - 1$ folds. The trained model is then tested using the i^{th} fold. The estimated errors in each iteration are averaged to provide an estimate of the model's accuracy. N-Fold cross-validation, however, if used on our data set runs the danger of overestimating accuracy as models will learn *specific* machine/network characteristics when data from one machine *leaks* between folds. Therefore, we divide data from each machine into its own unique fold. We define *CV error* as the average error of CV when each fold contains data from a single machine.

To show why this is important, we tested cross-validation on our data set using both methods[4]. Using vanilla cross-validation, we observe an error of 1.5%. However, when we partition the data based on machine label, we get 10.55% error. This further indicates that if data from the same machine is used for both training and at runtime one may get much higher predictive accuracy than those reported in this paper.

Normalization. We normalize TCP statistics that depend on the data being sent. Namely, features marked with $(*)$ in Table 2.2 were divided by the number of bytes posted by the application in that epoch in order to minimize dependency on the application's transmission pattern.

[4] This was done without the use of tournaments.

2.5 TCP Behavior under Faults

The hypothesis behind NetPoirot is that the different types of failures, albeit not network related, cause TCP to react differently. To study this further, we examine the changes in various TCP parameters in the presence of different failures. Decision trees allow for not only classifying faults, but also illuminating features (in this case TCP metrics) affected by each failure. It is this basis that allows us to develop NetPoirot and why we use decision trees in this section.

The algorithm used in NetPoirot is agnostic to the choice of decision tree algorithm as we use random forests which rely on weak classifiers as their basis. With regards to the results in this section, we experimented with different types of decision trees, and will present our results based on a decision tree algorithm called *C5.0* [83]. We used the 0.1.0.24 version of C5.0 in the 3.2.1 version of R. C5.0 is based on *information gain* and aims to greedily reduce the amount of uncertainty with respect to the data's label. Nodes higher up the tree provide the most information gain (the most reduction in uncertainty) with respect to the output.

For each type of failure, we train a decision tree from data that includes that failure as well as Normal data. For each failure, we select the top three TCP metrics (features) in the tree. For each of these top features, we also measure the correlation between its value and the actual ground truth. This is done by computing the Pearson correlation (PC) between each feature value and the corresponding ground truth label (encoded as 0 for normal, and 1 for faulty) when that feature value was recorded. The PC value is then computed across all epochs for the duration in which the failure occurs.

PC is a measure of the *linear* correlation between the feature values and the labels (failure type/Normal) and provides further insight into the level of (linear) dependency between the features. PC's value ranges between $(-1, 1)$. The closer the absolute PC is to 1, the higher the linear correlation and therefore the easier it is to identify such correlations.

While PC is not required by NetPoirot for classification, we calculate the PC value so as to provide insight into how easy it would be for an operator to simply decipher the relationship between the failure and the metrics used by the decision tree. The human brain is relatively good at identifying linear relationships between variables. Our results show that often times, the relationship is non-linear, meaning that a manual classification approach is not likely to work.

2.5.1 Fault Injection and Application Workloads

Failure injection. Supervised learning requires labeled data for training. In order to train our model, we injected failures in the communication pattern of two different types of applications running on a subset(30) of the VMs in four of our production data centers located in West and Central USA, North and West Europe. Our network serves over 1 billion customers and handles petabytes of traffic per day. Given that networks in a data center environment are highly symmetric, training on a small subset of machines in a given cluster is sufficient for high accuracies at runtime. Some data centers already purposefully inject faults on a regular basis in their production environment in order to evaluate their degree of fault tolerance [129]. The traffic of the nodes in the path of these failures can be observed for training purposes.

Application workload. Applications are designed with a degree of fault tolerance, and, therefore, react to any failures that might occur. To isolate the impact of failures on TCP behavior (and not applications), we experiment using two applications that were designed to remain passive when failures occur. The first is a *duplex* application, where we recorded a hypervisor's communication to a remote service (the one that causes EX) for 6 hours. This trace is then replayed to and from the server. There is no fault tolerance logic in the application. The second application is a *simplex* application that opens 128 connections[5] to a server and sends a constant number of bytes on each connection every second. We use three sending rates: 100, 500, 10000 Bps.

These applications are designed to capture the *worst case* performance of NetPoirot . For example, the simplex application involves communication in one direction. Therefore, some metrics that depend on communication from the server back to the client would not be captured. The duplex application, on the other hand, is an extreme application that does not react to failures. Overall, the simplex application results in decision trees with higher CV errors (Section 2.4.2) as a result of fewer metrics being influenced by the failure.

Our data is gathered over a period of 6 months (July-December 2015). All datasets are labeled with the corresponding machine ID where the data is collected, to be used for cross-validation (CV) described in Section 2.4.2.

2.5.2 Results

We inject failures listed in Table 2.1 to study their impact on TCP. Tables 2.3 and 2.4 summarize our main findings. Each row corresponds to the top three features selected by a decision tree trained on the input dataset. For each feature, we include in parenthesis () the corresponding PC value as described earlier. We also measure the CV error, which represents the accuracy of the decision tree. For the purpose of this section, we do not apply the normalization described in Section 2.4.2 in order to gain more visibility into the direct impact of TCP on the raw metrics.

We observe that each type of failure can be defined succinctly using a few features. To validate this, we used a standard machine learning technique, *Principal Component Analysis (PCA)* [111], where we identify the highest Eigenvalues of the

[5] This number is based on the number of connections opened by clients connecting to one of our services.

General label	**Fault**	**Features Selected**	**CV Err**

Server	High CPU load on server	Probingsd(PC=0.27)
		Timeouts95(PC=0.009)
		Duration50(PC=0.22)
			6%

	Slow reading server	LCWNDmin(PC=0.78)
		Duration50(PC=0.21)
		Durationmax(PC=0.15)
			3%

	High I/O on server	BDelivered95(PC=0.58)
		BReceivedmax(PC=−0.72)
			0.2%

	High memory load on server	BDeliveredTOBReceived(PC=0.01)
		BReceived(PC=−0.74)
			0.1%

Client	High CPU load on client	Duration95(PC=−0.33)
		Duration50(PC=−0.03)
		Durationmin(PC=0.4)
			0.6%

	High I/O on client	BReceivedmax(PC=−0.75)
		Probingsd(PC=−0.61)
		BDeliveredToBReceived95(PC=0.01)
			1.7%

	High memory load on client	BReceivedmax(PC=−0.82)
		LCWND10(PC=0.24)
		BDelivered95(PC=0.3)
			15%

Network	Bandwidth throttling	BReceivedmax(PC=−0.78)
		Durationsd(PC=−0.14)
		BDelivered95(PC=0.26)
			0.05%

	Sporadic packet drops	BReceivedmax(−0.73)
		MRTTmax(0.12)
		BDelivered10(0.34)
			3%

	Packet reordering	MCWNDmean(PC=−0.07)
		MCWNDsd(PC=0.01)
		TDupAcks95(PC=0.79)
			0.1%

	Latency	MRTTmean (PC=0.91)	0

Table 2.3: Important features in identifying failures for the **d**uplex application. Statistics proceed the variable name.

dataset for each failure type. The sum of the Eigenvalues of a dataset equals its variance. Interestingly, for almost all failures, the sum of the 2 highest Eigenvalues captured more than 95% of the variance in the feature set. This is important as it shows that the space of each failure (as represented by TCP metrics) is compactly representable on two dimensions[6].

We next discuss interesting highlights of our analysis.

High CPU load on the server. This failure was induced by a multithreaded program, where each thread performed a CPU intensive system level operation in a loop. For the duplex application, the algorithm used the standard deviation of the time spent in zero window probing as its top feature. Zero window probing oc-

[6] These dimensions can be derived from the original features using PCA.

curs on the client when the server runs out of receive buffer. In the presence of high CPU load, the server would not read from the receive buffer as regularly as normal operations resulting in higher variance in the size of the receive buffer and by extension the time spent in zero window probing at the client.

Slow reading server Detecting a slow reading server should be simple in principle if it results in zero window probing. We induced a random delay of 0-100ms before the server reads from the TCP socket in order to test this theory. Surprisingly, the decision trees use measures of the congestion window and duration instead. It seems that the secondary effect of the delay was more pronounced on these metrics.

High I/O load on the client side. To induce high I/O load, we use the SQLIO tool [92] from Microsoft. High I/O load on the client has all the markings of an application limited connection. The decision trees both use the maximum of BReceived, the standard deviation of the time spent in zero window probing, as well as the 95^{th} percentile of the ratio of BDeliveredToBReceived.

High memory usage on the server. The selected values point directly towards a problem on the server, these values include the 95^{th} percentile of BDeliveredToBReceived and the maximum of BDelivered. This shows that the reduction in memory on the server side has caused an impact on the amount of data transmitted from the server. Note, that such data is not available on the simplex application. Here, instead the decision tree relies on the number of connections, as well as the congestion window related metrics in order to do classification.

High memory usage on client. As part of our analysis, for all the failures, we used fast correlation-based filters for feature selection and compared the results with the top 3 features of the decision tree. We found that feature selection for this particular failure returned an unexpected subset of the features. It returned the mean time spent in zero window probing and its standard deviation. On all machines, the client had zero Probingmean when the memory usage on the client

General label	Fault	Features Selected	CV Err.
Server	High CPU load on server	MRTT10(PC=0.87), MRTTsd(PC=−0.12), Timeoutssd(PC=0.03)	1.2%
	Slow reading server	NumberofFlows(PC=−0.22), DCWNDsd(PC=0.16), LCWND95(PC=0.37)	13%
	High I/O on server	NumberofFlows(PC=−0.08), MCWNDmax(PC=0.11), LCWNDmax(PC=0.46)	14%
	High memory load on server	NumberofFlows(PC=−0.04), DCWNDsd(PC=0.08), LCWNDmax(PC=0.49)	16%
Client	High CPU load on client	MCWND10(PC=−0.07), BpostedtoSentmin(PC=−0.05)	0%
	High I/O on client	NumberofFlows (PC=0.32), MCWNDmean(PC=−0.31)	15%
	High memory load on client	NumberofFlows (PC=0.57), MRTTmax(PC=−0.47)	13%
Network	Bandwidth throttling	NLossRecovery95(PC=0.66)	0.02%
	Sporadic packet drops	MCWND95(PC=−0.47), BPostedmean(PC=0.24), Durationmin(PC=−0.07)	4%
	Latency	MRTTmax(PC=0.94)	0.02%

Table 2.4: The important features for identifying each type of failure in the **s**implex application.

was increased, whereas it was positive in the normal data. The client, having less memory, is pushing less data to the server allowing it to "keep up".

Packet drops. Using a commercial tool called SoftPerfect [4], we induce 5, 10, and 30 percent packet drop rates on all connections to the service. Metrics pointing to TCP throughput are those mostly used to identify the failure. While packet drops do result in a decrease in throughput it is surprising that DupAcks are not the most prominent metric. Indeed, the maximum number of BReceived has twice as high information gain than any of the DupAck statistics. We plotted the CDF's for both these metrics. Both showed a significant difference between failure/normal data. And therefore, we can only explain this choice by noting that the impact of BRecieved had a more pronounced effect on information gain compared to the number of DupAcks.

Connection drops. This type of failure is one of the easiest for our classification tool to identify as tracking SYN, SYN/ACK ETW events suffice in understanding whether all or only a random subset of connections are being dropped. Thus, these failures are identified with 100% accuracy. With clear indicators, that identify why they occur.

Overall takeaways. TCP reacts differently to different failures. Therefore, the top three features selected by the decision tree also vary across failures. PC values are not a good predictor of the importance of TCP metrics. In fact, the top three features on occasion have low absolute PC values, suggesting that the relationship between faults and TCP metrics may be more complex than a straightforward linear one. This makes manual classification difficult and motivates the need for automated approaches.

2.6 Evaluation

We have developed a prototype of NetPoirot , which we deployed in a production data center with failure injection and application workloads as described in Section 2.5.1. The focus of our evaluation is to measure the accuracy of NetPoirot in identifying each failure type. To this end, we compute the *confusion matrix* [27] of the test set on the trained model. A confusion matrix illustrates what each class of failure in the test set is classified as by NetPoirot .

Within each class, we further report the precision and recall, defined as follows:

Precision is defined as the ratio of true positives divided by the sum of true positives and false positives. It is a measure of reliability. For example, if the precision of the network failures are reported as 96%, it means that when NetPoirot allocates responsibility to the network for a failure it is the culprit with 96% likelihood.

Recall is the ratio of true positives to the actual number of instances in a class and is a measure of NetPoirot 's ability to recognize that an entity is indeed responsible for a failure.

We will first describe the performance of NetPoirot when only TCP metrics from the client side are used. We then show that by augmenting network information with high-level counters, e.g. CPU load, on the *same client machine* one can achieve almost perfect classification.

We use the workload described in Section 2.5 and partition the measured data into two sets for training and testing. The partition is done by using disjoint sets of machines for the training and test sets in order to avoid any bias in favor of NetPoirot . The datasets used for training and testing are roughly the same size and contain aggregated information for over 37 million connections.

General label	**Fault**	$P(\textbf{fault} \mid \textbf{error} \cap \textbf{General label})$	$P(\textbf{error} \mid \textbf{fault})$
Server	High CPU load on sever	66.36%	24%
	Slow reading server	2.81%	4.06%
	High I/O on server	6.36%	9.66%
	High memory load on server	24%	17.51%
Client	High CPU load on client	4.78%	9.86%
	High I/O on client	33%	17.61%
	High memory load on client	61.52%	11.2%
Network	Bandwidth throttling	96.06%	20.06%
	Sporadic packet drops	0.21%	0.13%
	Packet reordering	3.71%	1.52%
	Latency	0	0

Table 2.5: The classification errors of NetPoirot in each general label broken down in terms of faults.

2.6.1 Overall Accuracy and Confusion Matrix

Duplex application. Fig. 2.3 shows the confusion matrix of NetPoirot when tested on the duplex application. We focus here on identifying the entity responsible for a failure. This could be the Client, Server, Network, or Normal (non-failure). Individual failure classification is deferred to Section 2.6.2. For better visualization, we show the confusion matrix as a bar graph in each class, where labels on the x-axis show the ground truth. The bars show what each failure was classified as by NetPoirot . The bar matching the ground truth label on the x-axis represents the value of recall. For example, Normal (green bar) has a high recall value close to 100% (height of the green bar). Precision values are reported on the x-axis next to the ground truth labels in parenthesis.

We make the following observations. First, network failures are the easiest to diagnose from TCP statistics and the most reliable among the three types of failure classes. NetPoirot has 99% precision for network failures, indicating that an output of "Network" from NetPoirot can be trusted. Server failures are the hardest to identify, given the lack of direct access to server side information. Recall was 82% for these failures, with the majority of errors going to Normal.

To understand the source of our errors, we looked into the specifics of the misclassified subclasses. Table 2.5 shows this information, limited to the misclassified data points from Fig. 2.3. $P(\text{fault} \mid \text{error} \cap \text{General label})$ describes the probability of a failure given that it was erroneously classified by NetPoirot and that it belonged to a particular class (Network/Server/Client). In other words, this column shows the breakdown of the error in each class to show how much was attributed to each particular failure. The second column, $P(\text{error} \mid \text{fault})$, shows the probability of classification error given a failure type. This shows the likelihood that a failure of a particular type will be misclassified by NetPoirot .

As can be seen, the error within each class is usually less than 20%. However, *High CPU load on server* seems to be the most problematic failure type with 24% misclassification as normal data. Also, note that even though Network failures have a high recall of 90%(as shown in Fig. 2.3), almost all the error in the remaining 10% can be attributed to bandwidth throttling. We found that within this failure, throttling at 50 and 1 Mbps caused the most problems as they did not significantly disrupt application performance.

The above shows the performance of NetPoirot when it relies *only* on TCP metrics from the client side. We can achieve near 100% precision/recall on Server and Client failures by augmenting network information with CPU/IO/Memory load from *only* the client machine (this is indicated by the high precision in the Network class). We can then locally check whether a client-side problem has occurred. If not, NetPoirot can check whether the failure is due to a Network problem. If both tests are negative, by elimination, the Server is the cause of failure. Note that simply relying on client information without TCP metrics (and NetPoirot) would not work – it would provide high precision/recall for client failures but fail (0% recall) for all other failures.

Finally, NetPoirot exhibits high precision in all classes of failures indicating that the entity output by NetPoirot can be trusted with high probability as the source of the failure.

Simplex application. The duplex application shows the worst case performance of NetPoirot on typical applications that have bidirectional communication. We also examine a more extreme situation where data is only transmitted from the client machine. Fig. 2.4 shows that while precision and recall remain high for network failures, it is more difficult to differentiate between Client/Normal and Server/Normal. We note, however, that the errors are not uniformly spread across all failures. In fact, NetPoirot achieved a recall of 99.9% in detecting high CPU load on the server. However, high I/O and high memory on the server side contribute to the majority of the misclassifications. This is because the simplex application lacks information that relates to data transmitted by the remote application, given that communication is only in one direction. To understand why this information is critical, we observe from Table 2.3 that in the case of the duplex application, the amount of data transmitted by the remote application (BReceived) plays a significant role in identifying the failure related to high memory on the server side.

We present the simplex application as an extreme scenario. Given that most applications have bidirectional communication similar to the duplex application, we focus on the duplex application for the rest of this section.

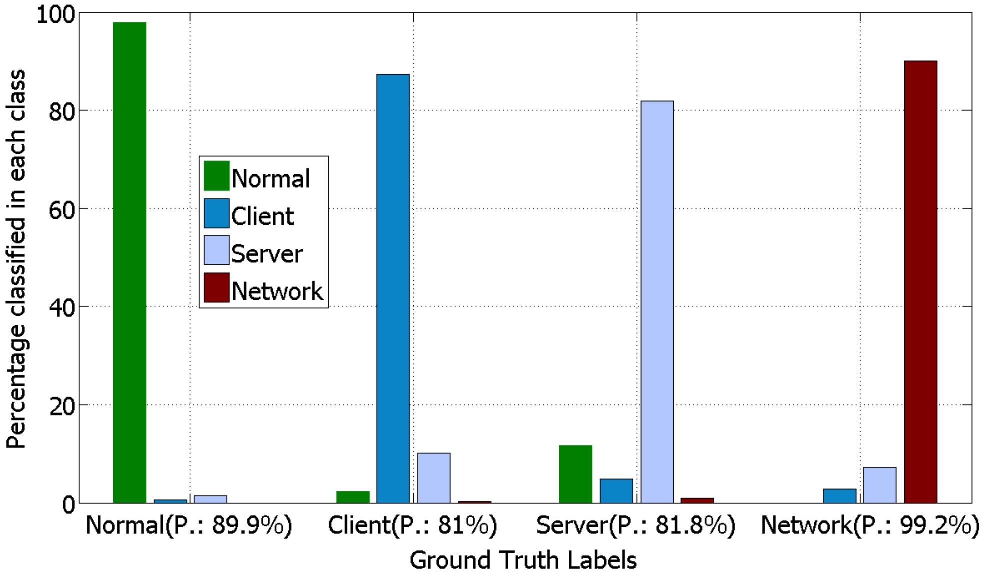


Figure 2.3: Confusion matrix on the **duplex** application's failure. Recall on each class is as follows: Normal: 97.94%,Client: 87.32%,Server: 81.89%, Network: 90%. Precision values are included in the x axis for each class.

2.6.2 Individual Failure Classification

NetPoirot is primarily designed to identify the entity responsible for a failure. As a more ambitious goal, it is highly desirable to also identify the actual failure type itself. To test the extent to which NetPoirot can accurately classify specific failures (as opposed to responsible entities), once the failure class is identified (Section 2.6.1), we use detailed failure labels to train an additional diagnosis function to identify the type of failure within the entity. The results are shown in Table 2.6. The third column only uses TCP metrics at the client, while the fourth column includes additional client-side information such as CPU, IO, and memory.

Table 2.6 shows that NetPoirot can classify the majority of failures, particularly on the network and server. In fact, NetPoirot is accurate enough in these classes

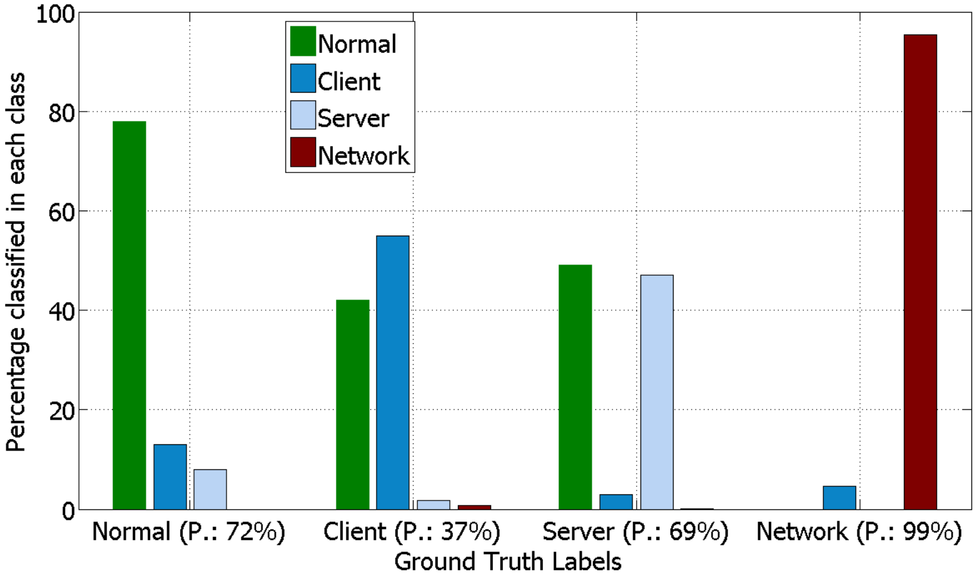


Figure 2.4: Confusion matrix on the **simplex** application's failure. Recall on each class is as follows: Normal: 78%,Client: 47%,Server: 55%, Network: 95.4%. Precision values are included in the x axis for each class.

so that we simply used a standard random forest trained for the failures without having to resort to running tournaments. For the client side failures, however, NetPoirot does not perform as well and we have to run tournaments. Distinguishing between high I/O and memory load on both the client and server proved extremely difficult without client-side information as TCP behavior remains largely the same in the presence of either failure. They are presented as a single class in Table 6. We note that, simply by augmenting the TCP statistics used by NetPoirot with client-side information, one can achieve high accuracy in identifying the individual failure type.

2.6.3 Untrained Failures

NetPoirot 's design is based on supervised learning and thus requires labeled data for training. This means that it should not be able to detect failures for which

General label	Fault	With only client network stats	With all client stats
Server side	High CPU load on server	Precision:99.69%, Recall:75.54%	Precision:99.78%, Recall: 76%
	Slow reading server	Precision:83.21%, Recall:95.66%	Precision:83.63%, Recall:96.44%
	High I/O or Memory on Server	Precision:65.78%, Recall:98.75%	Precision:76.97%, Recall:98.75%
Client side	High CPU load on client	Precision:75.64%, Recall:88.05%	Precision:100%, Recall:100%
	High I/O or Memory on Client	Precision:82.63%, Recall:98.48	Precision:100%, Recall:100%
Network	Bandwidth throttling	Precision:91.4%, Recall:79.94%	Precision:92.14%, Recall:85.53%
	Sporadic packet drops	Precision: 75.54%, Recall: 97.72%	Precision: 75.54%, Recall: 97.72%
	Packet reordering	Precision: 99.73%, Recall: 66.53%	Precision: 99.73%, Recall: 66.84%
	Latency	Precision:99.47%, Recall: 100%	Precision: 99.47%, Recall=100%

Table 2.6: Detailed fault classification with and without additional client side information.

it was not specifically trained. Thus, it is important to understand NetPoirot 's typical behavior in the presence of such failures. The situation can occur in one of the following two ways:

Dormant failures: A previously unknown type of failure is present during training and is labeled as Normal.

Unknown failures: A failure occurs for the first time during runtime.

While we cannot anticipate what the "unknown/dormant" failures would be, we attempt to illustrate this behavior by purposefully changing our original training data to reflect each of these behaviors. Ideally, we would like these failures to be either classified as Normal by NetPoirot , or as their ground truth class (actual entity).

To emulate dormant failures, we deliberately mislabeled each class in our training data as Normal before training NetPoirot . Similarly, to emulate unknown failures, we remove failed classes from the training data. We then investigate what entity will be output as being responsible for these unknown failures. Fig. 2.5

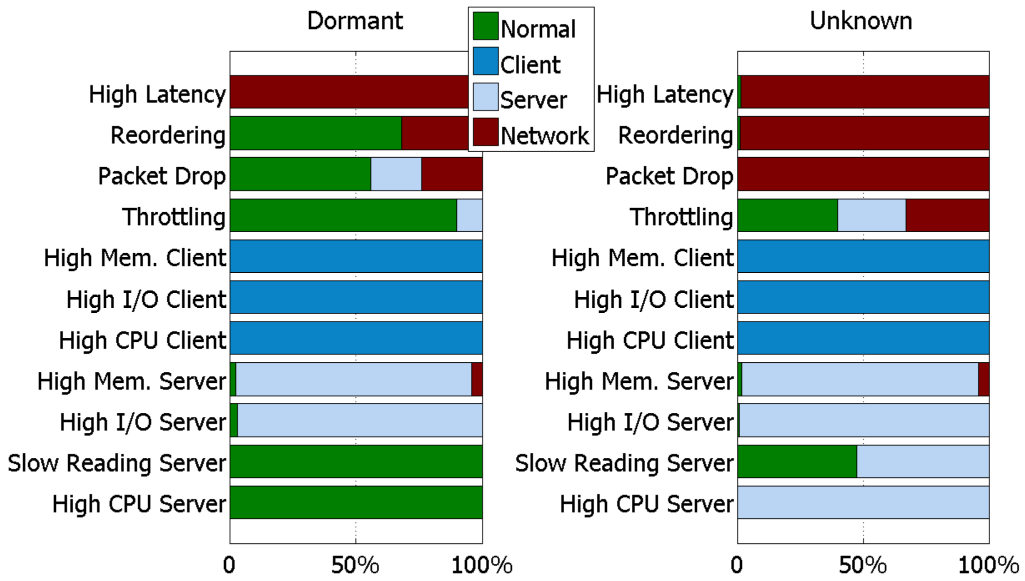


Figure 2.5: NetPoirot performance on dormant and unknown failures, when all client statistics are used.

shows that dormant failures result in most of the dormant failure being classified as Normal (what we are hoping for). This shows that in the presence of dormant failures, an output of Normal from NetPoirot may require further investigation to uncover the source of the problem. Fig. 2.5 also shows that NetPoirot is resilient to unknown failures, as failures from each entity have similar characteristics[7]. In the few failure types where significant misclassifications occur, they usually result in classification to Normal.

2.6.4 Sensitivity Analysis

We do a series of sensitive analysis to explore NetPoirot 's ability to detect failures across data centers.

So far we have shown the accuracy of NetPoirot when the data center locations

[7] Fig. 2.5 shows the results for when both client and network statistics are used. Our results showed that NetPoirot performed almost equally as well when using only network information.

of the machines used for testing only *partially* overlap with those used in training. Here we investigate whether the location of the machines used for training and testing influences the performance of NetPoirot .

Sensitivity to Cross Data-center Effects. In this experiment, we train NetPoirot for the duplex application using client machines in data centers on the west coast and one of the southern states. We then test the result on data from machines in the same southern state, and also on those in a midwest data center. Table 2.7 summarizes the precision and recall for test data collected for two machines in the southern data center, and two machines in the midwest. We observe that NetPoirot is mostly resilient to the location on which it has been trained. Interestingly, server-side failures are the most sensitive to location. We conjecture that this may be caused by cross-traffic effects as RTT variance increases. Further investigation is needed as part of our future work.

General label	**Southern State**	**Midwest**
Normal	Precision: 98.85% Recall: 98.2%	Precision: 83.34% Recall: 99.07%
Client	Precision: 100% Recall: 100%	Precision: 100% Recall: 100%
Server	Precision: 84.71% Recall: 94.67%	Precision: 97.75% Recall: 78.91%
Network	Precision: 98.85% Recall: 88.99%	Precision: 100% Recall: 98.82

Table 2.7: Performance breakdown by machine location. All client statistics are used for classification.

Sensitivity To Failure Duration. We next investigate NetPoirot ’s ability to identify short-lived failures. Table 2.8 summarizes our results for three failures (high IO at server, high IO at client, and high network latency). For training, we use failures that last for the application’s lifetime. For each test case, we inject failures with durations ranging from less than 30 seconds, to 5 to 6 minutes. We allow at least a 15 minute gap between each failure. Our results indicate that NetPoirot is not overly sensitive to the duration of the failure. Note the relatively higher accuracy in identifying high I/O on the server side as compared to Table 2.5. This is because the VMs we use for testing and training in this scenario are all from the same data centers.

Duration	Client (High IO)	Server (High IO)	Network (High Latency)
< 30s	100%	100%	100%
[30s,1min]	100%	99.99	100%
$[1, 3]$min	100%	96.96%	97%
$[3, 5]$ min	100%	99%	100%
$[5, 6]$ min	100%	96.01%	100%

Table 2.8: Sensitivity to failure duration. Recall numbers are shown here.

Sensitivity to Per-connection Training. NetPoirot is trained on a per-application basis, where the decision-tree based forests are built using TCP metrics aggregated across all connections of an application to the same service. This aggregation method has three advantages: 1) It allows us to take advantage of the differences across connections to better detect failures, 2) It allows us to capture an application's reaction to failure, and 3) It reduces the runtime/training overhead of NetPoirot .

General label	Normal	Client	Server	Network
Precision	74.2%	100%	48.16%	89.53%
Recall	85.77%	100%	41.46%	77.82%

Table 2.9: Performance of NetPoirot when used for per connection classification.

We explore another method of training, where we build the forest on a per-connection basis. Table 2.9 shows the precision/recall achieved by NetPoirot , on failure detection using the duplex application. Here, NetPoirot is trained using per-connection metrics. We observe that training on a per-connection basis requires $100X$ increased learning time. Hence, we are restricted to only 40% of the earlier training set. We observe that the recall and precision numbers are lower. For example, server-side failures are the hardest to classify, with a recall of 41.46%. Overall, per-application aggregation is a more accurate and efficient approach, and the alternative approach should only be considered if the operator requires per-connection diagnostics information.

2.6.5 Real Application Analysis

Our test applications present challenging scenarios that help us explore the limits of NetPoirot 's ability to detect failures. As noted in Section 2.5.1, these are extreme applications as they do not modify their communication patterns in response to failures. Here, we explore the performance of NetPoirot on two real-world applications, one based on video streaming, and another based on traces from our production data center containing EX failures.

YouTube video streaming. We tested NetPoirot on data collected while streaming YouTube videos in a browser. On 9 of our VM's located in three different data centers, we induced Client and Network failures[8] while streaming YouTube Videos. Table 2.10 shows the results for when only network statistics are used. Compared to our results in Section 2.6.1, we observe that NetPoirot does significantly better on the streaming application than on the duplex and simplex applications.

General label	**Normal**	**Client**	**Network**
Precision	97.78%	99.7%	100%
Recall	99.68%	98.25%	99.37%

Table 2.10: Performance of NetPoirot when used to identify failures when streaming YouTube videos.

Production applications experiencing EX. We run an experiment to validate NetPoirot 's effectiveness in identifying causes of EX failures based on real production traces. We use Syslog entries from our production machines to identify when an EX event caused a VM reboot. As training and test sets, we extract information captured by our monitoring agent,which is deployed on all compute nodes in our data centers, during the same time period . The monitoring agent used to capture this data is an older version of NetPoirot and does not report metrics such as time spent in zero window probing which we added in subsequent releases. We additionally use resolved tickets to identify the cause of failures.

[8] Server failures are excluded as we do not control the servers.

We use a large subset of machines (162/175) for training as EX occurrences on any one machine tend to be low (typically at most 1), though in aggregate, they occur frequently. The remaining machines are used for testing. We lack direct access to the ground truth labels and have to use a combination of log analysis and resolved tickets to identify the start-time/type of the failure that lead to EX. Note that in a real-world deployment, we would have injected faults as described in Section 2.3 in the training phase to induce and learn about all types of EX failures, leading to a more accurate failure labeling process.

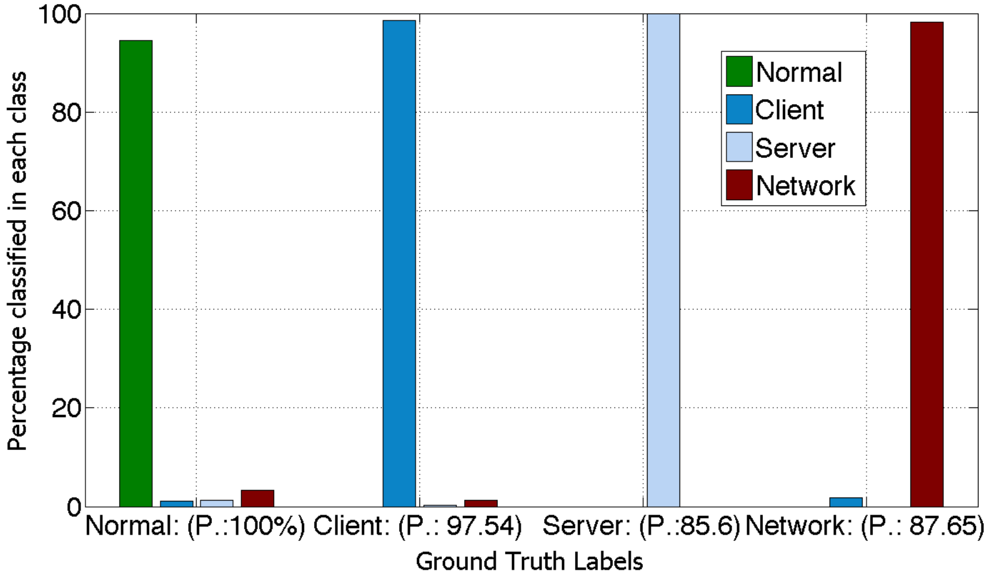


Figure 2.6: Confusion matrix on EX. Recall on each class is as follows: Normal: 94.53%,Client: 98.54%, Server: 100%,Network: 98.2%.

In the training set, we mark the duration of each failure based on a simple heuristic. We know when the failure ended as the VM experiencing EX reboots (detectable from Syslog), we set the failure start time to 3 minutes prior to this reboot. We do not use data after the reboot. 3 minutes is a conservative estimate that we apply. However, this is at best an estimate. After talking to operators of a large scale datacenter, they confirmed that the duration of failures that cause EX are highly variable and can last from a couple of seconds to minutes.

To improve label accuracy, we first pass each class of failure in the test set through a classifier that identifies *that* failure (e.g. Client) from Normal data. We then label the data based on the output as failure/Normal making sure that each class occurs with contiguous time stamps. Similarly for the data that we label as normal, based on the 3 minutes time-frame, we first pass it through each of these classifiers and label it as failure if its timestamp is a continuous extension of the failure on the same machine. This allows us to extend the failure labeling past the 3 minute conservative estimate.

Fig. 2.6 shows the confusion matrix similar to the format described in Section 2.6.1. Since client information was not collected, the results are based on NetPoirot when using network statistics alone. We make the following observations. First, despite lacking the actual ground truth on failures and the time interval in which they occur, NetPoirot never misclassifies failures as other types of failures. Second, given course granularity labels of failures vs normal, NetPoirot is able to achieve very high recall values, consistently above 94%, and in fact, achieves 100% recall for server failures. This suggests that NetPoirot is a promising approach at pinpointing the entity causing EX.

2.7 Discussion

Our results in Section 5.4 demonstrates the effectiveness of NetPoirot in pinpointing the source of failures in a data center. We perform a range of sensitivity analysis, on different applications to fully explore the efficacy of NetPoirot . In doing so, we have identified possible extensions of NetPoirot , as well as gained a better understanding of the scenarios in which it is most useful. We briefly discuss these possibilities here, as well as point out directions for future extensions.

NetPoirot uses lightweight endpoint monitoring, is non-intrusive, and incurs minimal overhead to machines in a data center. Despite being lightweight, NetPoirot is able to perform accurate failure detection of the entity responsible for the failure, and even the type of failure itself. This precludes identification of the actual device, e.g. the physical router that causes a network problem. Another limitation of NetPoirot is its reliance on TCP metrics, which precludes failures observed by UDP. Given that most traffic within data center uses TCP, a wide range of applications (e.g. web apps, database applications) can still benefit from NetPoirot .

NetPoirot is a lightweight failure identification tool for data center applications. We have identified several avenues to further improve our results via more sophisticated learning techniques. These include:

Cross-application learning. NetPoirot requires per-application training, which we argue is feasible, given that learning can be done offline for each new application being deployed in a data center. One interesting idea we are exploring is the use of a concept in machine learning, known as *transfer learning* [109], in which the feature space of one application can be modified so that it can be used in NetPoirot for identifying the cause of failures in another application.

Non-production training. NetPoirot exploits the fact that data centers tend to have homogeneous setups, where failures can be induced on a subset for machines for supervised learning. This has been explored by others [129]. As an enhancement, we plan to explore training in a staging environment. Our sensitivity analysis in Section 2.6.4 suggests that NetPoirot is resilient to learning across data centers, suggesting it is possible to limit the training to a single cluster within a data center, and still apply the results to another cluster with similar configuration.

Improving accuracy. In Section 2.4.2 we reported that traditional CV on our data set yields an error of 1.5%. Cross-validation error where each fold comprises of data from only a single machine is much higher (11%). This suggests that *if* samples of faulty/normal data from a machine exist in the training data, the classification error on that machine dramatically decreases. Thus, continuously changing the machines used in training should improve the accuracy of NetPoirot even further over time.

Chapter 3

Finding The Device Responsible For The Failure

In Chapter 2 we introduced NetPoirot, a system which allows for the identification of the entity responsible for a failure that is completely contained to the end host. NetPoirot, however, is only the first step in the diagnosis process. Once a network failure is detected it still remains to detect the link/switch that caused the failure. For example, during our deployment of NetPoirot in a production datacenter, we found that on a typical day 17% of VM reboots were due to network related failures. Such networks are comprised of thousands of links and switches. Thus finding the cause of failures is often equivalent to finding a needle in the proverbial haystack.

In this chapter we will outline why state of the art diagnosis algorithms fall short in identifying the cause of many of such failures and present a new system Vigil which not only allows for identifying the cause of failures but also allows us to go one step further and to identify the link most likely to have dropped packets on each individual TCP connection.

3.1 Introduction

Vigil started with an ambitious goal: For every TCP retransmission in our data centers, we wanted to pinpoint the network link that caused the packet drop that triggered the retransmission with negligible diagnostic overhead or changes to the networking infrastructure.

This goal may sound like an overkill—after all, TCP is supposed to be able to deal with a few packet losses. Packet losses might occur due to simple congestion instead of network equipment failures. Even network failures might be transient. Above all, there is a danger of drowning in a sea of data without generating any actionable intelligence.

These objections are valid, but so is the need to diagnose TCP "failures" which can result in severe problems for applications. For example, in most data centers, VM images are stored in a storage service. When a customer boots a VM, the image is mounted over the network. Thus, even a small network outage can cause the host kernel to "panic" and reboot the guest VM. In fact, our observations show that 17% of VM reboots in a production data center were caused by network issues and in over 70% of these, none of the available monitoring systems were able to pinpoint the link(s) that caused the problem. To illustrate this problem, in a one-day snapshot, there were on average 10 VM reboots per hour across these data centers caused by network problems that could not be attributed to a specific network link or device.

Since VM reboots directly affect the end customer, we place very high value on understanding their root causes. Any persistent pattern in such transient failures is a cause for concern and is potentially actionable. An example of such failure is silent packet drops [65]. Such problems are nearly impossible to detect with traditional monitoring systems (e.g., SNMP counters). If a switch is experiencing such problems, we may want to reboot or replace it. Such interventions are "costly" in that they affect a large number of flows/VMs. Therefore, we need a system to correctly assign the blame in face of such transient failures.

There is a lot of prior work in the area of network failure diagnosis and blame assignment. However, none of the existing systems meet the ambitious goal we have set for ourselves. Pingmesh [65] sends periodic probes to detect link failures and can therefore leave “gaps” in coverage, as it must manage the overhead of probing. Also, since it uses out-of-band probes, it cannot detect failures that affect only in-band data. NetPoirot, identifies the network as a likely cause of performance issues, but cannot find the specific device that causes the problem. Roy et. al. [119] report a system that monitors all paths in the network for possible link failures. Their system requires modifications to routers and assumes a specific topology. Everflow [145] cannot be directly used to pinpoint the location of packet drop, since it would require capturing all traffic, which is not scalable.

To address these limitations, we propose Vigil, a simple, lightweight, always-on monitoring tool. Vigil records the path of TCP connections that suffer from one or more retransmission and assigns a proportional “blame” to each link on the path. Vigil then provides a *ranking* of the links that represents their relative packet drop rates. Using this ranking, it can find the most likely cause of packet drops on each TCP connection.

Vigil has several noteworthy properties. First, it does not require any changes to the existing networking infrastructure. Second, it does not require changes to the client software—the monitoring agent is an independent entity that sits on the side. Third, Vigil detects in-band failures and is hence more useful than tools such as Pingmesh [65]. Fourth, Vigil continues to perform well in the presence of noise, as opposed tools such as to NetDiagnoser [48]. Finally, Vigil's overhead is negligible.

While the high-level design of Vigil is deceptively simple, the practical challenges of making Vigil work and the theoretical challenge of *proving* that Vigil works are non-trivial. For example, its path discovery is based on a traceroute-like approach. Due to the use of ECMP, traceroute packets have to be carefully crafted to ensure that they follow the same path as the TCP connection. Also, we needed to ensure that we do not overwhelm the routers along the path by sending too many traceroutes (traceroute responses are handled by control-plane CPUs of the routers, which are quite puny). To this end, we had to do careful calculations to ensure that our sampling strikes the right balance between the need for accuracy and the overhead on the switches. On the theoretical side, we are able to show that Vigil's simple blame assignment scheme is highly accurate even in the presence of noise.

In summary, this paper makes the following contributions: (i) we design Vigil, a simple, lightweight, and yet accurate fault localization system for data center networks; (ii) we prove that Vigil is accurate without imposing excessive burden on the switches; (iii) we prove that the simple blame assignment scheme used by Vigil correctly finds the failed links with high probability; and (iv) we show how to tackle numerous practical challenges involved in deploying Vigil in a real data center.

Our results from deploying Vigil in a tier-1 data center show that it detects every problem identified by other previously deployed monitoring services while also pinpointing the sources of problems for which information is not provided by these monitoring services.

3.2 Problem and challenges

The goal of Vigil is to identify the cause of TCP retransmissions with high probability. Our current focus is mainly on infrastructure traffic, especially connections to services such as storage as these can have the most severe consequence (see §3.1). Note that we are deliberately not excluding congestion-induced retransmissions.

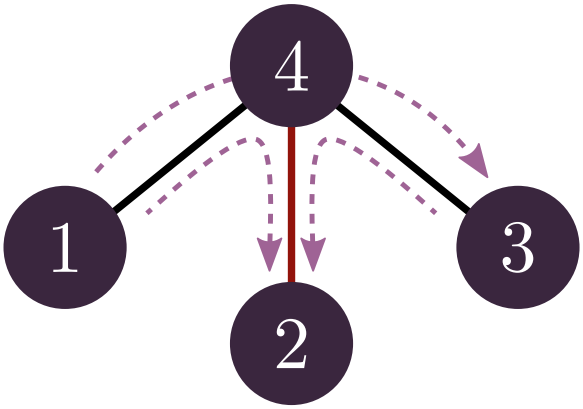


Figure 3.1: Simple tomography example, where a simple optimization problem can find the problematic link.

If episodes of congestion (however short-lived) are common on a link, we want to be able to flag them. Of course, in practice, any such system needs to deal with a certain amount of "noise"—which we will formalize later.

The design of Vigil is driven by two practical requirements: (i) it should scale to data center size networks and (ii) it should be deployable in a running data center with as little change to the infrastructure as possible.

Generally, there are three ways to monitor link failures to determine the cause of packet drops. First, one can continuously monitor switch counters. These are, however, inherently unreliable [136] and monitoring thousands of switches in a data center at a fine time granularity is not scalable. Having to correlate this data with each TCP retransmission *significantly* exacerbates this problem. Second, one can use a system like PingMesh [65] that sends probe packets to monitor link status. Such systems suffer from a rate of probing trade-off: sending too many probes creates unacceptable overhead whereas reducing the probing rate leaves temporal and spatial gaps in coverage. More importantly, the probe traffic does not capture what the end user and the TCP connections see. Thus, we choose the third alternative, which is to use data traffic itself as probe traffic [119]. Using data traffic has the advantage that the system introduces little or no monitoring overhead.

As one might expect, almost all traffic in our data centers is TCP traffic. One way to monitor this type of traffic is to use a system like Everflow. Everflow inserts a special tag in every packet and has the switches mirror tagged packets to special collection servers. Thus, if a tagged packet is dropped, we can easily determine the link on which it happened. Unfortunately, there is no way to know in advance which packet is going to be dropped, so we would have to tag and mirror every single TCP packet. This is clearly infeasible. We could tag only a fraction of packets, but doing so would result in another sampling rate trade-off.

Hence, it follows that we must rely on some form of network tomography [143, 71, 85]. We can take advantage of the fact that TCP is a connection-oriented, reliable delivery protocol, so that any packet loss results in retransmission[1],which can be easily detected. If we knew the path of every TCP connection, we can set up a standard optimization problem to determine, with high confidence, which link may have dropped the packet. For example, in Figure 3.1, the link between nodes 2 and 4 is lossy and drops packets. As a result, connections 1-2, and 3-2 suffer from retransmissions, but connection 1-3 does not. A straightforward set cover optimization formulation that attempts to minimize the number of "blamed" links will correctly identify the cause of drops on each TCP connection.

Still, there are two issues with this approach: (i) the optimization problem is known to be NP-hard [22] and solving it on the data-center scale is not feasible; (ii) tracking the path of every single TCP connection in the data center is not scalable in our setting.

One can use alternative solutions such as using Everflow to track the path of SYN packets or use a system like the one described in [119]. However, both these schemes rely on making changes to the switches. The only way to capture the path of a TCP connection without making any special infrastructure support is to run something like a *traceroute*. However, traceroute relies on getting ICMP TTL

[1] False retransmissions are rare and we handle them (§3.4).

exceeded messages back from the switches. These messages are generated by the control plane, i.e., the switch CPU, not the ASIC that drives the dataplane. To avoid overloading the CPU, our datacenter administrators have capped the rate of ICMP responses to 100 per second. This severely limits the number of TCP connections we can track.

Given these limitations, what can we do? In this paper, we show that the answer, for data center networks, is deceptively simple. We show that if (a) we track the path only of those TCP connections that have suffered retransmissions, (b) assign each link on the path of such a connection a vote of $1/h$, where h is the path length, and (c) sum up the votes during a given period, then the top-voted links are almost always the ones that are dropping packets (see §3.5)! Unlike the solution of the optimization problem, our scheme is able to provide a *ranking* of the links in term of their drop rates, i.e. if link A has a higher total vote than B, it is also dropping more packets (with high probability). This allows Vigil to find the link most likely responsible for each connection's packet drops.

In the next three sections, we describe Vigil in more detail, beginning with an overview of its architecture.

3.3 Design Overview

Figure 3.2 shows the overall architecture of Vigil. Within our data centers, Vigil is deployed side-by-side with other applications on each end-host as a user-level process running in the host OS. Vigil (shown in light purple/gray) consists of three agents for TCP monitoring, path discovery, and analysis.

At a high level, the *TCP monitoring agent* detects potential retransmissions at each end-host. The TCP monitoring agent is deployed on all end hosts in the data center. The Event Tracing For Windows (ETW) [91] framework[2] notifies the agent

[2] Similar functionality exists in Linux.

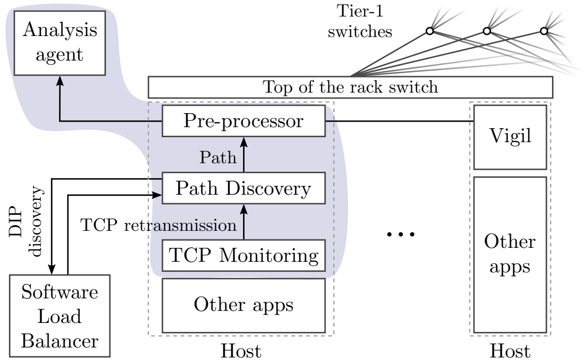


Figure 3.2: Overview of Vigil architecture

as soon as an active TCP connection established by the host suffers a retransmission (Vigil currently does not monitor TCP connections in the VMs).

Upon a retransmission, the TCP monitoring agent then triggers the *path discovery agent* (§3.4) to identify the set of links along this connection. The path discovery agent uses a modified version of traceroute to discover this path to the destination IP (DIP).

At each end-host, at regular intervals (30 seconds in our implementation), the voting scheme described in §3.5 is carried out based on the reported paths of connections that suffered retransmission within the epoch. The results of the vote is sent to a centralized analysis agent to identify the top-voted links across the entire data center.

Overall, the Vigil implementation consists of 6000 lines of C code at the end-host. The analysis engine contains an additional several hundred lines of code. Its memory usage never goes beyond 6 KB on any of our production hosts and its CPU utilization is minimal ($+1\%$ overhead for each core). The bandwidth utilization of Vigil due to traceroute is minimal (maximum of 200 KBps per host). As our results will show, Vigil can determine the most likely culprit for each connection, even with loss rates as low as 0.05%.

We now describe the path discovery agent and the analysis agent in more detail.

3.4 The path discovery agent

The path discovery agent uses traceroute functionality to discover the path of TCP connections that suffer retransmissions. We first ensure that the number of traceroutes sent by the agent does not overload our switches (§3.4.1). We then briefly describe the key engineering issues and how we solve them (§3.4.2). Our data centers use IP routing and thus the path discovery agent captures the L3 path. The architecture of the path discovery agent is shown in Figure 3.2.

3.4.1 ICMP Rate Limiting

Generating ICMP packets in response to traceroute consumes switch CPU, which is a valuable resource in a data center. In our network, there is a hard cap of $T_{\text{max}} = 100$ on the number of ICMP messages a switch will send per second. We want to limit the traceroute load on the switches such that the number of ICMP messages is below T_{max}. To do so, we first checked the hop count of all host's TCP connections during a one hour period in one of our production data centers to see how many connections might go through a T_3 switch. We observed that 97.9% of these connections had a hop count less than or equal to 5 and *did not* go through a T_3 switch. Given that our network is a Clos topology and assuming that hosts under a top of the rack switch (ToR) communicate with hosts under a different ToR uniformly at random, we show:

Theorem 1. *The rate of ICMP packets generated by any switch due to a traceroute is* *below T_{max} if the rate C_t at which hosts trigger traceroutes is upper bounded as*

$$C_t \leq \frac{n_1 n_2 T_{max}}{H \max\left[n_2, \frac{n_0^2(n_{\text{pod}}-1)}{n_0 n_{\text{pod}}-1}\right]}, \tag{3.1}$$

where n_0, n_1, and n_2 are the numbers of ToR, T_1, and T_2 switches respectively and H is the number of hosts under each ToR.

Proof. See appendix.

□

For example, the upper bound of C_t in one of our data centers is 10. This means that as long as hosts do not open more than 10 connections per second, we can *guarantee* that the number of traceroutes generated by Vigil will not go above T_{max}. We use the hard threshold C_t obtained in the above calculation to limit the number of traceroutes allowed on each host to prevent overwhelming the switches. The agent triggers path discovery no more than once every epoch *for a given connection* to further limit the number of traceroutes. We use epochs of 30 seconds. We will show in §3.5 that this number is sufficient to ensure high accuracies in detecting links dropping packets in the network.

3.4.2 Engineering Challenges

Using the correct five-tuple: Like most modern data centers, our network also makes extensive use of ECMP. All packets of a given flow, defined by the so-called five-tuple, follow the same path [68]. Thus, the traceroute packets must have the same five-tuple as the TCP connection we are attempting to trace. To ensure this, we must account for the use of load balancers.

TCP connections are initiated in our data center in a manner similar to that described in [110]. The connection is first established to a virtual IP (VIP) and the SYN packet (containing the VIP as destination) goes to a software load balancer

(SLB) which assigns that connection to a physical destination IP (DIP) and a service port associated with that VIP. The SLB then sends a configuration message to the virtual switch (vSwitch) in the hypervisor of the source machine that registers that DIP with that vSwitch. The destination of all subsequent packets in that connection have the DIP as their destination and do not go through the SLB. In order for the path of the traceroute packets to match that of the data packets, its header should contain the DIP and not the VIP. Thus, before tracing the path of a connection, the path discovery module first queries the SLB for the VIP-to-DIP mapping for that connection and uses the DIP as the destination address for the traceroute packets. Note that there are cases where the TCP connection establishment itself may fail due to packet loss. Path discovery is not triggered for such connections. It is also not triggered when the query to SLB fails to avoid tracerouting the internet.

Re-routing and packet drops: We must consider the possibility that the traceroute itself may fail. For example, this may happen if the link drop rate is very high. This actually helps us, since it directly pinpoints the faulty link. Our analysis engine, described in the next section, is able to use such partial traceroutes.

A more insidious possibility is that the routing may change by the time traceroute starts. We use BGP in our data center network. A lossy link may cause one or more BGP sessions to fail, triggering rerouting. Then, the traceroute packets may take a different path than the original connection.

The design of VigilVigil helps avoids this problem. The RTTs in a data center are typically less than 1 or 2 milliseconds, so the TCP sender retransmits a dropped packet quickly. The ETW framework notifies the TCP monitoring agent immediately, which invokes the path monitoring agent. The only additional delay is the time required to query the SLB to obtain the VIP-to-DIP mapping, which is typically less than a millisecond. Thus, as long as paths are stable for a few milliseconds after a packet drop, the traceroute packets will follow the same path as the TCP connection.

Our network also makes extensive use of link aggregation (LAG) [75]. However, unless all the links in the aggregation group fail, the L3 path is not affected.

Router aliasing We note that the problem of router aliasing [64] is easily solved in a data center, as we know the topology, names, and IP addresses of all routers and interfaces. We simply map the IP's from traceroutes to the switch names (which we have from the topology).

To summarize, Vigil's implementation is as follows: Once the TCP monitoring agent notifies the path discovery agent that a connection has suffered a retransmission, the path discovery agent checks its cache of discovered path for that epoch and if need be, queries the SLB for the DIP. It then sends 15 appropriately crafted TCP packets with TTL values ranging from $0 - 15$. In order to disambiguate the responses, the TTL value is also encoded in the IP ID field [3]. This allows for concurrent traceroutes to multiple destinations. The TCP packets deliberately carry a bad checksum to ensure that they do not interfere with the ongoing connection.

3.5 The Analysis Agent

Next we describe the analysis agent, focusing first on its voting-based scheme, before presenting an analytical optimal solution that is NP-hard for comparison.

3.5.1 Voting Based Scheme

Vigil's analysis agent operates based on a simple voting scheme. If a connection sees a retransmission, Vigil votes the links of that connection to be *bad*. Each vote carries a value that is tallied at the end of each epoch, giving a natural ranking of each link in the network. We set the value of good votes to 0 (if a connection has no retransmission, no traceroute needs to be issued). Bad votes are assigned a value of $\frac{1}{h}$, where h is the number of hops on the path since every link on the path is equally likely to be responsible for the drop.

The ranking obtained by Vigil after tallying the votes allows us to identify the most likely cause of packet drops on each connection by comparing their ranking (which are higher for links with higher drop rates). To further guard against high levels of noise, we can use our knowledge of the topology. Namely, when finding the rankings, the votes of the links lower in the list can be adjusted (assuming ECMP distributes connections uniformly at random) using an approximation of the number of votes due to the links higher up in the list. We saw in our evaluations that this can result in a 5% reduction in the number of false positives.

Note, that Vigil can also be used to find the most problematic links in the network at each point in time. This can be done using Algorithm 1. The algorithm first sorts the links based on their votes. It then checks whether the top voted link's votes are higher than 1% of the total votes cast. If not, it stops. If yes, it marks it as one of the most problematic links, removes it from the set of links, and adjusts the votes of the remaining links as described earlier. It continues to iterate in this manner until the stop criterion is met.

3.5.2 Voting Scheme Analysis

Can Vigil deliver on its promise of finding the most probable cause of packet drops on each TCP connection? In Vigil's voting scheme failed connections contribute to increases in the tally of both good and bad links. Moreover, in a large data center such as ours, occasional, lone, and sporadic packet drops can and *will* happen due to good links (links that have very low drop rates). These failures are akin to "noise" and can cause severe inaccuracies in any detection system [99], Vigil included. Thus, the answer to this question is not obvious. We show that the

Algorithm 1 Finding the most problematic links in the network.

```
$\mathcal{L} \leftarrow$ Set of all links
$\mathcal{P} \leftarrow$ Set of all *possible* path
$v(l_i) \leftarrow$ Number of votes for $l_i \in \mathcal{L}$
$\mathcal{B} \leftarrow$ Set of most problematic links
$l_{max} \leftarrow$ Link with the maximum votes out of $\forall l_i \in \mathcal{L} \cap \mathcal{B}^c$
while $v(l_{max}) \geq 0.01(\sum_{l_i \in \mathcal{L}} v(l_i))$ do
    $l_{max} \leftarrow \max_{l_i \in \mathcal{L} \cap \mathcal{B}^c} v(l_i)$
    $\mathcal{B} \leftarrow \mathcal{B} \cup \{l_{max}\}$
    for $l_i \in \mathcal{L} \cap \mathcal{B}^c$ do
        if $\exists \quad p_i \in \mathcal{P}$ s.t. $l_i \in p_i$ & $l_{max} \in p_i$ then
            Adjust the score of $l_i$
        end if
    end for
end while
return $\mathcal{B}$
```

likelihood of Vigil making these errors is small. Taking advantage of the fact that we know our topology (Clos network):

Theorem 2. *For $n_{\text{pod}} \geq \frac{n_0}{n_1} + 1$, Vigil will rank with probability $1 - 2e^{-\mathcal{O}(N)}$ the $k < \frac{n_2(n_0 n_{\text{pod}} - 1)}{n_0(n_{\text{pod}} - 1)}$ bad links that drop packets with probability p_b higher than all good links that drop packets with probability p_g if*

$$p_g \leq \frac{1 - (1 - p_b)^{c_l}}{\alpha c_u}$$

where N is the total number of connections between hosts, c_l and c_u are lower and upper bounds, respectively, on the number of packets per connection, and

$$\alpha = \frac{n_0(4n_0 - k)(n_{\text{pod}} - 1)}{n_2(n_0 n_{\text{pod}} - 1) - n_0(n_{\text{pod}} - 1)k}. \tag{3.2}$$

Proof. See appendix.

□

A single connection is unlikely to go through more than one "failed" link in a network with thousands of links. Theorem 7 ensures that the tally of votes on links with higher drop rates is larger than that of good links. Combining these two observations allows Vigil to find the most likely cause of packet drops on each connection.

Note that a straightforward corollary of Theorem 7 is that in the absence of noise ($p_g = 0$), Vigil can pinpoint all bad links with high probability for large enough N. In the presence of noise, Vigil can still identify the bad links, as long as the probability of dropping packets on non-failed links is low enough (i.e., the signal-to-noise ratio is large enough). This number, however, is compatible with typical values found in practice. Indeed, take as an example c_l and c_u to be the 10^{th} and 90^{th} percentiles, respectively, of the number of packets sent by TCP connections across all our hosts in a 3 hour period. If $p_b \geq 0.5\%$, the packet drop rate on good links can be up to 1.8×10^{-6}. For $p_b \geq 0.05\%$, then Theorem 7 only requires $p_g \leq 7.7 \times 10^{-7}$. Note that packet drop rates in a production data center are typically below 10^{-8} [146].

Another important consequence of Theorem 7 is that it establishes that the probability of errors in Vigil's results diminishes exponentially with N, so that even with the limits imposed by Theorem 3.1 we do not need to use epochs longer than $30s$. Moreover, it is worth noting that the conditions in Theorem 7 are sufficient but not necessary. In fact, §3.6 illustrates how well Vigil performs even when the conditions in Theorem 7 do not hold (e.g., for a larger number of failed links).

3.5.3 Optimal Solution

While Vigil's voting scheme works with high probability in a set of situations (those considered in Section 3.5.2), it is not guaranteed to work in all situations. In this section, we illustrate an alternative solution in the form of an optimization problem. Given enough resources and time, this method can find the set of links dropping packets on any given connection with high fidelity. The issue is that finding a solution to this optimization problem is NP-hard in general [22]. We compare Vigil to this benchmark in our evaluations.

Our goal is to find the most likely cause of failures. This results in the following optimization problem:

$$\begin{aligned} \text{minimize} \quad & \|\boldsymbol{p}\|_0 \\ \text{subject to} \quad & \boldsymbol{A}\boldsymbol{p} \geq \boldsymbol{s} \\ & \|\boldsymbol{p}\|_1 = \|\boldsymbol{s}\|_1 \\ & p_i \in \mathbb{N} \cup \{0\} \end{aligned} \tag{3.3}$$

where $\mathbb{N}$ is the set of natural numbers, $\boldsymbol{A}$ is a $C \times L$ routing matrix, $\boldsymbol{s}$ is a $C \times 1$ vector that collects the number of retransmissions suffered by each connection during an epoch, L is the number of links in the network, C is the number of connections in an epoch, and $\|\boldsymbol{y}\|_0$ denotes the cardinality of the support of $\boldsymbol{y}$. Note that the i-th element of $\boldsymbol{p}$ estimates how many retransmissions were caused by link i, so that it can be used to rank the network links.

The formulation assumes that the minimum subset of links that explain all packet drops is close to representing the set of links dropping packets in reality (this can be shown to be equivalent to a maximum likelihood formulation). Formulating the problem as 3.3 allows us to relax the stringent $\boldsymbol{A}\boldsymbol{p} = \boldsymbol{s}$ constraint employed in tomography, which allows for finding the most probable cause of failure. A different binary optimization formulation which only marks the links as good/bad is also possible. In fact, it is the optimization problem that [48, 80] approximate. Namely,

$$\begin{aligned} \text{minimize} \quad & \|\boldsymbol{p}\|_0 \\ \text{subject to} \quad & \boldsymbol{A}\boldsymbol{p} \geq \boldsymbol{s} \\ & p_i \in \{0, 1\} \end{aligned} \tag{3.4}$$

Similarly to (3.3), this minimum set cover formulation is not tractable (it is NP-complete [22]). Moreover, it does not provide a ranking of the links and does not perform well in the presence of noise. Our evaluations showed that Vigil (Algorithm 1) significantly outperformed this binary optimization (by more than 50% in the presence of noise). We illustrate this point in Figures 3.4 and 3.9, but otherwise omit results for this optimization in §3.6 for clarity.

In the next three sections, we present our evaluation of Vigil in simulations §3.6, in a test cluster §3.7, and in one of our production data centers §3.8.

3.6 Evaluations: Simulations

We start by evaluating in simulations where we know the ground truth. In these simulations, Vigil finds connections whose drops were due to noise and marks them as "noise drops". It then finds the link most likely responsible for packet drops on the remaining set of connections ("failure drops"). A noisy drop is defined as one where the link responsible for the drop only dropped a single packet. We found that Vigil never marked a connection into the noisy category incorrectly. We therefore focus on the accuracy for connections that Vigil puts into the failure drop class.

Performance metrics. Our primary measure for performance for Vigil is *detection accuracy*, which is defined as the likelihood of being able to correctly identify the cause of a failed connection (e.g., pinpoint the correct link that failed). For simplicity, we refer to this only as *accuracy* in this section. For evaluating Algorithm 1, we use the alternative performance metrics of *recall* and *precision*. Recall is a measure of reliability and shows how many of the failures Vigil can accurately detect (false negatives). For example, if there are 100 failed links and Vigil detects 90 of them, its recall is 90%. Precision is a measure of accuracy and shows to what extent Vigil's results can be trusted (false positives). For example, if Vigil flags 100 links as bad, but only 90 of those links actually failed, its precision is 90%.

Simulation setup. We use a flow level simulator implemented in MATLAB. Our simulated topology consists of 4160 links, 2 pods, 20 ToRs per pod. Each host in the simulator establishes 2 connections per second to a random ToR in the network outside of its rack. The simulator has two types of links. For *good links* that are operating without failure, packets are dropped at a very low rate chosen uniformly from $(0, 10^{-6})$ to simulate noise. On the other hand, *failed links* have a much higher drop rate to simulate failures. By default, drop rates on failed links are set to vary uniformly from 0.01% to 0.1%, though to study the impact of drop rates, we do allow this rate to vary as an input parameter. The number of good and failed links is also a tunable simulation parameter. Every 30 seconds of simulation time, we send up to 100 packets per connection, and drop packets based on the rates above as they traverse links along the connection. The simulator records all connections with at least one packet drop, for each such connection, the link with the most drops.

As a basis for comparison, we compare Vigil against the optimal solution described in in §3.5.3. We refer to the optimal solution as "Integer Optimization" in the figures, and "optimization" in the text when comparing with Vigil.

3.6.1 In the optimal case

In our first experiment, we use simulation settings where the bounds in Theorem 7 do hold. Recall that these bounds are sufficient conditions for Vigil's accuracy. The first experiment aims to validate that Vigil can trivially achieve high levels of accuracy as expected. Given the simulation setup above, we set the drop rates on the failed links to be between $(0.05\%, 1\%)$. At drop rates below 0.05%, the theorem bounds will no longer hold.

Accuracy. Figure 3.3 shows that Vigil has an average accuracy that is higher than 96% in almost all cases. Furthermore, due to its robustness to noise, it also outperforms the optimization algorithm (§ 3.5.3) in most cases.

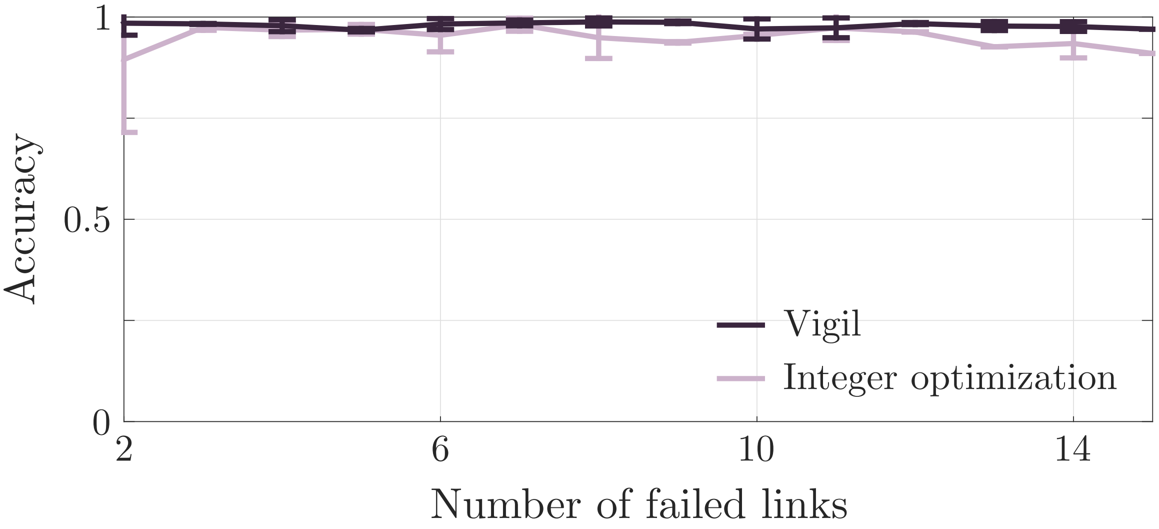


Figure 3.3: When Theorem 2 holds.

Recall and precision. Figure 3.4 shows that even with low packet drop rates on the failed links, Vigil detects the failed links with high recall and precision. This confirms that when the conditions in Theorem 7 hold, Vigil can be trusted to find the right set of links.

In the rest of this section, we proceed to evaluate Vigil's performance when the conditions in Theorem 7 *do not* hold (i.e. packet drop rates below 0.05%). This illustrates that while sufficient, these conditions are not necessary for the good performance of Vigil.

3.6.2 Varying Drop Rates

Our next experiment aims to push the boundaries of Theorem 7 by varying the "failed" links drop rates well below the conservative bounds of Theorem 7.

Single Failure. Figure 3.5a shows simulation results for different drop rates on a single failed link in the network. Our results show that for high drop rates, Vigil can find the cause of packet drops on each connection with high accuracy. Even as the drop rate decreases below the 0.05% bound obtained from Theorem 7, we

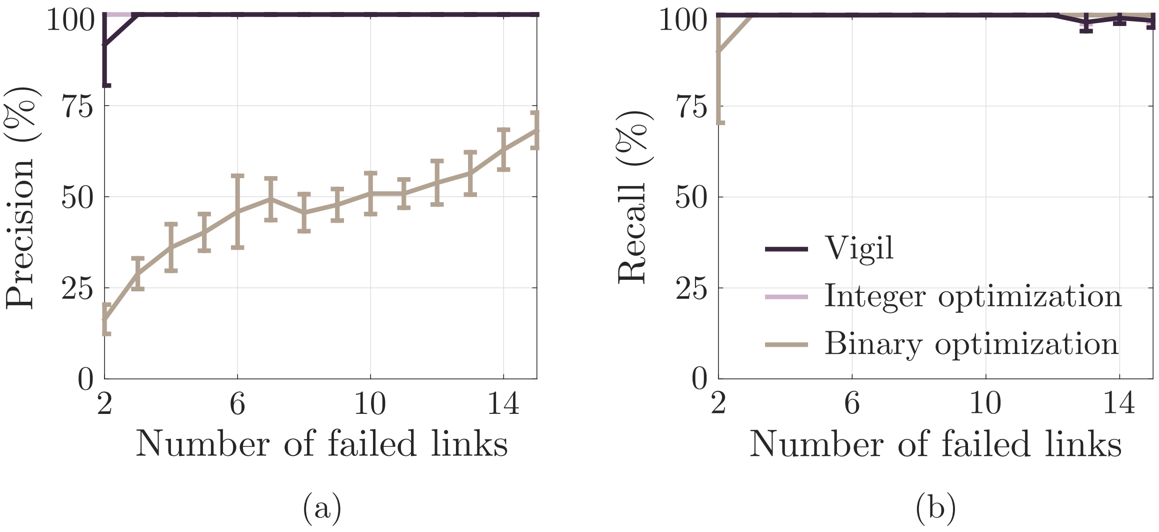


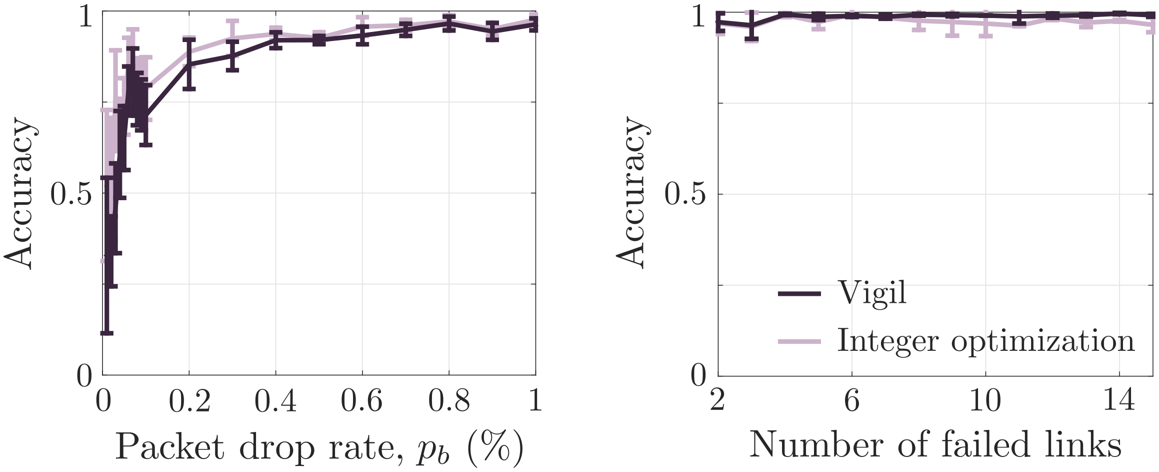
Figure 3.4: Testing Algorithm 1 when Theorem 2 holds.

observe that Vigil is still able to maintain a high level of accuracy ($\geq 80\%$), on par with the optimal solution.

Multiple Failures. A noted shortcoming of past work has been in identifying problematic links in the network where the failed links have very different drop rates [119]. We test Vigil's ability to find the cause of packet drops in the presence of such failures. Figure 3.5b illustrates this point for different number of failed links under the default drop rate setting described in the simulation setup. Vigil is almost always successful at identifying the link responsible for a packet drop. In fact, it *surpasses* the optimization algorithm (albeit slightly) in §3.5.3. This is due to the optimization's susceptibility to lone packet drops due to noise.

3.6.3 Impact of Noise

Single Failure. We next vary the level of noise in the network to see how well Vigil can attribute the cause of packet drops in the presence of various degrees of noise. This is done by changing the drop rate of the good links across different ex-



(a) Single failure

(b) Multiple failures

Figure 3.5: Vigil's accuracy for varying drop rates.

perimental runs. Figure 3.6a shows the accuracy results in the case of a single link failure as the drop rate varies. Higher noise levels could make it harder for Vigil to detect the link failure through Algorithm 1. However, as shown in Figure 3.6 they have little impact on its ability to find the cause of packet drops on individual connections.

Multiple Failures. We repeat this experiment for the case of 5 failed links. Figure 3.6b shows the results. Again, Vigil shows little sensitivity to the increase in noise when identifying the cause of per-connection packet drops. Note that the large confidence intervals of the integer optimization problem show its high sensitivity to noise.

3.6.4 Varying Number of Connections

In previous experiments, all hosts opened 2 connections per second. Here, we allow hosts to choose the number of connections they create in each epoch uniformly at random between $(10, 60)$. Recall from Theorem 7 that a larger number of connections from each host helps Vigil improve its accuracy.

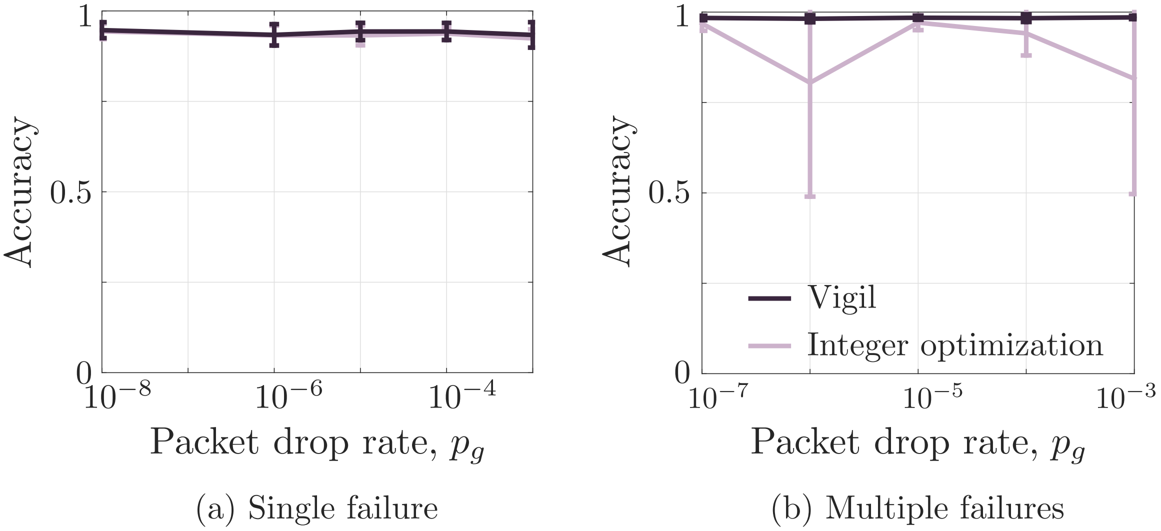


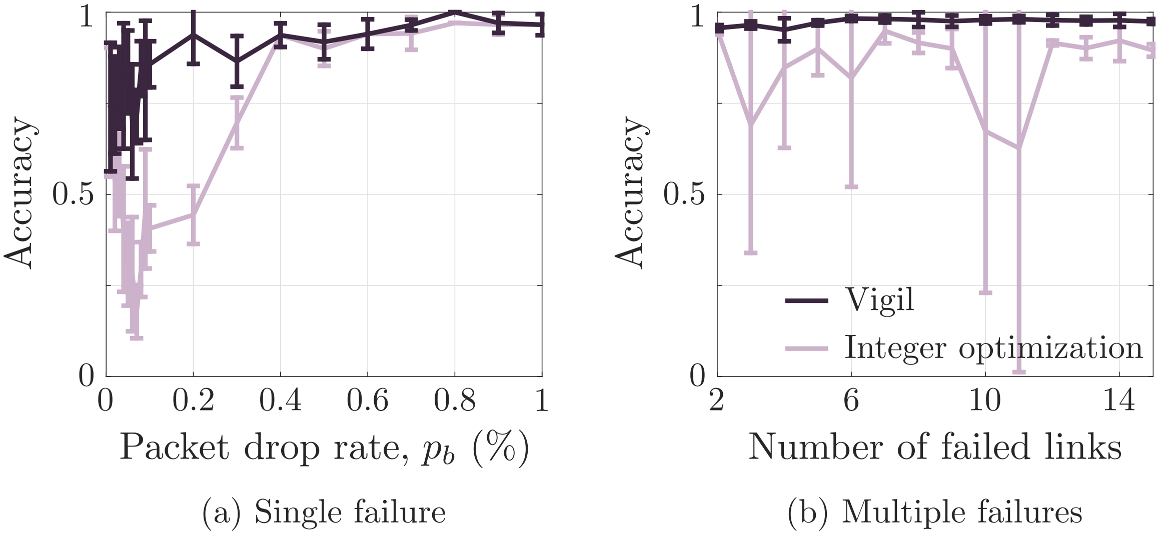
Figure 3.6: Vigil's accuracy for varying noise levels. Lone/sporadic drops are not of interest to a network provider/operator. Vigil can successfully ignore such drops and continue to perform well in the presence of high degrees of noise.

Single Failure. Figure 3.7a validates this effect for the case of a single filed link. Our results show that Vigil accurately identify the cause of packet drops on connections with high probability. Vigil also significantly outperform the optimization when the failed link has a low packet drop rate. This is because the optimization has multiple optimal points and is not sufficiently constrained.

Multiple Failures. Figure 3.7b repeats the same experiment as above, but we vary the number of failed links under the default drop rate settings. The integer optimization problem suffers from the lack of information to constrain the set of results. It therefore has a huge variance (shown in terms of confidence intervals). Vigil on the other hand maintains high probability of detection (at low variance) no matter the number of failed links.

3.6.5 Impact of Traffic Skews

Single Failure. We next demonstrate Vigil's ability to detect the cause of connection packet drops even when traffic is heavily skewed. We pick 10 ToRs at random



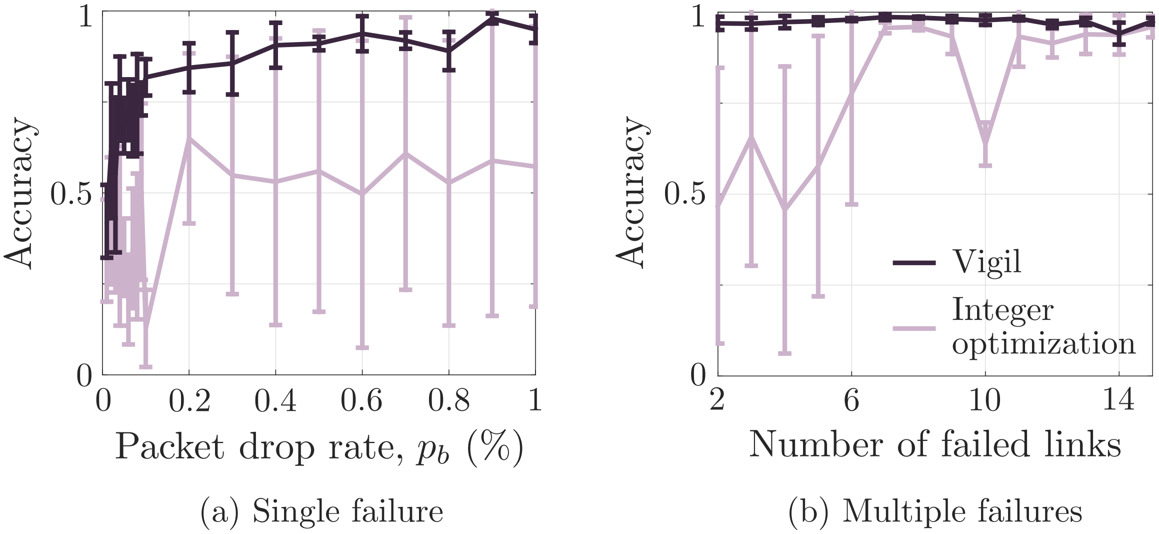
(a) Single failure

(b) Multiple failures

Figure 3.7: Vigil's accuracy for varying number of connections. Each host opens between $(10, 60)$ connections. Where the number is chosen unifromly at random.

from the set of all ToRs in the network (25% of the ToRs). To skew the traffic, 80% of the connections have destinations set to hosts under these 10 ToRs, an the remaining connections route to hosts randomly chosen in the network. Figure 3.8a shows that despite the traffic skews, the optimization is much more heavily impacted by the skew than Vigil. Vigil continues to detect the cause of packet drops with high probability ($\geq 85\%$) for drop rates higher than 0.1%.

Multiple Failures. We repeated the above experiment for multiple failures. Figure 3.8b shows that the optimization's accuracy suffers. It consistently shows a low detection rate as its constraints are not sufficient in guiding the optimizer to the right solution. On the other hand ,Vigil maintains a detection rate of $\geq 98\%$ at all times.



(a) Single failure

(b) Multiple failures

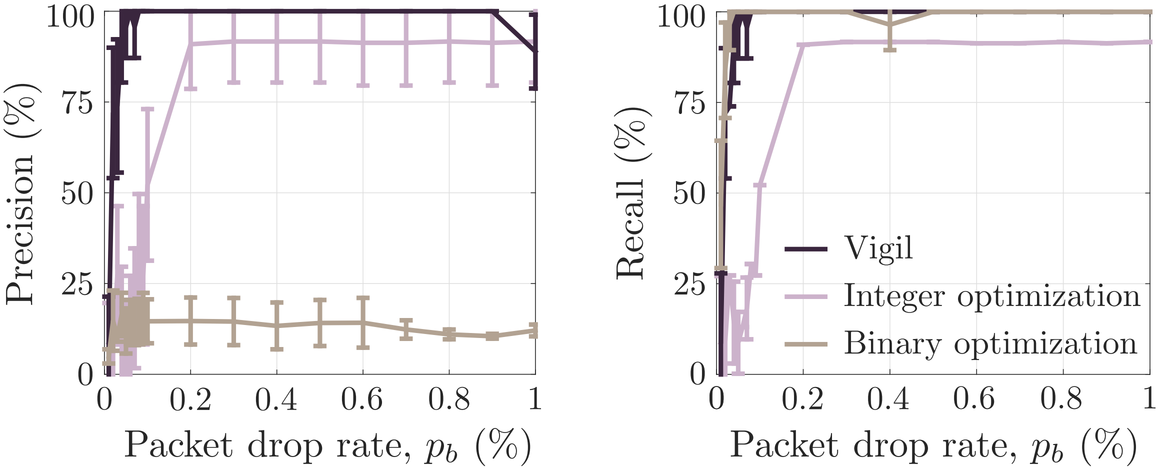
Figure 3.8: Vigil's accuracy under heavily skewed traffic. The large confidence intervals of the optimization problem are a reflection of its sensitivity to noise.

3.6.6 Detecting Bad Links

In all our previous experiments, the focus has been on detection accuracy from each connection's point of view. In our next experiment, we evaluate Vigil's ability to detect bad links using the metrics of recall and precision.

Single Failure. Figure 3.9 shows the results for finding the failed link in the network using Algorithm 1. In the presence of a single failure, our approach outperforms the optimization algorithm as it does not require a fully specified set of equations in order to provide a best guess as to which link is failed.

Multiple Failures. In this scenario, the drop rates on the failed links are heavily skewed. More specifically, at least one failed link is assigned a drop rate in the range of $(10\%, 100\%)$ while all others have a drop rate in $(0.01\%, 0.1\%)$. This scenario is one where past approaches have reported as potentially hard to detect [119]. Figure 3.10 shows that Vigil can detect up to 7 failures with recall/precision above 90%. However, its recall drops as the number of failed links increase. This is due to the fact that the increase in the number of failures drives up the votes of all other links increasing the cutoff threshold and therefore increasing the likelihood of false negatives. In fact if instead the top k links had been selected Vigil's recall would have been close to 100% [1]. Further evaluations can be found in [1].



(a)

(b)

Figure 3.9: Algorithm 1 with single failure. Vigil can accurately detect the cause of problems with recall/precision above 90%. However, its recall drops as the number of failed links increase.

3.6.7 Effects of Network size

In our final experiment, we explore how Vigil performs as the network size grows. We observe that its accuracy was on average 98% when the network consisted of 1 pod, 92% for 2 pods, 91% for 3 pods, and 90% for 4 pods when finding a single link failure. In contrast, the optimization problem (3.3) had an average accuracy of 94%, 72%, 79%, and 77% for the respective number of pods.

We further evaluate both Vigil and the optimization's ability to find the cause of per connection drops when the number of failed links in the network was ≥ 30. We observe that both approach's performance remained unchanged for the most part, e.g., the accuracy of Vigil in identifying drop causes in an example with 30 failed

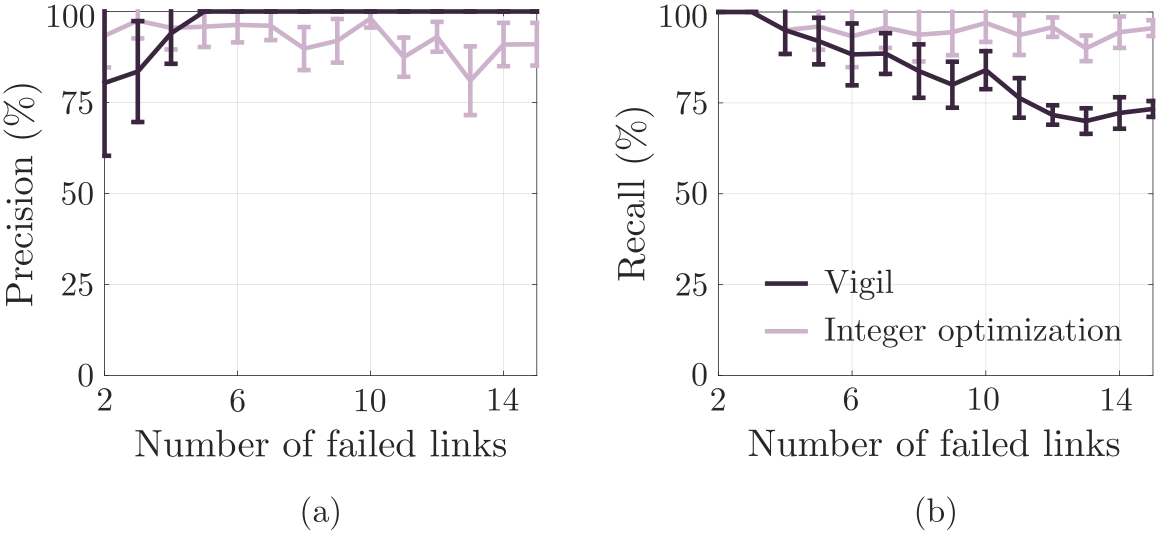


Figure 3.10: Algorithm 1 with multiple failures. The drop rates on the links are heavily skewed. Prior work have noted the difficulty of detecting links with high drop rates in such scenarios. Vigil, however, continues to exhibit high precision/recall.

links is 98.01%. Figure 3.11 shows the performance (recall) of using Algorithm 1 as the pod sizes vary. We observe that recall stayed high even as the network increases in size. The precision was always 100% and we omit the graph for brevity.

3.7 Evaluations: Test Cluster

We next evaluate Vigil on the more realistic environment of a test cluster consisting of 10 ToRs with a total of 80 links. We only control 50 hosts in the cluster, while others are production machines. Therefore, the T_1 switches see real production traffic. We recorded 6 hours of TCP traffic from a host in production and replayed that traffic from our hosts in the cluster (with different starting times varying by 1-2 minutes). Using Everflow-like functionality [145] on the ToR switches, we induced different rates of packet drops on T_1 to ToR links. Our goal is to identify the cause

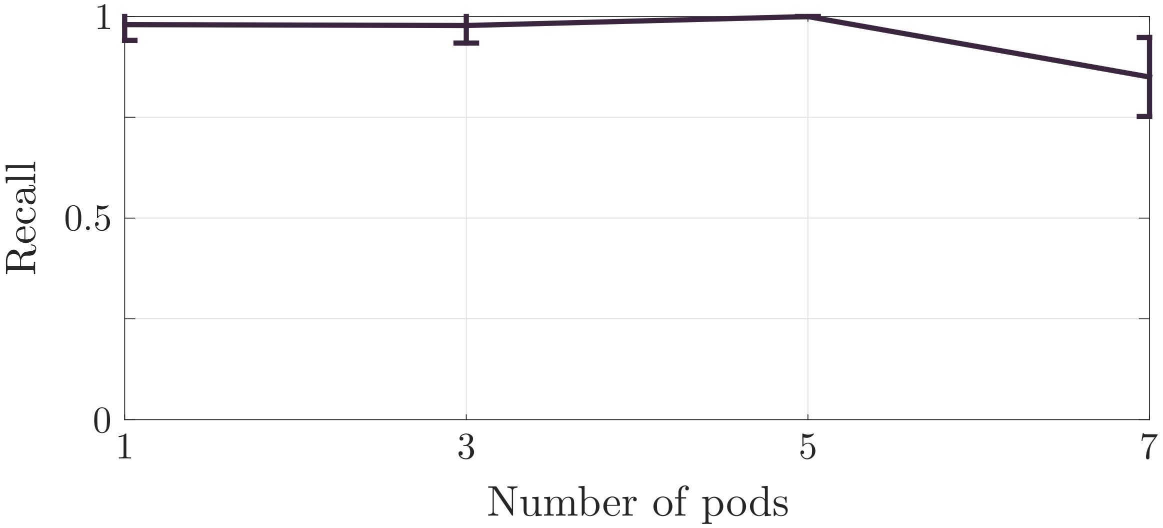


Figure 3.11: Recall of Algorithm 1 for different numbers of pods. A six pod network consists of 12480 links. In all cases Vigil continues to have Recall higher than 90%. Note, that its precision is 100% for all cases.

of packet drops on each TCP connection §3.7.2 and to validate whether Algorithm 1 works in practice §3.7.3.

3.7.1 Clean Testbed Validation

We first validate a clean testbed environment. We repave the cluster by setting all hosts and ToR switches to a clean state. We then run Vigil without injecting any failures. We observe that in the newly-repaved cluster, links arriving at a particular ToR switch had abnormally high votes, namely 22.5 ± 3.65 in average. We thus suspected that this ToR is experiencing problems. After rebooting it, the total votes of the links went down to 0, validating our suspicions. We then proceed with our controlled experiments.

3.7.2 Per-connection Link Failure Analysis

We aim to quantify the accuracy of Vigil in identifying the cause of packet drops when links have very different drop rates. We induce a drop rate of 0.2% and 0.05% on two different links in the network over an hour period. Since we do not know the ground truth when the connection path does not go through either of the two links, we only consider connections that go through at least one of the two failed links. In 90.47% of the connections seen in a one hour period, Vigil was able to attribute the packet drop to the correct link (the one with higher drop rate).

3.7.3 Identifying Failed Links

We next validate Vigil's ability to correctly identify a failed link in the presence of a single failure. We inject different packet drop rates on a chosen failed link and seek to determine whether there is a correlation between total votes and drop rates. Specifically, we look at the difference between the vote tally on the bad link and that of the most voted good link. We induced a packet drop rate of 1%, 0.1%, and 0.05% on a T_1 to ToR link in the test cluster.

Figure 3.12 illustrates the distribution for the various drop rates on the bad link. As the figure indicates, the failed link has the highest vote out of all links when the drop rate is 1% and 0.1%. When the drop rate is lowered to 0.05%, the failed link becomes harder to detect due to the smaller gap between the drop rate of the bad link and that of the normal links. Indeed, the bad link only has the maximum score in 88.89% of the instances (mostly due to occasional lone drops on healthy links). Nevertheless, it is always one of the 2 links with the highest vote.

Figure 3.12 also shows the high correlation between the probability of packet drop on a links and its vote tally. Note that this trivially shows that Vigil is 100% accurate in identifying the cause of packet drops on each connection given a single link failure: the failed link has the highest votes among all links. We compare Vigil with the integer optimization problem in (3.3). We find that the latter also returns the correct result every time, albeit at the cost of a large number of false positives. To illustrate this point: the number of links marked as bad by (3.3) is 1.54 ± 0.09, 1.18 ± 0.04, and 1.47 ± 0.13 times higher than the number given by Vigil for the drop rates of 1%, 0.1%, and 0.05% respectively (averaged across each 30-seconds epoch in a one hour period).

Finally, we investigate Vigil's ability to identify multiple link failures. This is a harder experiment to configure due to the smaller number of links in this test cluster and hence lower path diversity. We induce different drop rates ($p_1 = 0.2\%$ and $p_2 = 0.1\%$) on two links in the test cluster. We note that the link with higher drop rate is the most voted 100% of the time. The second link, however, is the second highest ranked 47% of the time and the third 32% of the time. Still, it always remained in the top 5 links. This shows that by allowing a single false positive (in this case, identifying three instead of two links), Vigil can detect all failed links 80% of the time even in a setup where the traffic distribution is skewed. This is something past approaches [119] could not achieve. Note that in this example, Vigil identifies the true cause of packet drops on each connection 98% of the time.

3.8 Evaluations: Production

We have deployed Vigil in one of our production data centers. In this section, we share some of our observations from this deployment. We note that the number of traceroutes generated by Vigil never exceeded T_{max} during its one month deployment in production.

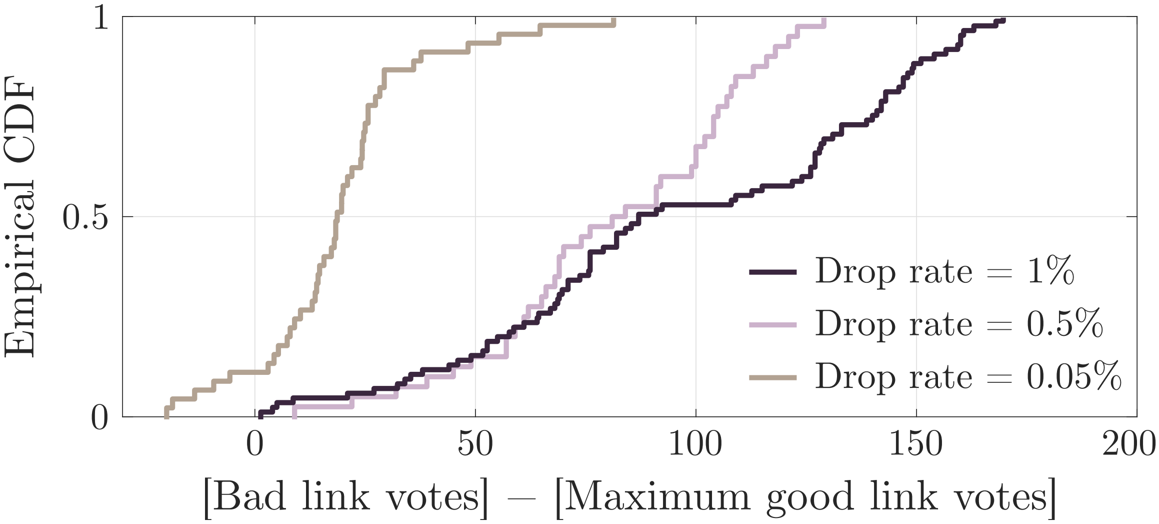


Figure 3.12: Distribution of the difference between votes on bad links and the maximum vote on good links for different bad link drop rates. The numbers clearly show a large correlation between the drop rates induced and the votes on the link dropping packets.

3.8.1 Comparison to EverFlow

Knowing the true nature of failures in a production data center is hard. Therefore, we perform a semi-controlled experiment in one of our production data centers to test the efficacy of Vigil. The cluster consists of thousands of hosts/links. This provides evidence that Vigil works well in practice. To find the "ground truth", we compare its results to that obtained by an EverFlow-like system, hereby referred to as EverFlow, deployed in our production data center (this system is not deployed in our test cluster and could not be used in that setting). EverFlow captures all packets going through each switch on which it was enabled. Therefore, it is expensive to run for extended periods of time. We thus only run EverFlow for a period of 30 minutes and configure it to capture all outgoing IP traffic from two random hosts in the cluster. Furthermore, we specifically capture connections to our storage service. We filter all connections that were detected to have at least one retransmission during this time. We also tracked their packets and find where any where dropped (using EverFlow). We then check whether the detected link matches that found by Vigil. We found that Vigil was accurate in every single case.

3.8.2 Finding The Cause of VM Reboots

During our deployment, there were 278 VM reboots in the cluster for which there was no explanation. Vigil identified a link as the cause of problems in each of these instances. In 261 cases, there was high drop rates on the host to ToR link. In another 15, the endpoints of the identified links were undergoing configuration updates. In the remaining 2 instances, the link was flapping.

3.9 Discussion

A number of novel innovations are enabled by Vigil. As an example, using Vigil, the host can determine a *healthy* DIP pool to use at each point in time. This is because due to encapsulation, the path of each connection is completely determined by the DIP. Thus, Vigil can potentially help avoid unhealthy path. Vigil is deployed in our production data center. A number of additional factors to consider when using Vigil include:

Source NATs. Source network address translators (SNATs) change the source IP of a packet before it is sent out to the network to a VIP. The ICMP messages will have to carry the right source address for Vigil to get the response to its traceroutes. This can be enabled through a query to SLB. Details are omitted.

ACK Loss on reverse path. In rare cases, it is possible that the packet loss on the reverse path is so severe that loss of ACK packets trigger timeout at the sender. If this happens, the traceroute would not be going over any link that triggered the packet drop. Since TCP ACKs are cumulative, this is typically not a problem unless loss rates are very high. Spurious retransmissions triggered by timeouts may also occur if there is sudden increase delay on forward or reverse paths. This can happen due to a variety of reasons including rerouting, or large queue buildup. Currently, we treat these retransmissions like any other.

Vigil's dependency on network topology. The calculations in §3.5 take advantage of the fact that our network topology is known and is similar to a Clos network. However, the same calculations can be repeated for *any* known topology. The accuracy of Vigil is heavily tied to the degree of path diversity in the network and that multiple path are available at each hop: the higher the degree of path diversity, the better Vigil performs. This is also a desired property in any data center topology [113].

Header specific drops. Vigil may result in a large number of false positives when a packet drops are *header specific*, either due to policy misconfigurations or hardware bugs. In this case, packets may be dropped for one connection, while other connections along the same path are unharmed. Vigil can only narrow down the cause of these drops to all the links along the path without being able to localize the device itself. Such problems are rare, Vigil still provides useful information to the operator in narrowing down the set of devices to be investigated.

Vigil's ranking. Vigil's ranking approach will naturally bias towards the detection of failed links that are frequently used. This is an intentional design choice as the goal of Vigil is to identity high impact failures that affect many connections.

Finding the path of all connection. One may think that if one knew the topology, and the details of the ECMP functions on all the routers, the path of a packet can be identified by simply inspecting its header. However, ECMP functions are typically proprietary and also have initialization "seeds" that change with every reboot of the switch. More importantly, ECMP functions change in the face of link failures and recoveries. Keeping track of all link failures/recoveries in real time is not feasible at data center scale.

Diagnosing VM traffic problems. Vigil's current goal is to find the cause of packet drops on infrastructure connections (and through those find the failed links in the network). In principle, we can build a Vigil-like system to diagnose TCP failures for connections established by customer VM's as well. However, a number of problems need to be solved to build such a system. Including security and scalability issues. This is part of our future work.

3.10 Conclusion

In this chapter we presented Vigil, an always on and scalable monitoring/diagnosis system for data center networks that is completely contained within the end hosts. Vigil can accurately identify drop rates as low as 0.05% in data centers with thousands of links through monitoring the status of ongoing TCP connections and "informed path discovery". It provides a ranking of links that allows for finding the most likely cause of packet drops on each individual TCP connection. We demonstrate mathematically, in simulations, and empirically in a production environment that Vigil continues to provide high accuracy (recall and precision) in the presence of noise. This is achieved while rate limiting path discovery messages to stay within production limits.

Both NetPoirot and Vigil allow end hosts to assist operators in the diagnosis process. However, such diagnosis (as well as recovery) takes time. It is important to maintain high QoS while recovery is in progress. In the following chapters we will investigate the efficacy of multipath approaches in allowing clients to circumvent failures if/when possible in various settings.

Chapter 4

Resilience to Failures at the Endpoints Through Multipath TCP

Our work in Chapters 2 and 3 aims to enables clients to assist in identifying the cause of failures when they occur. However, failures are disruptive and it's desirable to shield users from their impact. With the increasing number of users, providing performance guarantees to these users is becoming increasingly difficult. Similar reasons to those described in earlier chapters (lack of cooperation between the various organizations as well as high reaction times) reduce the effectiveness of distributed solutions to this problem. Thus, additional algorithms/protocols at the endpoints can help further improve client's resilience to such problems.

Multipath solutions are one such solution, as it is unlikely that all path should fail simultaneously. They have been extensively studied in the past e.g., [9, 105] in the context of datacenters and the Internet. For example, Multipath TCP (MPTCP) [105] has allowed end users to leverage multiple path at the transport layer in order to connect through multiple parallel paths. The protocol is designed to move traffic away from congested paths onto less congested ones using its joint congestion control algorithm [78]. Significant progress has been made in understanding the protocols behavior in terms of fairness, stability, etc. However, a complete understanding of the protocol, its performance, and especially the implications of MPTCPs joint congestion control algorithm on other modules in the protocol is lacking. In our work in [14], we observed various interesting (and in some cases unintuitive) behaviors by the protocol. Such behaviors show that a complete understanding of multipath protocols is still lacking. In this work we focus mainly on investigating such behavior in wide area networks, as most are fundamental to the protocol itself (irrespective of the setting in which it is used). Extending these investigations to the context of datacenter networks is the subject of future work.

4.1 The Shortcomings of MultiPath TCP

4.1.1 Motivation

Multipath TCP (MPTCP) [116] is an emerging protocol that enables transmissions over multiple paths, overcoming potential single path capacity limitations. These benefits are fueling a growing adoption with, for example, Apple iOS 7 now supporting MPTCP. The protocol is complex with many interacting parameters, and has been the focus of numerous empirical [107, 116, 13] and analytical [112, 135] studies. A complete understanding of the interactions of its many parameters, and in particular those associated with its complex congestion control algorithm, is lacking.

We make the following contributions:

Empirical evaluation. the Mininet [95] emulator, we study the impact of the choice of initial path in establishing an MPTCP connection. We uncover some surprising outcomes – that the choice of the initial path can not just have a lasting impact on MPTCP performance, but that in some cases, starting with an inferior path (with higher RTT) can improve overall throughput.

Analytical model. We develop a stylized analytical model to investigate this behavior based on the partitioning of MPTCP's congestion control into two distinct phases. Using this model and numerical analyses, we elucidate and validate empirical observations, and identify the non-linear coupling between paths introduced by MPTCP's congestion control as a major contributor.

4.1.2 Background and Related Work

MPTCP extends TCP with the ability to transmit on multiple paths. For backward compatibility purposes, MPTCP presents applications a TCP-like socket abstraction. The protocol then establishes and maintains multiple subflows, one on each path. A representative implementation of MPTCP [116] incorporates an API to add and remove paths from the set of available selections. The MPTCP layer handles out-of-order packet arrivals and congestion control across sub-flows. MPTCP's congestion control algorithm was designed with the following objectives: (1) Perform at least as well as a single TCP session on the best path; (2) Be fair to other TCP flows; and (3) Move traffic away from congested paths to less congested ones when possible.

To satisfy these goals, MPTCP performs *joint* congestion control over its multiple paths, which operates exactly as TCP would when in slow start or fast retransmit. However, MPTCP couples tuning of the congestion windows of each sub-flow during congestion avoidance.

A significant body of work has been devoted to characterizing and optimizing MPTCP's coupled congestion control. The authors of [112] perform a theoretical analysis of the problem, characterizing factors governing responsiveness, TCP-friendliness, and window oscillations of a joint congestion control method for MPTCP. Other works [123, 78, 135] focus on improving MPTCP's joint congestion control.

4.1.3 Empirical Evaluation

MPTCP's throughput is highly dependent on its ability to appropriately schedule transmissions across paths [13]. The scheme used in the current implementation of MPTCP favors lower RTT paths, since this in turns reduces the receive buffer size (lowering the memory footprint at the receiver). However, a comprehensive understanding of the interactions between this selection and congestion control is still lacking. To gain an initial understanding, we perform an empirical evaluation using an open-source MPTCP implementation [116] over the Mininet [95] emulator on a Ubuntu (version 3.11) OS. These experiments as well as our analysis, rely on the *Linked Increase Algorithm (LIA)* [135] as the coupled congestion control algorithms, but the results should apply to other schemes.

Experiment Setup

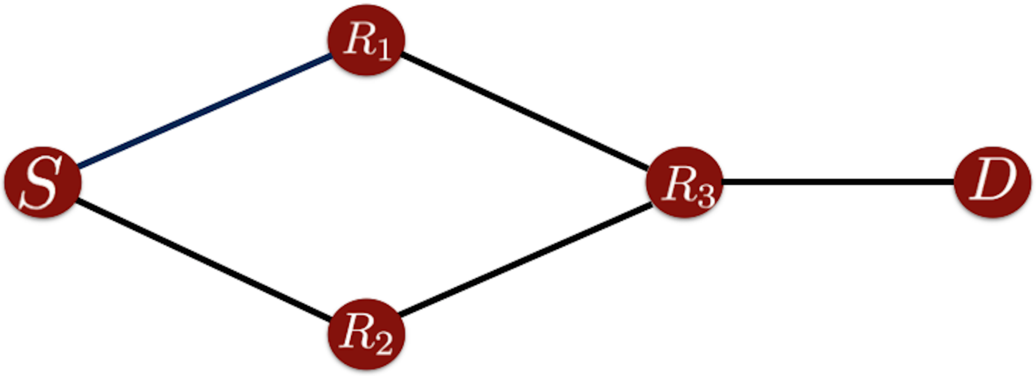


Figure 4.1: Mininet topology.

Figure 4.1 shows the simple topology used in our experiments. It consists of a source S and destination D. S is dual-homed (connected to routers R_1 and R_2). The link R_3/D is added so that D is single-homed. Consequently, there are only two candidate subflows $S \to R_1 \to R_3 \to D$ and $S \to R_2 \to R_3 \to D$. Since the main purpose of the $R3/D$ link is to make D single-homed, its propagation delay is set to 0. All other links have the same propagation delay and are assumed to be lossless.

In the rest of the paper, C_1 and C_2 denote the capacity of the first and second subflows respectively. We vary C_1 and C_2 during the experiments, and set the capacity between R_3 and D to be high enough $(100Mbps)$ to avoid it becoming the bottleneck. Each experiment runs for at least 200 seconds to allow MPTCP to reach steady state. The *iperf* tool is used to measure throughput, which we define as the total number of bytes (MB) transmitted successfully for each experiment's duration. To avoid interference across experiments, we turn off TCP route caching in Linux.

4.1.4 Impact of Path RTT on Throughput

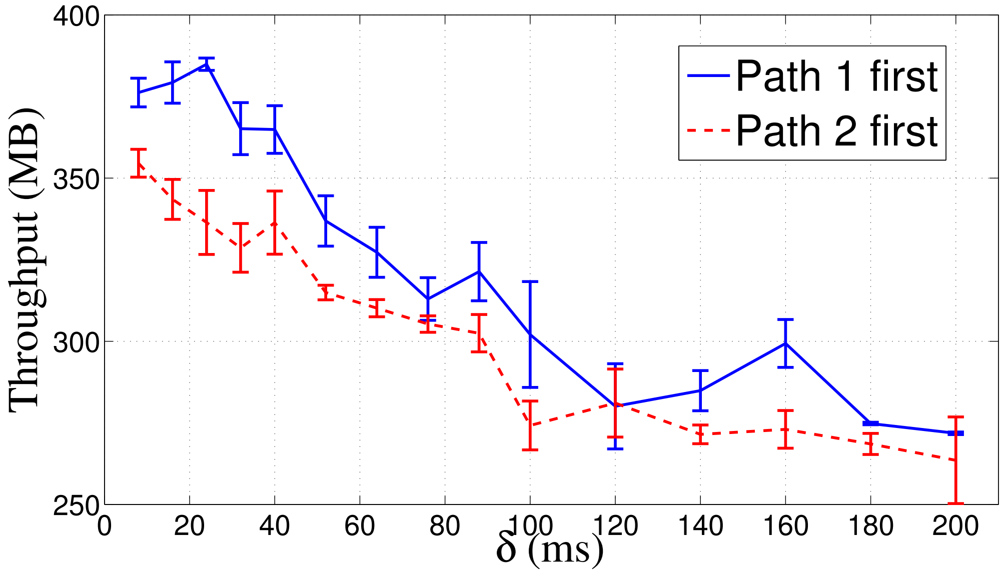


Figure 4.2: MPTCP throughput (MB) as path RTT difference increases.

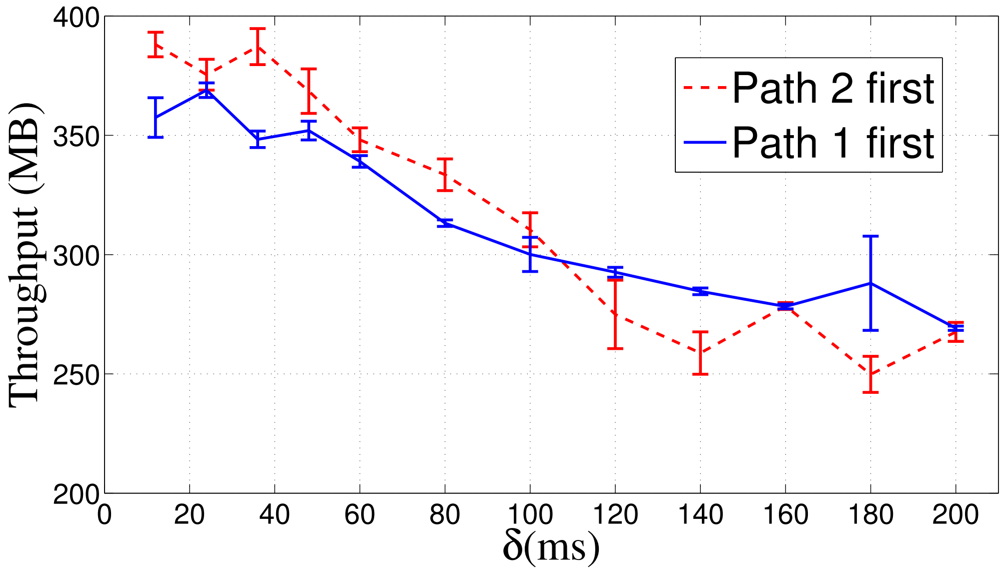


Figure 4.3: MPTCP throughput (MB) as path RTT difference increases.

Across experiments, the RTT of each subflow varies as follows. We fix τ_1, which is the RTT of subflow $S \rightarrow R_1 \rightarrow R_3 \rightarrow D$. The RTT of subflow 2, τ_2, is varied and always set to be larger than τ_1, with the difference denoted as δ. In all experiments, we set C_1 and C_2 to be $10Mbps$, τ_1 to be $200ms$, and vary δ from $10ms$ to $200ms$.

For each experiment, either the path with lower RTT τ_1 (*Path 1*) starts first, or the higher RTT path (*Path 2*) does. This is achieved by configuring the routing tables and deactivating the interface on the later path for the duration of the start time-lag. Note that no matter which path starts first, MPTCP's built in scheduler prioritizes the lower RTT path. Figures 4.2-4.3 summarize the MPTCP throughput (MB) as δ increases, across two sets of experiments. Each experiment is carried out over 15 runs, and we present average throughput results with 95% confidence interval bars to account for variations in CPU load.

Figure 4.2 corresponds to a scenario where both paths start immediately one after the other. Irrespective of which path is selected to start first, the lag between their respective establishment depends solely on the RTT of the second path, as it affects the duration of the three-way TCP handshake. The figure shows that when the lower RTT path is selected first (solid blue line labeled as *Path 1 First*), MPTCP's throughput is higher than under the *Path 2 First* scenario (dotted red line). This is as expected, given the scheduler's preference for the lower RTT path and its initial higher throughput.

Figure 4.3 shows our next set of results, where we add a 4 seconds time lag between the time the initial path is chosen, and when the second path is made available. This produces a *non-intuitive* result, namely, that when the longer RTT path is selected first, MPTCP's throughput is actually *higher* for δ values up to $100ms$! Beyond $\delta = 100ms$, *Path 1 First* begins to exhibit higher throughput again. We observed similar results when increasing the time lag to 6 seconds. Note also that the overall throughput is higher in this scenario than in the previous one.

4.1.5 MPTCP Model

Our empirical results in Section 4.1.3 suggest that whenever multiple subflows of different RTT are available, MPTCP throughput will depend on the initial path being chosen, and sometimes, picking the path with higher RTT as the starting point results in higher throughput – a non-intuitive observation. In this section, we present a stylized analytical model of MPTCP, to explain the behavior as a consequence of MPTCP's coupled congestion control.

4.1.6 Setting Up The Problem

Our model is based on a generalization of the topology in Figure 4.1, where we assume two paths 1 and 2, with an arbitrary number of intermediate routers. Each path, i, is assumed to have a bottleneck capacity C_i. The two paths are disjoint, and all queues in the network use the Drop-Tail policy.

In the analysis that follows, the congestion window of path i at time t is denoted[1] as $w_i(t)$, and the transmission rate on that path is represented as $x_i(t)$. We denote the round trip time on path i as τ_i. Note that τ_i can be considered as the sum of the two-way propagation delay on path i and the added queueing delay. We take both path's propagation delays to be constant[2].

Without loss of generality, we assume that in the absence of queueing delay $\tau_1 \leq \tau_2$ or equivalently $\tau_2 = \tau_1 + \delta$. Consistent with the Linux kernel implementation, the `ssthresh` is zero, which means that the connection begins in congestion avoidance. Upon reception of each acknowledgment on path i, MPTCP's congestion window is increased by $I_i(\mathbf{w}_s(t))$, where $\mathbf{w}_s(t) = (w_1(t), w_2(t))$.

Based on the LIA coupled congestion algorithm:

$$I_i(\mathbf{w}_s(t)) = \min\left(\frac{\max(\frac{w_1(t)}{\tau_1^2}, \frac{w_2(t)}{\tau_2^2})}{(\frac{w_1}{\tau_1} + \frac{w_2}{\tau_2})^2}, \frac{1}{w_i}\right). \tag{4.1}$$

We divide the operation of MPTCP subflows into two modes, each of which is modeled separately in Sections 4.1.8 and 4.1.9, and used to characterize the effect of MPTCP's choice of "first" path in initiating the connection.

4.1.7 The Two Modes of an MPTCP Subflow

Depending on congestion window size, each subflow's congestion control mechanism is divided into two modes.

Mode 1. When MPTCP subflows have a relatively small initial window, each subflow i will start by operating in mode 1. Here, an MPTCP subflow on path i can transmit up to $w_i(t)$ packets before it has to stop and wait for acknowledgments. The acknowledgment for a packet transmitted at time t over path i arrives at the

[1] We further assume that the source and destination have sufficient memory available such that TCP's congestion window is equal to the sending window and that the send buffer is large enough so that there is always data available to be sent if the sending window allows it.

[2] Variations in round trip times are due to variations in queueing delay.

source at time $t + \tau_i$. Thus, if a subflow's congestion window is smaller than the bandwidth delay product (BDP) of its path i.e., if $w_i(t) \leq \tau_i C_i$, the subflow remains idle until acknowledgments for the current window transmissions start arriving. When a subflow is operating in mode 1, it transmits at a rate below that of the bottleneck link. Thus, all queues along the subflow's path remain empty.

Mode 2. As the congestion window increases, if $w_i(t) \geq \tau_i C_i$ acknowledgments now arrive periodically, at regular intervals determined by the bottleneck capacity, C_i, the MPTCP subflow enters mode 2. This means that the subflow can continue transmission without interruption (until a packet is lost). In mode 2, the subflow is able to transmit at a rate higher than C_i allowing for the queue at the bottleneck link to grow. This results eventually in a dropped packet, as well as progressive increases in the path's observed round trip time until the packet drop. Note that depending on $w_i(t)$, at time t when the loss occurs, it is possible for the subflow to then return to mode 1.

4.1.8 Modeling The MPTCP Subflow in Mode 1

In mode 1, the subflow i's transmission rate is limited by its congestion window, *and* $x_i(t) \simeq \frac{w_i(t)}{\tau_i}$. This allows us to use the traditional fluid model [77], to model its behavior.

Subflow i is operating under the condition $w_i(t) \leq \tau_i C_i$. Since the congestion window is not large enough to use the entire capacity available on path i, upon transmission of $w_i(t)$ packets the subflow waits until acknowledgments arrive before it can send any more packets. Thus, the MPTCP subflow can be considered as sending $w_i(t)$ packets every τ_i seconds $(x_i(t) \simeq \frac{w_i(t)}{\tau_i})$. Since the subflow is transmitting at a rate below C_i, queues remain empty and there is no queueing delay.

Thus, following the approach in [112], we can model the change in congestion

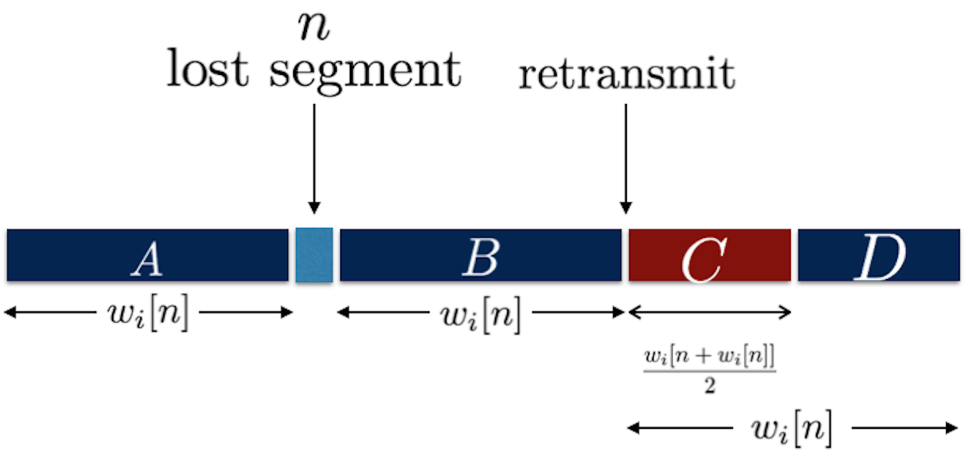


Figure 4.4: The NewReno fast retransmission algorithm. There is no transmission on the subflow during time block C.

window over a period of τ_i seconds, which we denote as ∂w_i, as follows:

$$\partial w_i = I_i(\mathbf{w}_s) w_i = \frac{\partial w_i}{\partial t}\tau_i. \tag{4.2}$$

Using (4.2) and the relation $x_i(t) \simeq \frac{w_i(t)}{\tau_i}$ we can write

$$\frac{\partial x_i}{\partial t} = \frac{x_i(t)}{\tau_i} I_i(\mathbf{w}_s), \tag{4.3}$$

which describes the transmission rate on flow i in mode 1.

This model will not be valid in mode 2, since not only $x_i(t) \neq \frac{w_i(t)}{\tau_i}$, but also we can no longer assume that τ_i is constant. Furthermore, it is possible for an MPTCP subflow in mode 2 to experience losses due to an overflow of the queue at the bottleneck link.

4.1.9 Modeling The MPTCP Subflow in Mode 2

When operating in mode 2, the MPTCP subflow has a congestion window that satisfies $w_i(t) \geq \tau_i C_i$. The subflow is no longer able to send $w_i(t)$ packets every round trip time, since τ_i is not enough time for all the acknowledgments of the previously transmitted window to arrive. Thus, the model described in Section 4.1.8 cannot be used to describe an MPTCP subflow's behavior in this situation.

The MPTCP subflow is allowed to transmit at a rate progressively higher than C_i (although acknowledgments are clocked, the continued increase in window size allows the transmissions of additional packets in each roundtrip). Therefore, the queue at the bottleneck link grows, eventually resulting in a packet loss as well as a steady increase in observed round trip time.

We denote the capacity of the queue on path i as N_i and the number of packets in the queue at time t as $l_i(t)$. Using this notation, and given that the rate of change in $l_i(t)$ is equal to the difference in the queue's input and output rates

$$\frac{\partial l_i}{\partial t} = x_i(t)\mathbf{1}_{l_i(t)<N_i} - C_i\mathbf{1}_{l_i(t)>0}, \tag{4.4}$$

where $\mathbf{1}_x$ is an indicator function evaluating to 1 when x is TRUE and 0 otherwise, ensuring that no packets are added to the queue once it reaches its maximum size and that it is not drained when $l_i(t) = 0$. Since propagation delays are constant, the rate of change in path i's RTT, τ_i, can be written as:

$$\frac{\partial \tau_i}{\partial t} = \frac{x_i(t)}{C_i}\mathbf{1}_{l_i(t)\leq N_i} - \mathbf{1}_{l_i(t)>0}. \tag{4.5}$$

Given that ACKs arrive periodically with a period $T_i = \frac{1}{C_i}$ in mode 2, we can assume that changes in $w_i(t)$ and $x_i(t)$ occur at equally spaced points in time $t = kT_i$ where $k \in \{1, 2, 3 \dots\}$. Thus, we can use a discrete time system with a step size of T_i to model the subflow's behavior and then generalize that model by setting $w_i(t) = w_i[k]$ and $x_i(t) = x_i[k]$ for k satisfying $kT_i \leq t < (k+1)T_i$, where [ ] is used to denote the discrete time counterparts of $x_i(t)$ and $w_i(t)$.

Therefore, in the absence of losses

$$x[k] = C_i + I_i(\mathbf{w}_s)C_i \tag{4.6}$$

$$w[k] = w[k-1] + I_i(\mathbf{w}_s). \tag{4.7}$$

We assume that notifications of losses arrive only due to triple duplicate ACKs, at which point the subflow enters fast retransmit [67]. At any time t, there are $w_i(t)$ outstanding packets while the subflow is in mode 2. We can use this to describe the subflow's behavior during fast retransmit. This is illustrated in Figure 4.4.

The end of block A in Figure 4.4 marks $t = nT_i$ where a packet was lost. At that time $w_i[n]$ packets remain unacknowledged on path i. The ACK which arrives[3] at the end of block B at $t = (n + w_i[n])T_i$, marks the arrival of the first of the triple duplicate ACKs. We approximate the arrival of the third duplicate ACK to be the same. At this time, the congestion window is halved and `ssthresh` is set to $\frac{w_i[n+w_i[n]]}{2}$.

At the beginning of block C, the congestion window is set to the reduced value $\frac{w_i[n+w_i[n]]}{2}$. Given that $w_i[n + w_i[n]]$ packets are outstanding at that time, $\frac{w_i[n+w_i[n]]}{2}$ packets need to arrive before the subflow can restart transmission. This occurs at the end of block C, i.e., $t = (n + w_i[n] + \frac{w_i[n+w_i[n]]}{2})T_i$. Finally, the acknowledgment for the lost packet arrives at the end of block D at $t = n + 2w_i[n]$, ending fast retransmission.

To model MPTCP's behavior during fast retransmission, we use $q_i[n]$ as the indicator variable set to 1 if a packet is dropped at $t = nT_i$ on path i and is zero otherwise, i.e., $q_i[n] = \mathbf{1}_{l_i(nT_i)=N_i}$. Note that a packet lost at time $t = nT_i$ implies[4]

$$w_i[n] = \tau_i[0]C_i + N_i. \tag{4.8}$$

Using the indicator function $q[k]$ we can model the congestion window on path

[3] Notice that the transmission time of a packet over path i is equal to T_i, which means that $w_i[n]$ packets are transmitted in $w_i[n]T_i$ seconds.

[4] the queue is full and $\tau_i[0]C_i$ packets are on the links

i as

$$\begin{aligned} w_i[k] &= A(1 - q[k - 2w_i[n]]) + Bq[k - 2w_i[n]], \\ \forall k &\in \{1, 2, 3 \dots\}, \\ A &= \Big(w_i[k-1] + I_i(\mathbf{w}_s)(1 - q[k - w_i[n]]) \\ &\qquad - \frac{w_i[k-1]}{2} q[k - w_i[n]]\Big), \qquad (4.9) \\ B &= w_i[k - w_i[n]], \qquad (4.10) \end{aligned}$$

where we have used A and B as auxiliary variables. Similarly, we can write $\texttt{ssthresh}_i = S_i$ as

$$S_i[k] = S_i[k-1](1 - q[k - w_i[n]) + \frac{w_i[k]}{2} q[k - w_i[n]]. \qquad (4.11)$$

Note that equations (4.9)-(4.10) do not track the congestion window *during* fast retransmit, but this is fine as it is reset when fast retransmit ends. To capture the evolution of the bottleneck queue and the transmission rate $x_i[k]$, we introduce the indicator variable $z_i[k]$ that tracks if and when a subflow is in block C. Specifically, we define $z_i[k]$ as

$$\begin{aligned} z_i[k] =& z_i[k-1](1 - q[k - \frac{w_i[n + w_i[n]]}{2} - w_i[n]]) \\ &+ (1 - z_i[k-1])q[k - w_i[n]], \qquad (4.12) \end{aligned}$$

Where we set $z_i[0] = 0$.

Note that both terms in (4.12) remain zero until $q[k - w_i[n]] = 1$ after which $z_i[k] = 1$, and remains constant until $q[k - \frac{w_i[n+w_i[n]]}{2} - w_i[n]] = 1$. Hence, capturing the duration of block C.

Because in mode 2, $I_i(\mathbf{w}_s) << 1$, we have $w_i[n + w_i[n]] \simeq w_i[n] + I_i(\mathbf{w}_s)w_i[n] \simeq w_i[n]$. Assuming that we allow the transmission of partial packets, we can now write

$$x_i[k] = \big(C_i + I_i(\mathbf{w}_s)C_i(1 - q[k - w_i[n]])\big)(1 - z_i[k]), \qquad (4.13)$$

thus completing our model for subflow i in mode 2. In section 4.1.10 we will use the model developed in this section to provide insight into the behaviors observed in Section 4.1.3.

4.1.10 Analyzing our MPTCP Model

We analyze our model in Section 4.1.5 to investigate why MPTCP's initial choice of path can impact its performance. We use $I_i(\mathbf{w}_s)$ as defined by (4.1) to represent MPTCP's joint congestion control. We will show that the coupled nature of $I_i(\mathbf{w}_s)$ is at the root of the differences observed in Section 4.1.3.

4.1.11 A Closer Look at $I_i(\mathbf{w}_s)$

MPTCP's design goals are noted in Section 4.1.2. Satisfying these goals requires that each of MPTCP's subflows "cooperate" in increasing their congestion window so as to, not violate the required fairness property. This introduces a coupling in the congestion control protocol across subflows. An example of such a congestion control protocol is the LIA of (4.1).

Designed to satisfy MPTCP's design goals[5], the coupling imposed by $I_i(\mathbf{w}_s)$ has unexpected consequences that need to be accounted for. One of these unintended consequences is that the coupling in increasing the congestion control windows also couples the effects of the initial choice of path to the performance of MPTCP. We illustrate why in the following.

Using $t = 0$ to mark the time where the second subflow is established, we first look at the case where the path with lower propagation delay i.e., path 1, is used to initiate the connection. In this situation $w_1(0) = w_2(0) + \Delta_w$ for some values of Δ_w. This allows us to arrive at the following proposition:

[5] The work in [78] has shown that this congestion control mechanism does not really satisfy all of the design goals.

Proposition 1. *If path* 1 *is used to initiate the connection, then* (4.1) *for* $I_i(\boldsymbol{w}_s)$ *reduces to*

$$I(\boldsymbol{x}_s) = \frac{x_1}{\tau_1(x_1+x_2)^2} \;\; \forall i \in \{1,2\}, \tag{4.14}$$

where $\boldsymbol{x}_s = (x_1(t), x_2(t))$*. Thus, the following set of equations can be used to characterize the solution to* (4.3)

$$\begin{aligned}
x_2(t) &= K' x_1(t)^{\frac{\tau_1}{\tau_2}} \\
x_1(t) + \frac{K'^2}{(2\frac{\tau_1}{\tau_2}-1)} x_1(t)^{(2\frac{\tau_1}{\tau_2}-1)} + \frac{2K'}{(\frac{\tau_1}{\tau_2})} x_1(t)^{\frac{\tau_1}{\tau_2}} &= \frac{t}{\tau_1^2} + K_2 \\
K' &= \frac{x_2(0)}{x_1(0)^{\frac{\tau_1}{\tau_2}}} \\
K_2 = x_1(0) + \frac{K'^2}{(2\frac{\tau_1}{\tau_2}-1)} x_1(0)^{(2\frac{\tau_1}{\tau_2}-1)} &+ \frac{2K'}{(\frac{\tau_1}{\tau_2})} x_1(0)^{\frac{\tau_1}{\tau_2}}.
\end{aligned} \tag{4.15}$$

It is clear from (4.15) that $x_1(t)$ and $x_2(t)$ have a nonlinear dependence on the initial values $x_1(0)$, $x_2(0)$ when the subflows are operating in mode 1.

Next, we look at the case where the path with higher propagation delay, path 2, is used to initiate the MPTCP connection. We arrive at the following:

Proposition 2. *If path* 2 *is used to initiate the connection then* (4.1) *for* $I_i(\boldsymbol{w}_s)$ *can be written as:*

$$I_1(\boldsymbol{x}_s) = \begin{cases} \frac{x_2}{\tau_2(x_1+x_2)^2} & \textit{if } \tau_1 > \frac{\delta(w_1(0)+\sqrt{w_1(0)^2+\Delta_w w_1(0)})}{\Delta_w} \\ \frac{x_2}{\tau_2(x_1+x_2)^2} & \textit{if } \tau_1 < \frac{\delta(w_1(0)-\sqrt{w_1(0)^2+\Delta_w w_1(0)})}{\Delta_w} \\ \frac{x_1}{\tau_1(x_1+x_2)^2} & \textit{otherwise} \end{cases}$$

$$I_2(\boldsymbol{x}_s) = \begin{cases} \frac{x_2}{\tau_2(x_1+x_2)^2} & \textit{if } \tau_1 > \frac{\delta(w_1(0)+\sqrt{w_1(0)^2+\Delta_w w_1(0)})}{\Delta_w} \\ \frac{x_2}{\tau_2(x_1+x_2)^2} & \textit{if } \tau_1 < \frac{\delta(w_1(0)-\sqrt{w_1(0)^2+\Delta_w w_1(0)})}{\Delta_w} \\ \frac{x_1}{\tau_1(x_1+x_2)^2} & 2\tau_1 > \tau_2 \\ \min(\frac{x_1}{\tau_1(x_1+x_2)^2}, \frac{1}{x_2\tau_2}) & \textit{otherwise} \end{cases}$$

If $I_2(\boldsymbol{x}_s) = I_1(\boldsymbol{x}_s) = \frac{x_2}{\tau_2(x_1+x_2)^2}$*, then we have that*

$$\begin{aligned} x_2(t) &= K' x_1(t)^{\frac{\tau_1}{\tau_2}} \qquad (4.16) \\ x_1(t) + \frac{K'^2}{(2\frac{\tau_1}{\tau_2}-1)} x_1(t)^{(2\frac{\tau_1}{\tau_2}-1)} + \frac{2K'}{(\frac{\tau_1}{\tau_2})} x_1(t)^{\frac{\tau_1}{\tau_2}} &= \frac{t}{\tau_1^2} + K_2 \\ K_2 &= x_1(0) + \frac{K'^2}{(2\frac{\tau_1}{\tau_2}-1)} x_1(0)^{(2\frac{\tau_1}{\tau_2}-1)} + \frac{2K'}{(\frac{\tau_1}{\tau_2})} x_1(0)^{\frac{\tau_1}{\tau_2}} \\ K' &= \frac{x_2(0)}{x_1(0)^{\frac{\tau_1}{\tau_2}}} \end{aligned}$$

Otherwise, if $I_2(\boldsymbol{x}_s) = I_1(\boldsymbol{x}_s) = \frac{x_1}{\tau_1(x_1+x_2)^2}$, $x_1(t)$ *and* $x_2(t)$ *can be defined as in* (4.15).

Proof of Proposition 1-2 is omitted due to space constraints. Propositions 1 and 2 clearly indicate that one should expect a difference in MPTCP transmission rate in mode 1 when the path starting the connection is changed. Further note that this impact is mainly due to the coupling introduced by $I_i(\mathbf{w}_s)$, and therefore, is inherent in the MPTCP protocol design. Propositions 1 and 2 do not quantify this difference, or indicate how long the difference would persist once both MPTCP subflows exit mode 1.

4.1.12 The Overall Impact Of the "First" Path

We have shown that MPTCP's initial choice of path can impact its performance through the coupled nature of its congestion control mechanism. The analysis, focused on the situation where both subflows were in mode 1. Beyond mode 1, in order to observe the impact of MPTCP's initial choice of path on its performance over a relatively long period of time and to see whether or not the effect lasts for the entire duration of its operation, we use a sliding window to characterize MPTCP's transmission rate over a period of 1000 seconds.

Figure 4.5 depicts this quantity for a situation where $C_1 = C_2 = C = 10$ Mbps, $\tau_1 = 200$ ms, $\tau_2 = 280$ ms, and $N_1 = N_2 = 85$ packets[6]. The size of the window

[6] This is done using a numerical solution to our model.

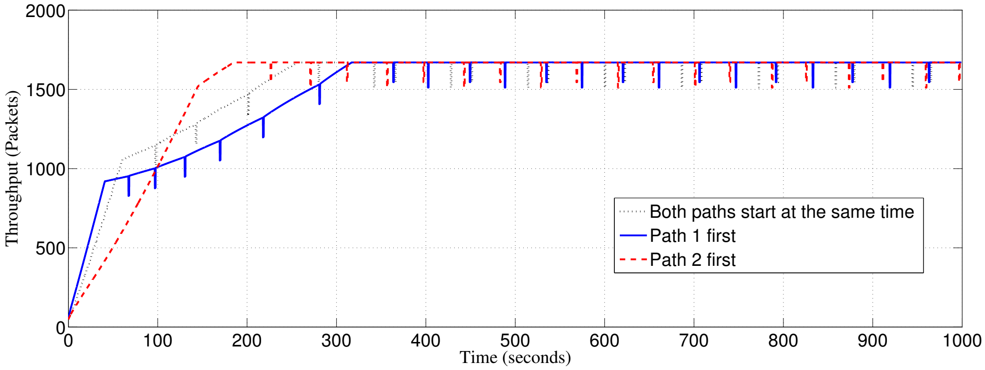


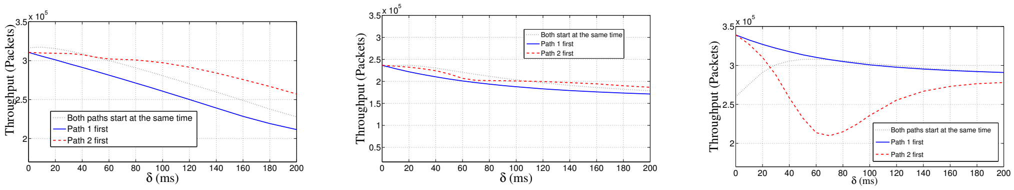
Figure 4.5: Change in transmission rate computed in a sliding window of 1 seconds for different "first" paths. $C_1 = C_2 = 10$ Mbps, $\tau_1 = 200$ms, and $\tau_2 = 280$ms. The path starting second, is established at $t = 3$ seconds.

itself is taken to be 1 seconds, and it is moved by the transmission time of a packet i.e., $\frac{1}{C}$. The queues are chosen to be large so that the congestion window on each subflow can grow before a loss occurs so as to stay in mode 2 even after the loss is detected.

Figure 4.5, depicts three different possibilities for starting the connection, either path 1 is used to initiate the connection (blue/solid line), or path 2 is used to initiate the connection (red/dashed line), or both paths start in the established state at the same time (black/dotted line). The figure shows a difference in MPTCP's transmission rate between the three different scenarios for $t < 300$ secs. Note that in each of these three scenarios there are two dominant points of discontinuity corresponding to where each of the subflows exited mode 1.

Our results in Figure 4.5 indicate that it is possible for the MPTCP subflows to take a relatively long time to reach "steady state". They also show that the optimal strategy in selecting the better path to start with, may vary depending on the amount of data being transmitted. This being said, once both subflows have exited mode 1, MPTCP's transmission rate is relatively stable.[7] Thus, the periodic

[7] Note, that the order in which paths started may indeed influence the frequency with which



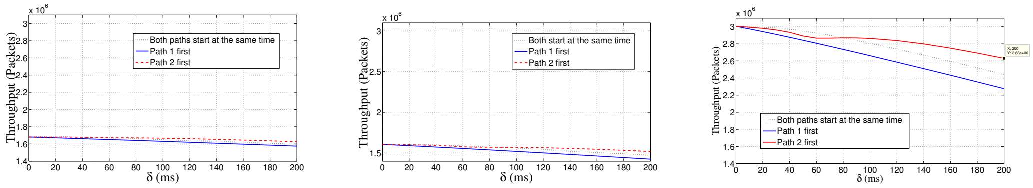
(a) $\tau_1 = 100$ ms, $C_1 = C_2 = 10$ Mbps.

(b) $\tau_1 = 200$ ms , $C_1 = C_2 = 10$ Mbps.

(c) $\tau_1 = 200$ ms, $C_1 = C_2 = 20$ Mbps

Figure 4.6: Total number of packets transmitted vs δ (ms). The total duration of the experiment is 200 seconds for each data point. The figures depict the influence of τ_1, and C on the impact of the initial path.

arrival of ACKs, at least in this example, eliminates the dependency on the initial starting path. One would expect that any parameter influencing transitions into mode 2 e.g., the BDP, competing flows, retransmission timeouts, or AQM queues should also influence the magnitude of the difference in overall throughput across these three scenarios.



(a) $\tau_1 = 100$ ms, $C_1 = C_2 = 10$ Mbps.

(b) $\tau_1 = 200$ ms , $C_1 = C_2 = 10$ Mbps.

(c) $\tau_1 = 200$ ms, $C_1 = C_2 = 20$ Mbps

Figure 4.7: Total number of packets transmitted vs δ (ms). The total duration of the experiment is 1000 seconds for each data point. The figures depict the influence of the duration of the transmission on the effects observed in Figure 4.7

.

Figure 4.6 shows how the BDP influences the impact of the starting subflow on performance. The y-axis ranges over $[170 - 350]$ KPackets ($[255 - 525]$ Mbytes), where each packet is 1460 bytes and each data point shows MPTCP's throughput over a period of 200 seconds. We can see in Figure 4.6b that the maximum differ-

losses are observed on each path. However, for the time being we assume that this effect is relatively minor for the situation considered, where there is no transition from mode 2 to mode 1.

ence in throughput between the three different possible choices of the first paths is 10%. Decreasing the BDP by reducing the round trip times, as in Figure 4.6a, the maximum difference becomes about 20%. Note that the impact of the decrease in RTT is two fold, not only does it effect the BDP, but it also increases $\frac{\partial x_i}{\partial t}$ (equation 4.3). We also see that by increasing the BDP by changing the bottleneck bandwidth, $C = 20$, the influence of the initial choice of path is increased to up to 45% and it is reversed so that using the path of lower RTT provides better performance. Finally, Figure 4.6 indicates, that it is possible that in some situations starting a connection with the "worse" path results in a performance improvement.

Based on the results presented in Figure 4.5, we also expect that the choice of "best" path to initiate the connection should also depend on the duration of the transmission[8]. Figure 4.7 confirms this hypothesis. Figure 4.7 shows MPTCP's throughput over a period of 1000 seconds in the same three scenarios of Figure 4.6 (the range of the y-axis is $[1.4 - 3.1]$ MPackets($[2.1 - 4.6]$ Gbytes)). We can see that the difference in the three paths performance has decreased to 1% in Figure 4.7a, 6% in Figure 4.7b, and 15% in Figure 4.7c. Notice that looking at a longer period of time in the scenario of Figures 4.7c and 4.6c, the use of the higher RTT path results in better performance.

It is also of interest to see how the the difference in the two paths start times (indicated as the time lag in our earlier experiments), Δt, effects the magnitude of the differences observed. In MPTCP's normal operation the Δt is only due to the delay caused by the three way handshake of the second path, which is relatively small. The results shown so far in this section use $\Delta t = 3$ seconds. We conduct further experiments to vary Δt. As expected the change in Δt has a higher influence when path 2 is the starting path (Preposition 2).

Our observations indicate that the choice of which path MPTCP starts with could have a significant impact on performance depending on the size of the file

[8] Or alternatively the size of the file being transmitted.

to be transmitted, the BDP of the paths, and whether or not, and how fast, the subflows are able to reach mode 2. Furthermore, our observations show, that this impact is more significant in better network conditions and that it is possible in some situations that starting with the worse path results in better overall performance.

4.1.13 Discussion

We have shown that the coupling introduced by MPTCP's congestion control mechanisms introduces unexpected behaviors in terms of the influence of its initial choice of path. In our work we flush out the cause of such behavior through a stylized mathematical analysis of the protocol (such a model may also be useful in studying other aspects of the protocol as well). It is expected that the design of a scheduler based on a more comprehensive study of this impact would allow for improved MPTCP performance. It is important to be aware of these impacts when studying the protocol in order to avoid undesirable interactions. This being said, we think it would be beneficial to decouple the two by initiating the multi path communication at the application layer. By doing so, we trade of fine grained control of the traffic distribution for less complexity in design. This also allows us to extend our definition of multi path to go beyond the use of different routes in the network in end to end protocols. In the next chapter we present our design of this system.

Chapter 5

Multipath at The Application Layer

In the previous chapter, we analyzed a simple model of the Multipath transport protocol and showed that there exists complex interactions between the various modules of the protocol. Such interactions can potentially have high impact on client performance. Due to this complexity and the lack of understanding of its possible implications, the protocol is not entirely suitable for performance sensitive applications such as video delivery. We instead propose a multi path approach in the application layer. Such an approach further allows us to expand our notion of multi path to take advantage of the geographic diversity of content delivery networks. Namely, in this context, multi path not only refers to multiple path inside the network but also potentially multiple end point locations. Furthermore, such a design allows us to avoid changing both endpoints. Instead a client side application can achieve the same level of resilience to failures as that of a multipath transport protocol without requiring cooperation from the server-side operators. This further allows for the new protocol to take advantage of caches that might exist in different parts of the network as they need not be updated to serve the new clients. Next, we propose one such solution and outline the various aspects that need to be considered in its design.

5.1 Introduction

The growing importance of video traffic is by now well-documented, with the share of video traffic on North American networks exceeding 70% of peak hour traffic in early 2016 and expected to surpass 80% by the end of 2020 [121]. And while its dominance is not as strong on mobile access links where it now represents about 40% of peak traffic, it is a major factor there as well. Furthermore, this growth appears unimpeded by continued progress in codec efficiency, e.g. x265 or VP9 codecs [82, 130, 104].

In the face of such a trend, or maybe because of it, there are, however, persistent problems when it comes to ensuring quality video delivery. For example, Conviva VXR reports, e.g.[42, 44, 43], which each year track various video performance metrics, report that buffering events (periods during which the video stalls to replenish its playback buffer) remain common (from 20% to 40% depending on location), as do drop in video resolution while playing (those affect over 50% of all videos). Equally important, degradation in video quality is also known to have a major impact on users' behavior and their satisfaction with Internet video [81]. This is obviously of concern, as articulated in several recent industry forums focused on video delivery [30, 31, 32, 33].

The first goal of this paper is, therefore, to explore possible solutions to remedy this situation and improve the quality of Internet video delivery. Of interest in this context is the use of multipath solutions, and in particular solutions that let video clients download video segments (chunks) simultaneously from multiple servers. We refer to such solutions as *Multipath-MultiServer (MuMS)*, e.g.[118, 40]. Reliance on multiple paths from distinct servers can help mitigate exposure to quality degradations caused by congestion or failure of individual paths, and server overload. In particular, multipath has been shown useful in improving throughput and reliability in both wired and wireless networks [10, 35, 37, 58, 63, 115], and there is initial evidence that it could also benefit delay [74] as well as rate stability [12]. The latter is of particular interest, as rate variations are a major factor in video quality degradation.

Specifically, a typical video streaming client operates in two phases [117]: pre-buffering and re-buffering. A video is split into segments (chunks), and in the pre-buffering phase the client requests chunks at the maximum rate allowed by the network[1]. Once there are enough chunks in the client's playback buffer, it starts playing the video and switches to re-buffering mode. In this mode, the client requests chunks at a fixed rate determined by the encoding rate. Video playback proceeds smoothly as long as chunks are in the playback buffer before they need to be played-out. Variations in transmission capacity between the server(s) and the client can result in late delivery or loss of video chunks. This, in turn, depletes the playback buffer and eventually induces video stalls or skips.

A popular approach for dealing with variations in transmission capacity is adaptive bit rates (ABR) [56, 34] – sometimes also called HTTP Adaptive Streaming (HAS). Under ABR/HAS, the client changes its encoding bit rate to match the capacity available in the network and avoid losses/delays. However, rate changes remain visible to the viewer and still translate in degraded quality of experience (QoE) [125, 59, 122], albeit at a lesser level. As a result, it remains desirable to devise solutions that eliminate or mitigate variations in transmission rate, and therefore preserve video quality without having to resort to (coding) rate adjustments. This is one of the goals of our solution, Sunstar, which seeks to leverage multiple paths to different servers to maintain a stable transmission rate even in the presence of network variations (on individual paths). In this respect, Sunstar is complementary to ABR-based solutions.

Another goal of Sunstar is to realize its goal of mitigating rate variations with-

[1] This holds for both video download and live-streaming, though with obvious limitations on the pre-buffering phase for the latter.

out impacting peering costs, i.e. the costs video providers incur from Internet Service Providers (ISPs) for delivering video to their customers. Ensuring that better quality in video delivery does not translate into higher peering costs is of particular importance given the low profit margins under which most video providers operate [19, 24, 46]. Of concern in our context is the extent to which reliance on multiple paths might increase peering costs. The possibility of such increases is, in hindsight, intuitive given the non-linear nature of most ISPs' charging model, i.e. most charge based on the 95^{th} percentile of usage in 5 mins intervals over a period of a month[2]. Spreading transmissions over multiple paths means that a separate 95^{th} percentile is now computed on each path, which, as discussed in Section 5.2, can result in a higher overall cost value.

In summary, Sunstar's main contributions are as follows:

A cost-neutral solution to improving video delivery. We develop a principled understanding of how different mechanisms for improving video delivery, including multipaths, contribute to higher (peering) costs. We use this understanding to develop a scheduler capable of delivering significant performance improvements with little to no impact on cost.

A video client improving users' QoE. We implement a user-space version of a Sunstar video client, and demonstrate its benefits by quantifying its ability to minimize video stalls/skips across a broad range of network impairments. Our Emulab [131] results indicate that Sunstar can improve video performance quality by up to 50% or more in various settings.

5.2 MuMS Benefits and Implications

Before presenting Sunstar, we first motivate a MuMS approach by showing the type of performance improvements achievable when downloading from multiple

[2] See `https://www.noction.com/blog/95th_percentile_explained.`

servers. Next, we offer insight into the relationship that exists between performance and (peering) cost, and in particular why reliance on multiple paths, as in MuMS, can result in higher costs. This serves as a motivation for Sunstar, which seeks a balance between performance and cost.

5.2.1 Performance Benefits

A MuMS solution should improve users' QoE as multiple servers (and the paths from those servers) are unlikely to simultaneously experience congestion or failures. To assess the significance of those gains, we compare the performance of a MuMS client to that of a single-server client in an Emulab experiment. To simplify our setup, in all cases clients use only a single path to each server.

The Emulab connection between the client and each server consists of two links separated by a shaping node with a buffer size of 50 packets. To create an environment that exercises bandwidth limitations, the available bandwidth to each server from the client was set to an average value equal to its download rate T (as determined by the video player), but with variations between $\frac{T}{2}$ and $\frac{3T}{2}$. This was achieved using dummynet [29] on the shaping node. Each client repeatedly downloads a large video (5 minutes or more) and is assigned to a fixed set of n servers where n is either 1, 2, or 3. In all cases, video download proceeds by issuing http get requests at a rate commensurate with that of the video. In the single-server case, TCP controls the actual download rate. In the multi-server case, a standard TCP-like application-level congestion control mechanism determines the available rate from each server, and when multiple servers are available, the lowest RTT server is selected. This represents a relatively basic "scheduler," which nevertheless serves the purpose of demonstrating the benefits of a MuMS solution.

In evaluating performance, we focus on two metrics of importance to video QoE, namely, the fraction of time clients are stalled (because their playback buffer is empty), the average stall duration, and the number of video chunks the client's player skips. Previous studies [18, 69, 50] have verified the correlation between user satisfaction and these metrics. We further verify their impact on user experience through a Mechanical Turk experiment described in Appendix B.1. Note that there is an inherent trade-off between the two metrics. A short time-out for chunks increases skips but minimizes stalls, while increasing the time-out has the opposite result.

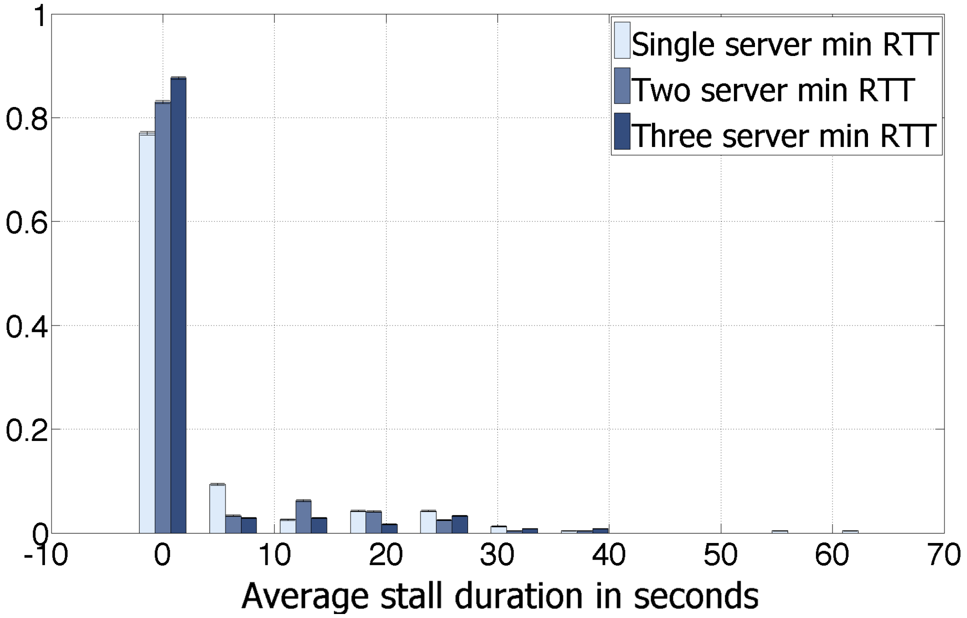


Figure 5.1: Average stall duration across clients.

Figures 5.1-5.2 report the distribution of the average stall time, as well as the fraction of the total download time clients, are stalled across clients for the 1, 2, and 3-server configurations respectively, while Figure 5.3 focuses on the distribution of the number of skipped video chunks for the same configurations. The confidence intervals in the figure show the 95^{th} percentile confidence interval based on assuming a binomial distribution for data in each bin. The figures establish the benefits of a MuMS solution, which is successful in reducing *both* stalls duration

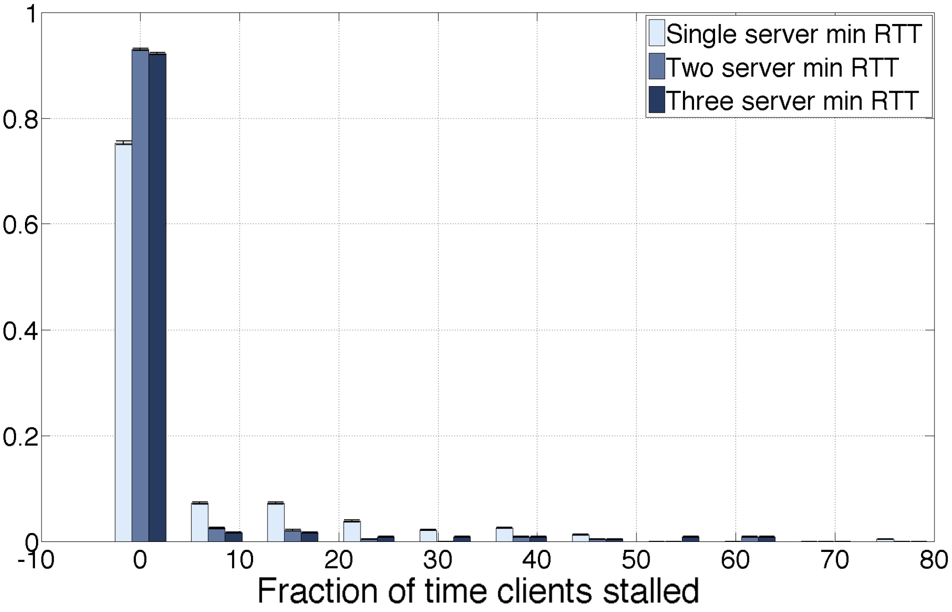


Figure 5.2: Fraction of total download time each client was stalled (across clients).

and the number of skips. For example, more than 91% of 3-server clients saw no stalls, while the number was 90% for 2-server clients, and 73% for single-server clients; and those benefits persist among clients that experienced longer stalls on average. Similarly, over 90% (82%) of 3-server (2-server) clients did not experience any skips, while this number drops to below 80% for single-server clients, with again the benefits of a MuMS solution extending to the tail of the distribution.

In Section 5.4, we show how Sunstar's more sophisticated scheduler can yield further improvements.

5.2.2 Cost-Performance Trade-offs

There are multiple options to improve video delivery. A simple approach is to download the video at the highest possible rate from one or more servers, as it should minimize the odds of a chunk arriving late and/or the playback buffer run-

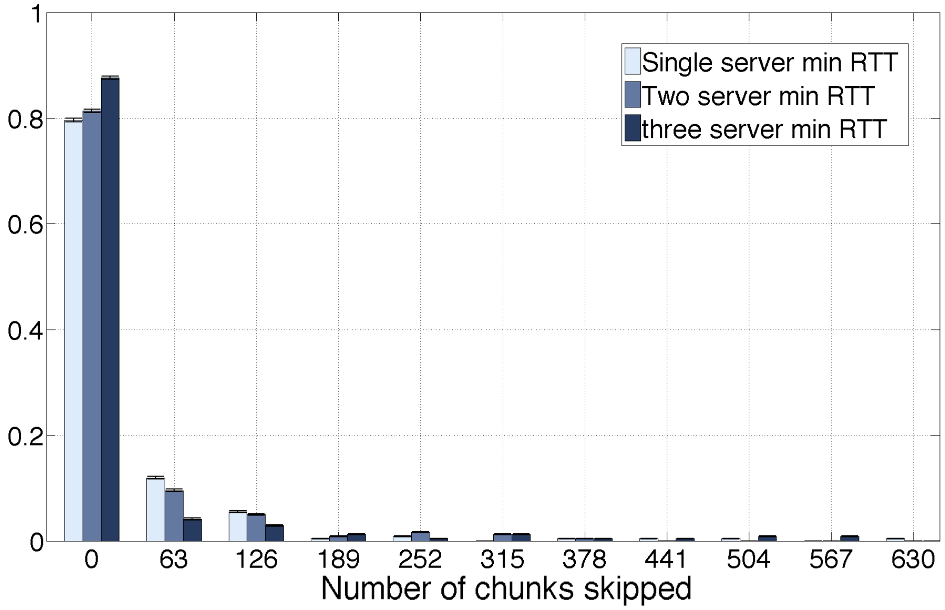


Figure 5.3: Distribution of the number of skipped chunks.

ning empty[3]. This is why works such as [40] focus on maximizing clients' throughput during their "on" period (the time during which the client fills its playback buffer).

However, while performance benefits are intuitive, it is unclear how such a scheme affects *cost*. On one hand, a download strategy that maximizes throughput has clients leaving the system earlier, which can reduce bandwidth usage when computed over 5 minute intervals. On the other hand, the higher download rates while clients are present can increase bandwidth usage. How these two opposing factors contribute to 95^{th}-percentile costs is not obvious at first sight.

To gain a better understanding of this trade-off, we develop a simple analytical model to evaluate the impact of the download rate on peering costs. We consider a scenario where: (1) clients connect to a fixed set of k servers with distinct peering links for each server; (2) clients download a video of size S at a constant aggregate

[3] Note though that this comes at the expense of larger playback buffers at the clients, and a potential waste of bandwidth when clients abandon watching the video halfway.

rate of T and distribute download requests equally across servers (the download rate for each server is T/k); (3) clients arrive according to a Poisson process of rate λ; and (4) peering costs follow a q-percentile model ($q = 95$ in a typical scenario).

Theorem 3 establishes that the higher the download rate T, the higher the peering cost. In other words, while a more aggressive download strategy may improve performance, it results in higher costs.

Theorem 3. *Given clients arriving according to a Poisson process and downloading equally from k servers at an aggregate rate of T, the q-percentile peering cost is an increasing function of T.*

Proof. Assuming a properly provisioned system, i.e. non-blocking, each server plus peering link combination behaves as an $M/G/\infty$ system whose occupancy probability is given by $\pi_i = \frac{e^{-\rho}\rho^i}{i!}$, where $\rho = \frac{\lambda S/k}{T/k} = \frac{\lambda S}{T}$. Assuming ρ is large, i.e. we are dealing with large systems, π_i can be approximated by a normal distribution with mean and variance equal to ρ.

The q-percentile occupancy $n(q)$ of the system (each client is assigned one "server" with a service/download rate of T/k) is then of the form

$$\phi\left(\frac{n(q) - \rho}{\sqrt{\rho}}\right) = q$$

where $\phi(x)$ is the CDF of the normal distribution. This implies $\frac{n(q)-\rho}{\sqrt{\rho}} = \alpha$ where α is a positive constant, e.g. for $q = 0.95$, $\alpha = 1.64$. Thus, $n(q) = \alpha\sqrt{\rho} + \rho$, and the q-percentile traffic volume on the corresponding peering link is $n(q)T/k$. Hence, under a q-percentile cost model, the peering cost for the system is $C_q(\lambda, T, k) \sim (c\sqrt{\rho} + \rho)T/k$, an increasing function of T. □

Although Theorem 3 relies on a number of simplifying assumptions, it nevertheless captures the key factor that while increasing download rates allows clients to leave the system faster (and improves video quality), its overall impact on cost is negative. In other words, downloading at the lowest possible rate that meets the video requirements yields the lowest cost. As described in Section 5.3, we leverage this insight in designing the Sunstar scheduler.

5.2.3 Impact of MuMS on Cost

The previous section established that a greedy/aggressive download strategy had a negative impact on cost. In this section, we show that the multiple paths of a MuMS' solution have a similar effect.

For that purpose and without loss of generality, we consider a system with a single server reachable by clients over either one or two peering links. When two peering links are available, clients split their traffic across the two links according to some strategy. Under the two peering links configuration, we denote as $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ the vectors of traffic volumes recorded in 5 minute intervals on links 1-2, respectively.

In the absence of significant rate variations on either peering link, the traffic volumes on the single peering link configurations are of the form $\boldsymbol{x} = \boldsymbol{x}_1 + \boldsymbol{x}_2$. This readily gives

$$\max(\boldsymbol{x}) \leq \max(\boldsymbol{x}_1) + \max(\boldsymbol{x}_2) \tag{5.1}$$

There are obviously additional factors at play when considering the multiple servers of MuMS clients and a 95^{th} percentile, rather than peak, charging model. Nevertheless, this captures a fundamental aspect of multipath scenarios that points to a negative impact on cost. Mitigating this impact is one of the goals of Sunstar.

5.3 Sunstar Client Design

As discussed in Section 5.2, realizing the lowest possible cost calls for downloading at the lowest possible rate, while meeting the video's target rate of T. Blindly following such a recommendation is, however, likely to result in poor performance. Sunstar seeks to instead minimize rate *variance*, while guaranteeing a sufficient average download rate. Minimizing rate variance still favors lower rates, but at the same time also aims for rate stability. A more stable download rate can, in turn, mitigate playback disruptions by ensuring a regular replenishing of the playback buffer, even during periods of network fluctuations.

5.3.1 Sunstar Client Design Overview

The Sunstar client design assumes that video content is replicated across multiple servers. Upon requesting a video, a client is assigned a given set of servers. This is done through a server selection algorithm imposed by the CDN and can be realized through manifest files that clients can download from a designated server. The client is an application layer client that downloads and plays back videos by sending requests for fixed size chunks to its assigned servers using http range requests. It adapts the number of chunks requested from each server at runtime based on current performance measures. This calls for both estimating the bandwidth available to each server to avoid congesting the network or servers, and for distributing requests across paths to meet cost and performance objectives.

Specifically, the current Sunstar client realizes the following functionality:

Bandwidth estimation. The client updates its estimates of the bandwidth available on each path using a TCP-like AIMD mechanism, which computes a window size (in chunks) for each path. This provides upper bounds for the number of requests that can be sent to each server. Specifically, we use a TCP CUBIC congestion control mechanism to adapt the window size to server i. CUBIC, however, relies on losses to reduce its window size, and since chunks are requested over http, there is no application level loss in the Sunstar client. We use a drop of more than 20% in the estimated average rate to server i as equivalent to a CUBIC packet loss.

Request scheduling. The client dynamically adapts the number of chunks to request from each server. In particular, the Sunstar scheduler (see Section 5.3.2) aims to optimize the user's QoE by allocating chunk requests to minimize rate variability.

Out-of-order delivery. As with other multipath solutions, chunks requested from different servers may arrive out of order. We take an approach similar to MPTCP in dealing with this issue, namely, split the client design into two layers. The lower layer consists of individual http connections or *subflows*, each responsible for sending requests to a single server. Upon receiving responses to requests, the subflow passes them up to the upper layer that reassembles them in the correct order.

Performance optimizations. The Sunstar client incorporates optimizations previously devised for MPTCP. In particular, it employs opportunistic retransmit. When latencies across servers are significantly different, opportunistic retransmit prevents the client from stalling while waiting for a chunk requested from a high latency server. A mechanism similar to TCP timeouts is also implemented for chunk requests. If a response is not received before a timeout, the request is re-sent to another server. Last, we limit the number of retries for chunk requests to bound playback stalls. Once this retry limit is exhausted, the chunk is skipped and the player relies on its codec to mask missing frames.

We acknowledge that an alternative to an application layer solution is a transport layer one, i.e. by extending a protocol such as MPTCP. A transport layer solution has advantages such as finer adaptation granularity and, therefore, faster reactions. However, extending MPTCP to work in a multi-server rather than single-server setting involves non-trivial changes to the network protocol stack. In addition, the tight coupling of MPTCP components can result in unexpected interactions [14]. Avoiding or predicting them calls for a careful evaluation of proposed changes. Hence, an application layer solution offers a number of benefits, in terms of flexibility and ease of deployment.

Note also that while, as mentioned earlier, Sunstar is complementary to ABR solutions, combining Sunstar with an effective ABR strategy, i.e. deciding when to adjust coding rates, is not addressed in the current Sunstar client design. Extending the design to make it fully compatible with ABR solutions is part of future work.

5.3.2 The Sunstar Scheduler

The scheduler is the core component of the Sunstar client[4]. Its main goal is to minimize playback stalls as their frequency and duration are known to have a significant influence on users' QoE [18, 69] (see also our Mechanical Turk experiments in Appendix B.1). Playback stalls are a direct consequence of an empty playback buffer, which arises when the download rate falls below the playback rate for an extended period of time. To minimize the odds of such occurrences, the Sunstar scheduler periodically runs an optimization process that computes how many chunks to request from each server to ensure a target average download rate while minimizing variations around that average rate.

The optimization is greedy and myopic, i.e. based on current performance estimates for the paths to each server and does not attempt to predict future performance. It operates in "epochs" or "rounds", whereby in each epoch it seeks to minimize the *increase* in variance in that round. The optimization takes as input the current window size of each server (an upper bound on its rate), its current estimate of the (average) rate to each server, and the number of pending requests

[4] We show in Sections 5.4 and 5.5 that it not only delivers significantly better performance than other schedulers, but that it realizes those performance improvements with little to no impact on cost.

to each server. It then computes a target rate (number of requests) for each server in the next epoch. This rate is used by the scheduler to schedule requests to the servers.

The rest of this subsection describes the formulation of this optimization process in more details.

Scheduler Optimization

Let $\mathcal{S}$ be the set of servers assigned to a MuMS client. The goal is to guarantee each client an average rate T, while minimizing *changes* in the running variance:

$$(\sum_{i \in \mathcal{S}} \alpha_i R_i - T)^2, \tag{5.2}$$

where R_i is the inverse of the time it takes for requests to arrive from server i (in other words R_i is the inverse of the application level round trip time), and α_i is the number of chunks that the client has requested from server i. The client's attained rate is thus $\sum_i \alpha_i R_i$.

This translates to solving the following optimization:

$$\begin{aligned} \min_{\alpha} & \mid \sum_{i \in \mathcal{S}} \alpha_i R_i - T \mid \\ s.t. \quad & \sum_{i \in \mathcal{S}} \alpha_i \widehat{R}_i \geq T \\ & \alpha_i \leq w_i, \end{aligned} \tag{5.3}$$

where $\widehat{R}_i$ is the expectation of R_i, and w_i the current window size to server i. Let R_i^u and R_i^l be upper and lower bounds for R_i, respectively. R^u and R^l can be estimated using Chebychev's inequality. Equation 5.3 is then equivalent to:

$$
\begin{aligned}
\min_{\alpha,t} \quad & t \\
s.t. \quad & \sum_{i \in \mathcal{S}} \alpha_i \widehat{R}_i \geq T \\
& \sum_{i \in \mathcal{S}} \alpha_i \widehat{R}_i^u \leq T + t \\
& \sum_{i \in \mathcal{S}} \alpha_i \widehat{R}_i^l \leq T - t \\
& \alpha_i \leq w_i.
\end{aligned} \tag{5.4}
$$

Equation 5.4 is derived by first converting the absolute value form of the problem to its linear form and then replacing the two resulting bounded constraints with tighter bounds through R_i^u and R_i^l.

The above formulation assumes integer values for α_i, but this can occasionally result in significant overshoots in the realized rate. We, therefore, use fractional values for α_i. However, because requests are for an integer number of chunks, we maintain a state variable that accounts for the "excess" rate y_i to each server. After solving the optimization, we compute $\max(\alpha_i R_i - y_i, 0)$ and use this value as the target rate to server i. When this corresponds to a fractional number of chunks, we then round it up and update y_i accordingly.

There are two other aspects to the above optimization that need discussion, as they affect the implementation of the Sunstar scheduler.

The first is *Dealing with Infeasibility.* As network bandwidth fluctuates, the above optimization may not always be feasible. However, we still want the client to request the best possible transmission rate. For that purpose, we progressively decrease the target rate T (by 10% in our experiments) and rerun the optimization until a feasible solution is found. Conversely, following a period of rate deficit, we seek to increase the target rate whenever feasible to make up for the deficit.

Specifically, we run the optimization for a higher target rate (typically 30%) until the deficit has been absorbed.

The second issue concerns *Breaking ties.* The optimization need not have a unique solution, e.g. with homogeneous paths with sufficient bandwidth, using any of them is a feasible and optimal solution. We, therefore, add two tie-breaking criteria to the optimization: (1) we minimize the number of servers used, and (2) we favor those that have been used more frequently in the past. Both criteria aim to reduce the number of out-of-order chunks.

$$\max(\boldsymbol{x}) = \max(\boldsymbol{x}_1 + \boldsymbol{x}_2) \leq \max(\boldsymbol{x}_1) + \max(\boldsymbol{x}_2). \tag{5.5}$$

Hence, under a peak rate charging model, a single path solution yields a lower cost.

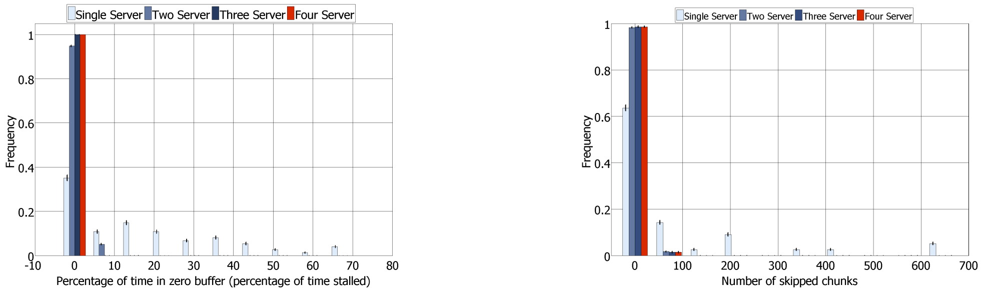
We show in Section 5.4.5 that the optimization of Equation 5.4 is practical in that it can be solved for up to 4 servers in less than 1ms without overloading an entry-level machine[5]. We also note that it is possible to improve the optimization run time by re-using past solutions as a starting point in each round [22]. Such improvements are, however, beyond the scope of this paper.

5.4 Performance Evaluation

We implemented a prototype of the Sunstar client, and carried out a number of experiments to evaluate its behavior along the following dimensions: (1) improvements in video quality over a single server client; (2) improvements in video quality compared to two representative multi-server schedulers, an RTT-based scheduler[6] similar to that used (at the transport layer) by MPTCP, and the YouTuber

[5] Note though, that the complexity of solving the problem grows relatively fast with the number of servers and variables.

[6] We refer to it in the paper as the *min-RTT* scheduler.



(a) Distribution of the fraction of time clients stalled in each run.

(b) Distribution of the number of chunks skipped in each run.

Figure 5.4: Comparison to single path. High and medium bandwidth scenarios were used with smooth bandwidth variations.

scheduler of [40]; (3) its run-time performance, in particular, that of its core optimization routine.

5.4.1 Evaluation Setup

Our evaluation setup is similar to that of Section 5.2, but spans a broader set of scenarios to offer a more a comprehensive evaluation of Sunstar's performance.

Our Sunstar prototype is a Linux-based MuMS client that retrieves video from Nginx web servers. The videos are 250 MB and broken into fixed chunks of 102 KB (we experimented with larger/smaller chunk sizes, and the results were similar). The clients have a target rate of $T = 4.08$ Mbps (experiments at $T = 360$ Kbps yielded similar results). The scheduler's epoch is 10 ms.

As before, the experiments were carried out on the Emulab testbed. Clients connect to servers via a dedicated "path" consisting of a link connecting the client to a machine running dummynet, followed by a link connecting that machine to the server. The dummynet machine on path i is used to add a fixed latency of 20ms and vary the capacity available to the client on path i from 0 to C_i with an average value of $C_i/2$. Those variations seek to capture the impact of interfering traffic on the bandwidth available between clients and servers. We consider two types of bandwidth variations *smooth* and *bursty*. Under smooth variations, the available bandwidth increases and decreases progressively in fixed small sized steps. This is intended to capture scenarios where congestion from cross-traffic changes relatively slowly. In contrast, bursty bandwidth variations are based on large sporadic changes in available bandwidth that seek to mimic abrupt changes in congestion, e.g. because of the start of a high-bandwidth download on a shared link.

In comparing the min-RTT and Sunstar schedulers, we explore both smooth and bursty bandwidth variations. In the evaluation of Sunstar and in its comparison to the min-RTT schedulers, both configurations yield similar results so that we typically report only one. In the comparison to the YouTuber scheduler of [40], we focus on bursty bandwidth variations as rapid changes are expected to be the most stressful to Sunstar's ability to detect and adapt to bandwidth changes. In configurations with smooth bandwidth variations both Sunstar and YouTuber performed similarly.

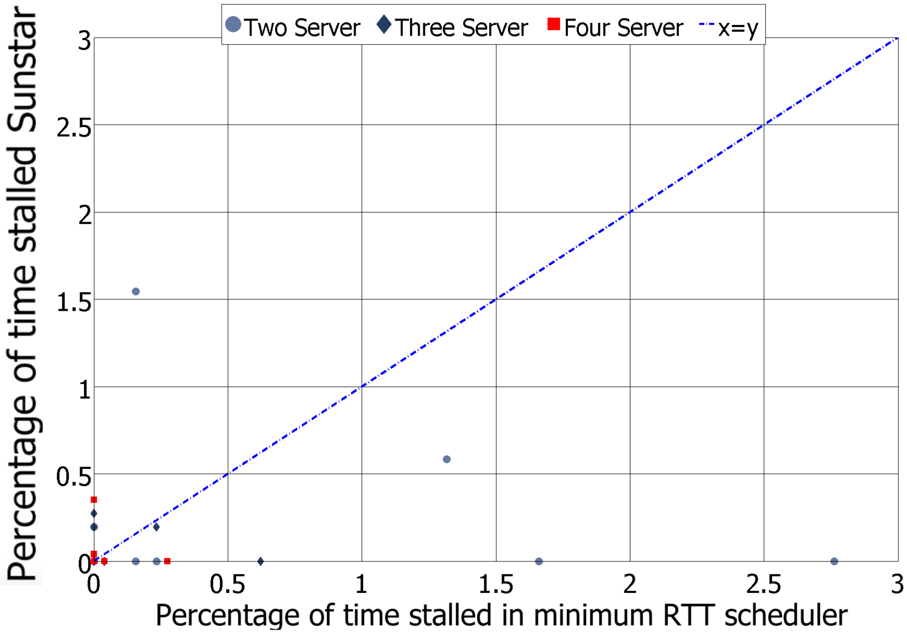


Figure 5.5: High C_i comparison of the two schedulers (smooth bandwidth variations).

We experiment with three configurations: (i) **High** C_i, where $T \ll C_i/2$ (the average available bandwidth significantly exceeds the client's target rate); (ii) **Medium** C_i where $T + 1$ Mbps $\leq C_i/2 \leq T + 2$ Mbps; (iii) **Low** C_i where $C_i/2 < T$. Note that in the **L**ow C_i scenario, multiple paths, and therefore servers, are necessary to meet the client's target rate.

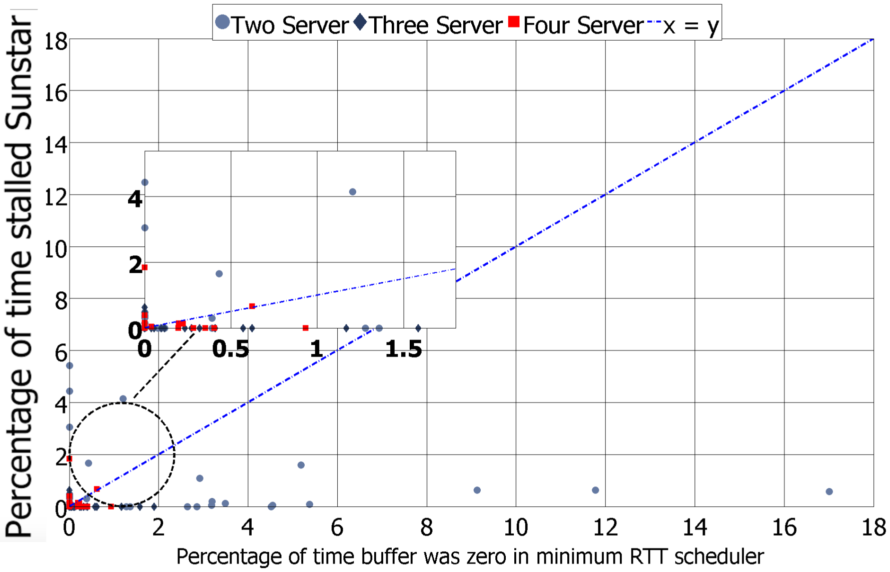


Figure 5.6: Medium C_i comparison of the two schedulers (smooth bandwidth variations).

Performance metrics. As in Section 5.2, we measure QoE-based on [18, 69, 50]: (1) the number of skipped chunks; and (2) the fraction of times clients are stalled.

5.4.2 Comparison to Single Server Clients

Our first set of experiments compares the performance of the Sunstar client to that of a single server client. This repeats the earlier MuMS validation of Section 5.2, with the main distinction that the Sunstar scheduler is now used instead of the min-RTT scheduler. We report results only for the **High** C_i and **Medium** C_i configurations, since they are the only two for which an individual path has enough (average) bandwidth for a client. Statistics for stall durations and number of skipped

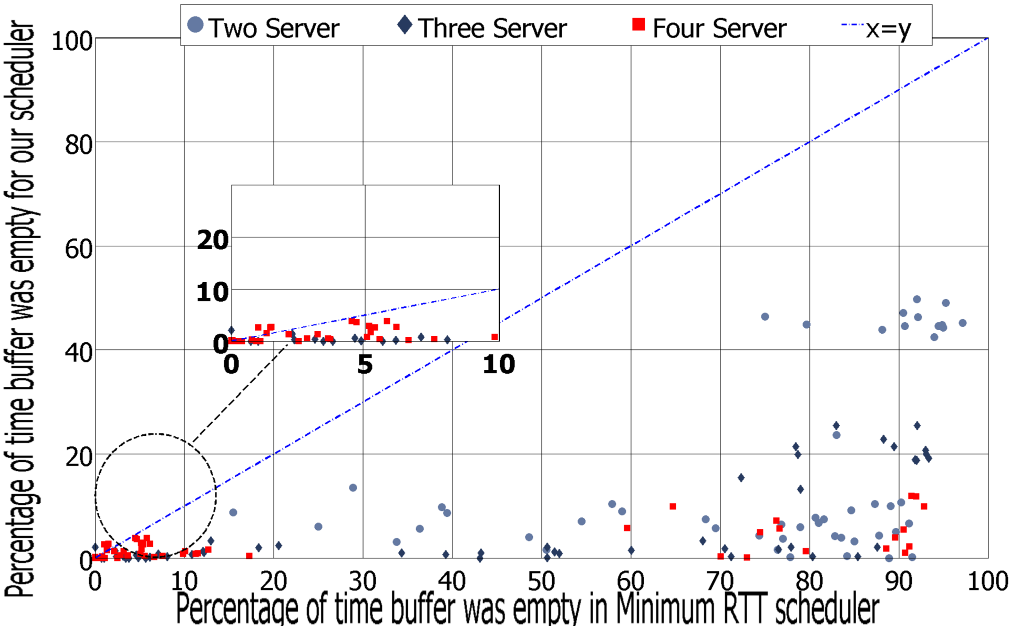


Figure 5.7: Low C_i comparison of the two schedulers (smooth bandwidth variations).

chunks across both configurations are combined and shown in Figures 5.4a,5.4b, respectively[7]. The figures confirm the results of Section 5.2, but now for the Sunstar client. A casual comparison of, say, Figure 5.3 and Figure 5.4b, also hints at the Sunstar scheduler's better performance over that of the min-RTT scheduler of Section 5.2. We explore this aspect further in the next section.

5.4.3 Comparison to the Min-RTT scheduler

We focus on a configuration where a single client downloads videos from the same set of servers (we vary the number of servers from 2 to 4). The client is either the Sunstar client or a client where the Sunstar scheduler is replaced by a min-RTT scheduler. Due to the high variance in the performance of the MinRTT scheduler,

[7] Error bars show the 95th percentile confidence intervals by assuming data in each bar is a Bernoulli distributed random variable.

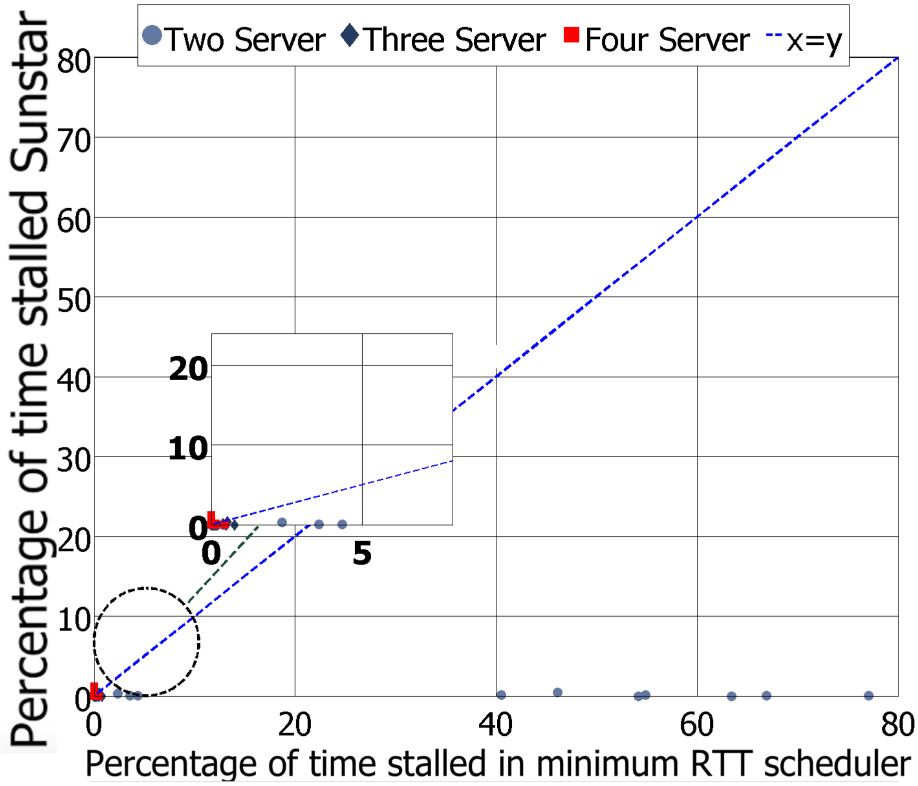


Figure 5.8: Percentage of time stalled in a medium bandwidth/bursty scenario.

we report stall statistics[8] in the form of scatter plots, where each datapoint compares the Sunstar client to the minRTT scheduler (the large confidence intervals for the min-RTT scheduler make the results hard to interpret otherwise). Points below the $x = y$ line indicate better performance for the Sunstar client. Results are shown in Figure 5.5 to Figure 5.7 for the **H**igh, Medium and **L**ow C_i configurations for smooth link variations. Figure 5.8 presents results for one representative configuration with bursty bandwidth variations, namely, the **M**edium configuration.

The figures show that the Sunstar scheduler consistently outperforms the minimum RTT scheduler irrespective of the number of servers used. As expected, the biggest improvements arise in the **Low** C_i scenarios, where the limited resources amplify the need for judicious scheduling decisions. The **Medium** C_i scenario still

[8] Results for skipped chunks statistics were of a similar nature.

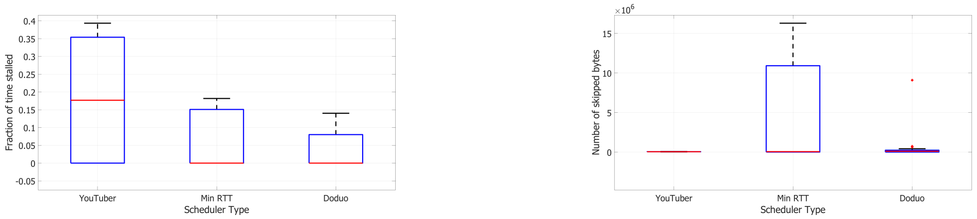
sees the Sunstar client outperforming the other two in most cases, while the differences are less pronounced in the **High** C_i scenario because the plentiful resources ensure that all schedulers perform well. In the rest of the paper (unless explicitly said otherwise), we focus on the **Medium** C_i scenario as it is more likely to arise in practice.

Number of servers	**Percentage of time buffer is empty – minRTT sched.**	**Percentage of time buffer is empty – our sched.**
2	50.75 ± 13.02	1.61 ± 0.15
3	1.25 ± 0.34	0.122 ± 0.098
4	0.114 ± 0.14	0.0145 ± 0.0236

Table 5.1: Percentage of time the playback buffers are empty when multiple clients share the available links/servers simultaneously. We use 95^{th} percentile confidence intervals.

5.4.4 Comparison to The YouTuber Scheduler

The next set of experiments compare the Sunstar client to a client based on **Y**ouTuber [40]. YouTuber works by estimating the bandwidth to each server, and requests chunks at a rate equal to that estimate. The original design of [40] only considers the case of two servers. Hence, to ensure a fair comparison, we limit our experiments to configurations with only two servers.



(a) Fraction of time stalled compared to the YouTuber scheduler.

(b) Number of skipped bytes compared to the YouTuber scheduler.

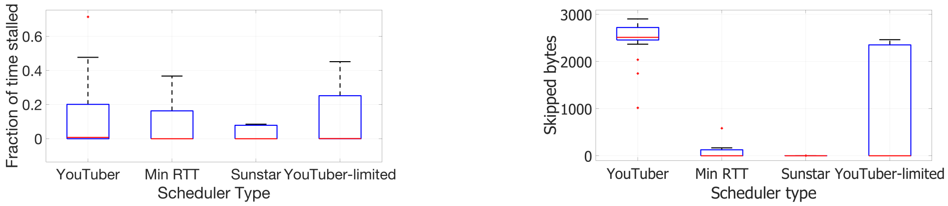
Figure 5.9: Comparing Sunstar scheduler with YouTuber scheduler for regular video downloads

Video Downloads. Our first set of experiments consider the case of stored video downloads (as opposed to live streaming, which we consider next). We also allow the YouTuber an infinite buffer so as to maximize its performance. YouTuber aims to estimate the bandwidth available in the network and send requests accordingly. We, therefore, compare our performance against that of YouTuber in a bursty environment to contrast the two algorithms' ability to adapt to such sudden changes in bandwidth (and therefore their resilience to prediction errors). Since the YouTuber [40] scheduler requires the use of variable sized chunks, we compare the number of skipped bytes instead of the number of skipped chunks. YouTuber also lacks an explicit timeout mechanism. We, therefore, add a timeout period of $10s$ to YouTuber when it stalls. At the end of the timeout period, 16 KB of data (its minimum chunk size) is skipped. This avoids penalizing YouTuber for the lack of timeouts (in which case it would stall for long periods of time).

Figures 5.9a-5.9b summarizes the comparison for the two QoE metrics of stalls and skipped bytes. The data is shown in the form of boxplots to be able to show the results for each metric separately. For completeness, we also include results for the min-RTT scheduler. The box boundaries show the 25th and 75th percentiles of the data while the red lines show the median. The whiskers show the most extreme data points in the data. Our results indicate that when link variations are busty, the Sunstar schedulers outperform the YouTuber scheduler in terms of the fraction of time stalled (both in median and in the tail). The Sunstar scheduler is slightly worse in terms of the number of bytes skipped, but the difference is mostly attributed to the explicit timeout added to YouTuber. When link variations are smooth, Youtuber performs similar to the Sunstar scheduler and we omit the presentation. All in all, Sunstar performs better than Youtuber, and the main reason is due to Sunstar's ability to do finer timescale adaptation accompanied by its use of opportunistic retransmits, while Youtuber instead relies solely on TCP to recover from losses, failures, and sudden bandwidth changes. We note that without the explicit timeout mechanism added to Youtuber, this protocol will end up stalling for long periods – a drawback that Sunstar avoids.

Mis-matched RTTs. We next look at the effect of mismatched RTTs on the different schedulers. Using two servers, we create a mismatch in RTT, by configuring the client to server latency for one to be higher by a delta (10-90ms) compared to the other server. Figure 5.11 illustrates the results. The relative performance of the different schedulers is preserved. The Sunstar scheduler continues to outperforms both the min-RTT and YouTuber scheduler.

Live Streaming. Our next experiment aims to compare the three schedulers (Sunstar, YouTuber, and min-RTT) when streaming live video content. Unlike the earlier workload, live content is generated at a constant rate. We emulate a live streaming experience by first writing the number of bytes equal to the client's pre-buffering threshold into a file. Subsequently, the server continues to write at a constant rate of T to the file until the entire file is created. Clients start sending requests after the pre-buffer portion of the file is created.



(a) Fraction of time stalled.

(b) Number of skipped bytes.

Figure 5.10: Live streaming schedulers comparison - **M**edium C_i, bursty bandwidth variations.

Figures 5.10a-5.10b illustrate the results. Sunstar outperforms all other schedulers in both metrics. YouTuber has the worst performance out of the three (with and without limited buffering) as it cannot accurately estimate the rate at which content is generated at the server. YouTuber's overly aggressive download strategy that maximizes throughput will backfire as this results in unnecessary stalls. The YouTuber design is based on an on/off model where the clients maximize throughput during "on" periods and fill their playback buffers after which they stop and wait until the contents of the buffer drop below a certain threshold at which point they go back into "on" mode. The player optimizes its performance by predicting the bandwidth available to each server and requesting at the maximum rate possible. However, parts of the video may not be available at the time a request is made. A limited buffer size should allow the client to pace itself to download at a more reasonable rate. However, this does not allow YouTuber to take full advantage of scenarios where a significant part of its benefits is coming from: maximizing its download rate. It is, therefore, clear that Sunstar is a better solution compared to YouTuber for handling live video content.

We repeated the above experiments with various sizes of pre-buffers, since YouTuber's performance may be affected by pre-buffer duration. We note that while YouTuber performance improved with the increase in pre-buffer size, the other two schedulers continued to significantly outperform it even with a pre-buffer duration of 1 minute.

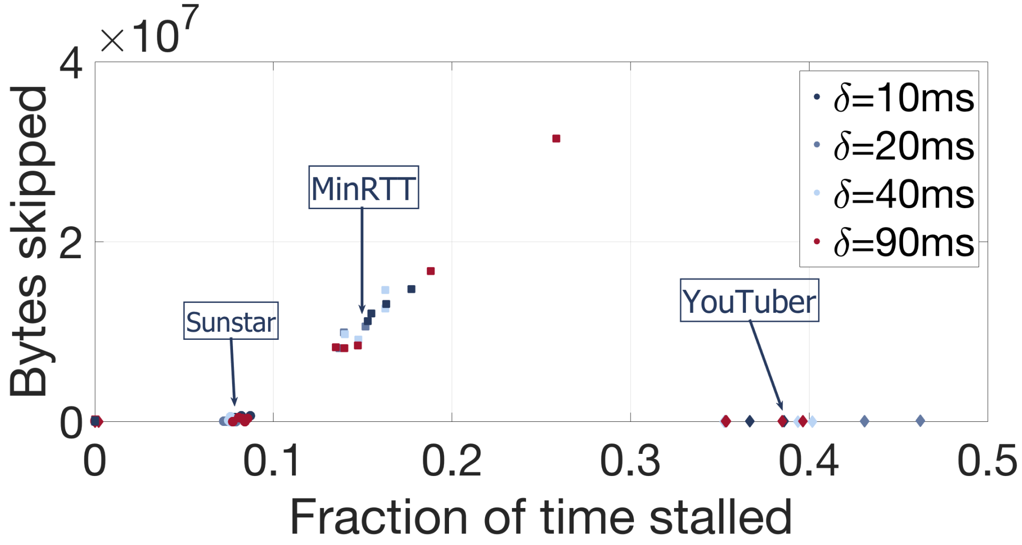


Figure 5.11: The effect of mismatched RTTs. RTT difference between the links is denoted as δ.

5.4.5 Scheduler Execution Time

The previous experiments validate Sunstar's ability to select high-quality paths to servers such that QoE metrics are achieved. Our next set of evaluations measure the runtime of Sunstar's scheduler. Recall that the optimization algorithm used by the scheduler is run every epoch. Hence, it is essential that the scheduler is efficient.

We conducted a number of experiments to evaluate the runtime performance of our scheduling algorithm. We use the Mosek Solver [96] for the optimizer implementation and it is executed every 10 ms. The Emulab machines used for our experiments are Dell PowerEdge 2850s with a single 3GHz processor, 2GB of RAM, and two 10,000 RPM 146GB SCSI disks [2]. Computing the optimal solution typically takes around $1-2$ ms. The run times increase with the number of servers, incurring an average time of 0.27 ± 0.0022 ms (two servers) and 2.20 ± 0.0026 ms (four servers). Beyond four servers, the run time increases significantly – suggesting that 4 servers offers a reasonable trade-off between execution times and performance improvements.

5.5 Cost Impact

Sunstar is motivated by the need to improve video quality, but do so without negatively impacting peering costs. The insight derived from Section 5.2 led to a design that seeks to minimize rate variations while keeping the download rate as close as possible to the target download rate. The previous section showed it was effective in improving video quality. The focus here is on showing that those benefits are realized without increasing the provider's cost. Because the YouTuber scheduler behaves aggressively when it comes to download rate, i.e. it seeks to use all the available bandwidth, we expect that it will perform poorly when it comes to cost.

We, therefore, focus our efforts on comparing the Sunstar client to a single server solution (our baseline), and to a client using the Min-RTT scheduler.

We evaluate peering costs using a setup similar to that of Section 5.4. The main differences are increases in both the number of clients simultaneously active, and the number of servers available to them (we now have 10 servers to chose from). The latter allows us to subsequently consider the impact of server selection on cost, as part of a sensitivity analysis. While this section is only concerned with cost, we also evaluated Sunstar's performance and verified that its benefits remain qualitatively similar to those of Section 5.4 in spite of the larger number of clients.

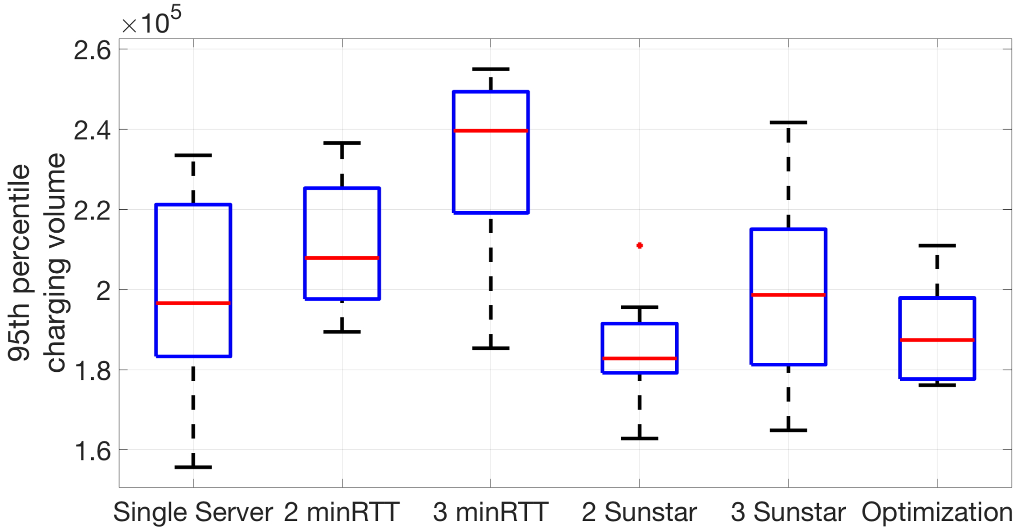


Figure 5.12: Peering costs under different schedulers and server selection approaches.

In a given round of experiment, clients connect to k out of 10 Emulab servers, where k is a parameter that varies from 1 to 3. Since Emulab has a limited number of physical machines available, we configure each machine to have 10 active clients at any point in time, for a total of 100 clients in the system. To emulate an environment with clients coming and leaving, clients watch videos of fixed duration chosen from a set of $\{5, 10, 20, 30, 60\}$ minutes long videos, and then leave to be replaced by a new client that randomly chooses a new video. Video selection is biased towards shorter videos (based on the observation [61] that clients tend to watch shorter duration videos).

Each physical machine has a dedicated link to a dummynet node through which clients originating on that machine experience bandwidth variations that are independent of those for clients on other machines. We use a **M**edium C_i configuration, as described in the previous section, but scale the bandwidth by a factor 10 (the number of clients on the link). Low and high bandwidth scenarios yielded qualitatively similar results. Each server is in turn logically connected to a single peering link shared by all clients accessing it. The bandwidth on the peering link is high enough to avoid congestion, independent of the number of clients assigned to the server. New clients first connect to a "master" server, which redirects them to a list of k servers, $k = 1, 2,$ or 3, from which to download their video. The master uses a round-robin server assignment strategy to select which k servers to assign to a new client.

To allow a meaningful comparison, the same pattern of link bandwidth variations and server assignments are used across schedulers in a given experiment. Individual experiments last for 4 hours, with the 95^{th} percentile cost (traffic volume) of the provider obtained by summing the individual 95^{th} percentiles costs of the 10 servers in those 4 hours. Statistics are then computed over a set of 10 independent experiments. The first five data points (from the left) of Figure5.12 report results for the following configurations: SS (Single Server, our baseline reference), mRTT-2 and mRTT-3 (for Min-RTT with 2 and 3 servers), and SS-2 and SS-3 (for Sunstar with 2 and 3 servers). The definition of the boxplots used in the figure is similar to that of Section 5.4.

We make two observations from the results of Figure 5.12. The first is a confirmation of the insight of Section 5.2.2, which forecast cost increases when using multiple servers. In particular, configurations with 2 and 3 servers and the min-RTT scheduler show clear increases over the single server case (about 20% for 3 servers). The second observation is that the strategy behind Sunstar is indeed successful in both improving performance (as shown in Section 5.4), and realizing those improvements with little to no impact on cost. In fact, our results suggest that Sunstar actually *reduces* cost on average, suggesting that Sunstar is more cost effective in some settings. Overall, our results show that Sunstar does improve video quality, and does so without affecting the peering costs of video providers; something they can ill afford.

In addition to assessing the benefits of Sunstar's "smarter" scheduling decisions, another question of interest is whether further improvements can be realized by also optimizing the set of servers that clients are assigned to. To explore this issue, we formulate the server assignment problem as a constrained optimization[9] (see Appendix B.2 for details on this optimizations) that seeks to greedily minimize increases in the 95^{th} percentile cost when assigning servers to new clients, while meeting client rate constraints.

The results from experiments combining this optimization with the Sunstar scheduler are shown in the right-most part of Figure5.12 (labeled "Optimization"). They highlight that the approach offers only limited benefits, at least when used with the Sunstar scheduler (other experiments with the min-RTT scheduler showed slightly larger improvements). This is likely due in part to the relatively balanced request patterns used in our experiments, as well as the limited ability of a greedy approach to accurately predict how different assignments affect the 95^{th} percentile. In addition, the Sunstar scheduler itself also contributes to this outcome, in that its goal of keeping rates low is likely to already realize much of the achievable gains in cost reductions. We repeated our experiment using a server selection strategy that picks the k closest servers (lowest RTT), and again found little to no differ-

[9] Note that unlike the optimization of the Sunstar scheduler, this optimization is required only once when a new client starts.

ences in cost. A full assessment of the benefits of the server selection approach of Appendix B.2 and other server selection algorithms across traffic scenarios is of interest, and something we expect to explore as future work.

Chapter 6

Related Work

In this work we presented various algorithms that can be employed at the client side of a communication to allow for higher resilience and network diagnosis at low overhead. In this chapter we describe how these algorithms are different from previous work in this space.

6.1 Network diagnosis

Identifying the source of failures in distributed systems and more specifically networks is a mature topic. In this section, we outline some of the key differences of NetPoirot and Vigil with some of these works.

Anomaly detection in distributed systems [55, 70, 57, 73, 141, 47, 79] detect when a failure has occurred using techniques such as PCA [57], Fourier transforms [141], and decision trees [55]. The goal of NetPoirot is to find the entity responsible for the failure. Vigil goes a step further by identifying the device responsible for the failure.

Inference and Trace-Based Algorithms [17, 5, 145, 65, 60, 88, 119, 87, 84, 66] either require (a) data not locally available to the client at runtime, (b) knowledge/inference of the probability distribution of failure on each device in the system, (c) high resource consumption at runtime, or (d) knowledge/inference of application dependence on the different network/service devices. Each of these requirements raises the barrier of adoption, as compared to NetPoirot /Vigil.

Everflow [145] represents a different class of these algorithms that aims at accurately identifying the path of packets of interest. However, Everflow does not scale to be used as an always on diagnosis system. Furthermore, it uses the DSCP bit which is typically reserved for other purposes.

Some inference approaches aim at *covering* the full topology, e.g. [65]. While useful in their own right, they typically only provides a sampled view of connection livelihood (this is also true of most tomography approaches) and do not achieve the type of always on monitoring that NetPoirot and Vigil provide. The time between probes for [65] for example is currently 5 minutes. Therefore, it is highly likely that failures that happen at finer time scales slip through the cracks of its monitoring probes.

Other such work, e.g. [88, 119, 87, 84] require access to both endpoints and/or switches. Such access may not always be possible.

Fault Localization by Consensus [108] violates the data locality requirement, but does not require any information from the network or service. The work assumes that a failure on a node common to the path used by a subset of clients will result in failures on all or a significant number of those clients. Therefore, if many clients in different locations report a failure, it is most likely caused by the service, whereas if only a single client fails the problem is likely local to that client. Figure 6.1 illustrates why this approach fails in the face of problems such as EX. Here, we use the Network Emulator Tool (NEWT) [94] to induce a 5% packet drop rate on all TCP connections to the remote service on a single machine in our stage cluster. Even though the 5% drop rate was present over the entire 6 hour period, only 3 EXs occurred. These events happened when data transmissions increased. This shows that even though a failure may be present in the network, not all clients will observe it at the same time or in the same way. NetProfiler [108] would erroneously, classify such a problem as a client side problem. These approaches require further information in order to provide reliable fault localization. Vigil builds on this idea, it provides a confidence measure that identifies how reliable a diagnosis report is.

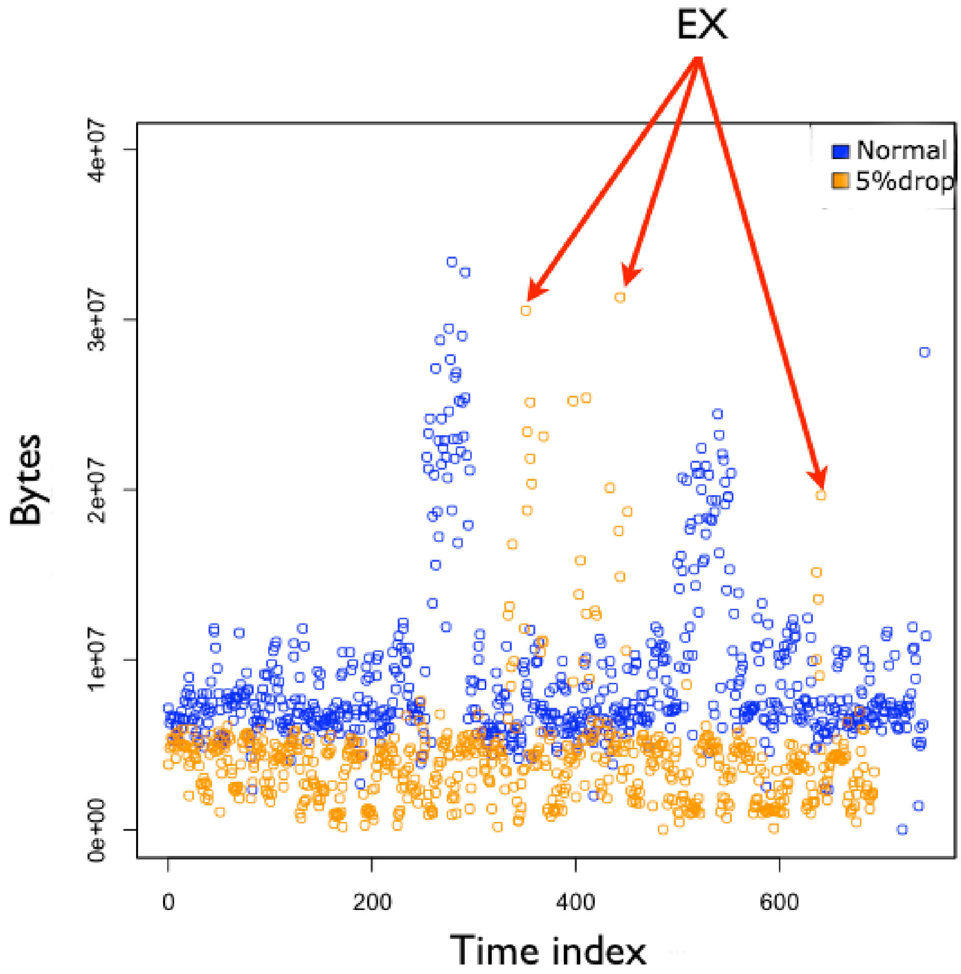


Figure 6.1: Each dot shows BPostedmax in time and is representative of a 30s epoch.

Fault Localization using TCP statistics [90, 134, 140, 138, 119] targets using TCP metrics for diagnosis. [90] requires heavyweight active probing. [134] uses learning techniques (SVM), however, it relies on packet captures from both end points and is limited in the scope of failures it detects (network and application problems only). T-Rat [140] infers TCP information from packet captures. It is used to understand why a TCP connection is rate limited. Our diagnosis goals are broader. T-Rat is too heavyweight to be used as always on. SNAP [138] requires sensitive information, such as topology, path information, and switch counters to perform diagnosis. NetPoirot eliminates the need to share such information with clients. SNAP [138] initially identifies performance problems/causes for single connections by acquiring TCP information which are gathered by querying socket options. It also gathers routing data combined with topology data to compare the TCP statistics for connections that share the same host, link or top of rank switch or aggregator switch. It then collects the mapping of connections to applications to further see if a particular application is experiencing more problems than others. Given their lack of continuous monitoring, all of these approaches fail in detecting the type of problems NetPoirot/Vigil are designed to detect. Furthermore, the goal of Vigil is more ambitious, namely to find the link that causes packet drops for each TCP connection.

Network tomography [143, 71, 85, 51, 80, 76, 103, 38, 144, 72, 52, 99] typically consist of two aspects: (i) the gathering and filtering of network traffic data to be used for identifying the points of failure [103, 143] and (ii) using the information found in the previous step to identify where/why failures occurred [71, 85, 51, 99, 48, 52, 20]. Vigil and NetPoirot utilize ongoing traffic to detect problems, unlike tomography-based approaches which require a much heavier-weight operation of gathering large volumes of data for analysis. Tomography-based approaches are also better suited for non-transient failures, while our systems can handle both transient and persistent errors. NetPoirot/Vigil also have coverage that extends to the entire network infrastructure, and does not limit coverage to only paths between designated monitors as some tomography approaches do. Some tomography techniques on analyzing failures [143, 51, 103, 99] are complementary and can be applied to NetPoirot/Vigil to improve our accuracy.

Learning Based Approaches [36, 49, 6] do failure detection. NetPoirot also uses machine learning techniques, but the application domain is different (home networks [6] and mobile video delivery [49]). [36] uses decision trees in order to locate the device responsible for a failure by observing the path traversed on the tree. It requires request to server mappings. NetPoirot does not require this additional information.

Application diagnosis [39, 7] aim at identifying the cause of problems in a distributed application's execution path. These works are useful in their own right, but the limitations of diagnosing network level paths and the complexities associated with this task are different. Obtaining all execution paths *seen* by an application, is plausible in such systems but is not an option in ours.

Fault localization in wide area networks [139, 102, 128, 127, 89, 21] shares some similarity to that of Vigil, the constraints in a datacenter network are vastly different.

Understanding datacenter failures [146, 62] aims to identify the various types of failures in data center networks. These approaches are useful in understanding the types of problems that arise in practice and to ensure that our monitoring/diagnosis engines are well equipped to find them. In fact, in our analysis of Vigil, we use the findings of [146].

Perhaps the most related body of work to Vigil is that of [119] which requires modifications to routers and assumes a specific topology, limitations that Vigil does not have. Moreover, in order to apply their approach to a traditional datacenter, a number of engineering problems need to be overcome as well, including finding a substitute for their use of the DSCP (used for other purposes in our data center).

6.2 Multi Server Video Delivery

There has obviously been much work on optimizing video transmission and using multipaths to overcome network impairments. Our intent is not to provide an exhaustive review, but rather to summarize major approaches and highlight similarities and differences with this work.

6.2.1 Video Delivery Optimizations

Adaptive bit rate (ABR) [56, 34] is, as mentioned earlier, a powerful approach for mitigating the impact of network rate variations by allowing clients to correspondingly adjust their video coding rate. The main drawback of ABR is that it requires servers and caches to store multiple encodings of the same video, or codecs to be able to dynamically update their coding rate. In addition, adjustments in coding rates still produce noticeable changes in video quality [125, 59, 122]. Our goal with Sunstar is complementary to ABR, in that we aim to leverage multiple paths to different servers to minimize (network) rate variations, and therefore coding rate changes.

Caching is another popular strategy. It improves performance by moving files as close as possible to clients through caches located at the network edge. This is, however, not always effective, in part because copyrights laws make much content un-cachable, and the combination of the long tail of video popularity [41] and the use of ABR can lower cache efficiency. Consequently, even smart caching algorithms only boast a cache efficiency of about 50% [114] (for ABR videos). Sunstar is meant to improve video delivery in instances when video cannot be served from a local edge cache.

OpenConnect [101] was proposed by Netflix. It relies on embedding appliances in ISPs′ networks to locate content closer to clients and to preemptively populate caches at off-peak hours to avoid cache warm-up and network congestion during peak hours. It calls for partnership between content providers and ISPs, e.g. locating appliances in the ISP's facility, which some large ISPs are reluctant to engage in as they have competitive businesses [120].

Content filtering limits content available to users to content that can be delivered with high quality, e.g. from caches. This is realized by applying filters that limit viewable listings to a subset of (popular) videos, or by steering users away from unpopular items [25]. Both approaches result in potential loss of revenue, e.g. removing such filters can increase the number of views by 45% [25].

Dynamic CDN switching is offered by companies such as Conviva and Cedexis which act as brokers in the CDN domain. They measure CDN status and switch between CDN providers based on performance. The main disadvantage is that CDNs no longer control how clients are redirected to servers, which can have unintended consequences, including higher costs [97]. In contrast, Sunstar keeps the assignment of clients to servers under the video provider′s purview and incorporates mitigating cost increase as an explicit criterion.

Hybrid CDN-P2P seeks to combine the best of CDN and peer-to-peer solutions [137]. Netsession [142] offers a representative example of the potential benefits of such an approach. Unlike Sunstar, it again does not offer an explicit control on how improving performance affects a provider′s cost. Additionally, aspects such as copyright management are traditionally difficult to handle in a P2P setting.

6.2.2 Multipath Solutions

The benefits of multipaths have been studied in numerous settings, e.g.[9, 124, 45], but perhaps most visible among them are studies of Multipath TCP (MPTCP) [116], whose investigations related to congestion control [123, 135, 112, 14] or scheduling [13, 106] are of most relevance, even if not directly applicable because of MPTCP's assumption of a single source and a single destination. Nevertheless, several techniques developed to improve MPTCP can be repurposed in a MuMS setting. For example, as discussed in Section 5.3, Sunstar is able to leverage MPTCP's opportunistic retransmit to improve its performance.

More directly comparable to Sunstar are works that explicitly target improving video delivery by relying on multiple servers. In most such settings, e.g.[118, 40], the focus has, however, been on optimizing download *rates*, which, as we shall see, can have a significant impact on cost. Specifically and as discussed in Section 5.2, while the more aggressive download strategies of [118, 40] can reduce the odds of skips and stalls, they typically result in higher costs. In contrast, Sunstar aims to realize comparable improvements in video quality, but with little to no increases in cost.

6.2.3 Server selection and cost optimizations

Another relevant body of work is that of server selection algorithms that optimize for a given metric, e.g. performance or cost [98, 133, 100, 23]. Extending those approaches to a MuMS' setting is, however, challenging. This is because the multipath nature of MuMS clients makes predicting variations in traffic volumes at peering links more difficult than with single path clients. In particular, clients are now free to choose how to distribute video requests across paths. How this impacts cost and performance adds a new non-trivial dimension to the problem.

In this context, the approach closest to Sunstar is [86]. It considers both performance and cost and adopts a cost minimization formulation with performance as a constraint, where for each CDN performance is based on long-term QoE measurements from clients in different regions. Given an expected request load, [86] computes a "prioritized" list of servers that a client should use when requesting content. Higher priority servers are to be used first as long as they have available capacity. A TCP-like AIMD mechanism is used to estimate the bandwidth available to each server. Sunstar's approach differs from that of [86] in that rather than minimizing cost and keeping performance as a constraint, it leverages its understanding of the relationship between cost and performance to select rate variation as its minimization target. In addition, Sunstar's scheduler offers a more responsive mechanism than that of [86], which is limited to the set of servers computed by its optimization. In some sense, the scheduler of [86] is similar to the min-RTT scheduler of Section 5.4, which, as we shall see, performs significantly worse that that of Sunstar in terms of both cost and performance.

Finally, of note in the context of cost optimization is [126], which attempts to account for the contributions of individual users to a 95^{th} percentile cost function. Although such an approach could be used to formulate an appropriate objective function for a server selection algorithm, it requires detailed knowledge of the exact traffic patterns of each user. This is unlikely to be feasible, especially in a multipath setting where variations on a given path affect traffic on all paths.

Chapter 7

Conclusion

Endpoints can play a significant role in improving their own quality of service when using networks such as the Internet or datacenters. In this work we demonstrate how clients can effectively assist network operators in finding the source of performance problems. We introduce two systems NetPoirot and Vigil that allow the client to help with diagnosis (and recovery) when failures occur.

Diagnosis and recovery are only one part of ensuring all clients achieve high QoS. Both take time to resolve performance problems if/when they occur. We show that through the use of multiple path (or multiple servers in the context of video delivery) clients can significantly improve their performance as all such path are unlikely to experience problems at the same time. We demonstrate the effectiveness of this idea in the context of video delivery through a new multipath-multiserver client (Sunstar).

There are a number of questions that remain unanswered in this work. For example, in designing NetPoirot we found that TCP reacts differently to different types of failures depending on their root cause. By deriving a better understanding of when, how, and why such reactions are triggered it should be possible to arrive at not only a more robust protocol but also to provide further assistance to operators when diagnosing failures in data center networks. Furthermore, the use of multipath changes the diagnosis problem significantly. How does one identify the subpath responsible for a failure? What is the implication for congestion control and scheduling across the different path? How fast can these algorithms recover from a transient failure and reroute traffic accordingly? What are the potential implications of these protocols for network security? These are problems that remain unanswered and should be investigated if these protocols are to be widely adopted in future.

Appendix A

Vigil Proofs

Definition: [Clos topology]

A Clos topology has n_{pod} pods each with n_0 top of the rack (ToR) switches under which lie H hosts. The ToR switches are connected to n_1 tier-1 switches by a complete network ($n_0 n_1$ links). Links between tier-0 and tier-1 switches are referred to as *level 1 links*. The tier-1 switches within each pod are connected to n_2 tier-2 switches by another complete network ($n_1 n_2$ links). Links between these switches are called *level 2 links*.

Remark 4 (Communication and failure model)**.** *Assume that connection occur uniformly at random between hosts under different ToR switches. Since the number of hosts under each ToR switch is the same, this is equivalent to saying that connections occur uniformly at random directly between ToR switches. Also, assume that link failure and connection routing are independent and that links drop packets independently across links and across packets.*

Remark 5 (Notation)**.** *We use calligraphic letter ($\mathcal{A}$) to denote sets. Also, we write $[M]$ to mean the set of integers between* 1 *and M, i.e., $[M] = 1, \dots, M$.*

Theorem 6. *In a data center with Clos topology, the rate of traceroutes T going through* *any link is bounded by*

$$T \leq \frac{CH}{n_1 n_2} \cdot \max\left[n_2, \frac{n_0^2(n_{\text{pod}} - 1)}{(n_0 n_{\text{pod}} - 1)}\right], \tag{A.1}$$

where C is the connection rate between hosts. Hence, a maximum traceroute rate of T_{max} will be met if the connection rate of each host satisfies

$$C \leq \frac{n_1 n_2 T_{max}}{H \cdot \max\left[n_2, \frac{n_0^2(n_{\text{pod}}-1)}{(n_0 n_{\text{pod}}-1)}\right]}. \tag{A.2}$$

Proof. Start by noticing that the number of hosts below each ToR switch is the same, so that we can consider connections to happen uniformly at random directly between ToR switches at a rate CH. Moreover, note that routing probabilities are the same for links on the same level, so that the traceroute rate depends only on whether the link is on level 1 or level 2.

Since the probability of a switch routing a connection through any link is uniform, the traceroute rate of a level 1 link is given by

$$R_1(r) = \frac{1}{n_1} CHr, \tag{A.3}$$

where r denotes the probability of issuing a traceroute, i.e., the probability of a TCP retransmission occurring during a connection. Similarly for a level 2 link:

$$R_2(r) = \frac{n_0}{n_1 n_2} \frac{n_0(n_{\text{pod}} - 1)}{(n_0 n_{\text{pod}} - 1)} CHr, \tag{A.4}$$

where the second fraction represents the probability of a host connecting to another host outside its own pod, i.e., of going through a level 2 link. Since (A.3) and (A.4) are increasing in r and $r \leq 1$, it holds that $T \leq \max\left[R_1(1), R_2(1)\right]$. Taking $\max\left[R_1(1), R_2(1)\right] \leq T_{\max}$ yields (A.2).

$\square$

Theorem 7. *In a Clos topology with $n_0 \geq n_2$ and $n_{\text{pod}} \geq 1 + \max\left[\frac{n_0}{n_1}, \frac{n_2(n_0-1)}{n_0(n_0-n_2)}, 1\right]$, Vigil will rank with probability $(1-\epsilon)$ the $k < \frac{n_2(n_0 n_{\text{pod}}-1)}{n_0(n_{\text{pod}}-1)}$ bad links that drop packets* *with probability p_b above all good links that drop packets with probability p_g as long as*

$$p_g \leq \frac{1-(1-p_b)^{c_l}}{\alpha c_u}, \tag{A.5}$$

where c_l and c_u are lower and upper bounds, respectively, on the number of packets per connection,

$$\alpha = \frac{n_0(4n_0-k)(n_{\text{pod}}-1)}{n_2(n_0 n_{\text{pod}}-1)-n_0(n_{\text{pod}}-1)k}, \tag{A.6}$$

and

$$\begin{aligned}\epsilon &\leq e^{-N\,\mathrm{D}_{\mathrm{KL}}((1+\delta)v_g\|v_g)} + e^{-N\,\mathrm{D}_{\mathrm{KL}}((1-\delta)v_b\|v_b)} \\ &= 2e^{-\mathcal{O}(N)},\end{aligned} \tag{A.7}$$

with v_g and v_b being the probabilities of a good and bad link receiving a bad vote, respectively, N being the total number of connections between hosts, and $\mathrm{D}_{\mathrm{KL}}(q\|r)$ denoting the Kullback-Leibler divergence between two Bernoulli distributions with probabilities of success q and r.

Before proceeding, note that when, as in our data center, $n_0 \geq 2n_2$, $\frac{n_2(n_0-1)}{n_0(n_0-n_2)} \leq 1$, so that case the condition on the number of pods reduces to $n_{\text{pod}} \geq 1 + \frac{n_0}{n_1}$.

Proof. The proof proceeds as follows. First, note that the number of votes on a bad (good) link is a binomial random variable with parameters N and v_b (v_g). Then, using large deviation theory [11], we proceed to show that if $v_b \geq v_g$, Vigil will rank bad links above good links with high probability for large enough N (Lemma 8). Finally, we derive bounds on v_b and v_g using Boole's inequality (Lemma 9) and use these results to show that $v_b \geq v_g$ if p_b and p_g satisfy (A.5).

Let us start by stating the lemmas and showing how they imply Theorem 7.

Lemma 8. *If $v_b \geq v_g$, Vigil will rank bad links above good links with probability $(1-\epsilon)$ for ϵ as in* (A.7).

Lemma 9. *In a Clos topology with* $n_0 \geq n_2$ *and* $n_{\text{pod}} \geq \max\left[\frac{n_0}{n_1}, \frac{n_2(n_0-1)}{n_0(n_0-n_2)}, 1\right] + 1$, *it holds that for* $k \leq n_0$ *bad links*

$$v_b \geq \frac{r_b}{n_0 n_1 n_{\text{pod}}} \tag{A.8a}$$

$$v_g \leq \frac{1}{n_1 n_2 n_{\text{pod}}} \frac{n_0(n_{\text{pod}}-1)}{n_0 n_{\text{pod}}-1} \left[(4 - \frac{k}{n_0})r_g + \frac{k}{n_0} r_b\right] \tag{A.8b}$$

where r_b *and* r_g *are the probabilities of a retransmission occurring due to a bad and a good link, respectively.*

From the (A.8) in Lemma 9, it holds that

$$r_b \geq \underbrace{\frac{n_0(4n_0-k)(n_{\text{pod}}-1)}{n_2(n_0 n_{\text{pod}}-1) - n_0(n_{\text{pod}}-1)k}}_{\alpha} r_g \Rightarrow v_b \geq v_g, \tag{A.9}$$

for $k < \frac{n_2(n_0 n_{\text{pod}}-1)}{n_0(n_{\text{pod}}-1)} < n_0$. Thus, in a Clos topology, $r_b \geq \alpha r_g \Rightarrow v_b \geq v_g$ for α as in (A.6).

However, (A.9) gives a relation between the probabilities of retransmission (r_g, r_b) instead of the packet drop rates (p_g, p_b) as in (A.5). Indeed, to obtain (A.5), note that the probability r of retransmission during a connection with c packets due to a link that drops packets with probability p is $r = 1 - (1-p)^c$. Since r is monotonically increasing in c, we have that $r_b \geq 1 - (1-p_b)^{c_l}$. Similarly, $r_g \leq 1 - (1-p_g)^{c_u}$. Using the fact $(1-x)^n \geq 1 - nx$ yields (A.5).

We now proceed with the proofs of Lemmas 8 and 9.

Proof of Lemma 8. We start by noting that in a data center-sized Clos network, almost every connection has a hop count of 5. In our data center, this happens to 97.5% of connections. Therefore, we can approximate links votes by assuming all bad votes have the same value. Thus, suffices to determine how many bad votes each link has.

Since links cause retransmissions independently across connections (see Remark 4), the number of bad votes received by a bad link is a binomial random variable B with parameters N, the total number of connections, and v_b, the probability of a bad link receiving a bad vote. Similarly, let G be the number of bad votes on a good link, a binomial random variable with parameters N and v_g. Vigil will correctly rank the bad links if $B \geq G$, i.e., when bad links receive more votes than good links. This event contains the event $\{G \leq (1+\delta)Nv_g \cap B \geq (1-\delta)Nv_b\}$ for $\delta \leq \frac{v_b - v_g}{v_b + v_g}$. Using the union bound $\mathbb{P}\left[\bigcup_i \boldsymbol{E}_i\right] \leq \sum_i \mathbb{P}\left[\boldsymbol{E}_i\right]$ [53], the probability of Vigil identifying the correct links is therefore bounded by

$$\begin{aligned}\mathbb{P}(B \geq G) &\geq \mathbb{P}\left[G \leq (1+\delta)Nv_g \cap B \geq (1-\delta)Nv_b\right] \\ &\geq 1 - \mathbb{P}\left[G \geq (1+\delta)Nv_g\right] \\ &\quad - \mathbb{P}\left[B \leq (1-\delta)Nv_b\right]\end{aligned} \tag{A.10}$$

To proceed, note that the probabilities in (A.10) can be bounded using the large deviation principle [11]. Indeed, let S be a binomial random variable with parameters M and q. For $\delta > 0$ it holds that

$$\mathbb{P}\left[S \geq (1+\delta)qM\right] \leq e^{-M\,\mathrm{D_{KL}}((1+\delta)q\|q)} \tag{A.11a}$$

$$\mathbb{P}\left[S \leq (1-\delta)qM\right] \leq e^{-M\,\mathrm{D_{KL}}((1-\delta)q\|q)} \tag{A.11b}$$

where $\mathrm{D_{KL}}(q\|r)$ is the Kullback-Leibler divergence between two Bernoulli distributions with probabilities of success q and r [**?**]. Explicitly,

$$\mathrm{D_{KL}}(q\|r) = q\log\left(\frac{q}{r}\right) + (1-q)\log\left(\frac{1-q}{1-r}\right).$$

Substituting the inequalities (A.11) into (A.10) yields (A.7).

□

Proof of Lemma 9. Before proceeding, let $\mathcal{T}_0$, $\mathcal{T}_1$, and $\mathcal{T}_2$ denote the set of ToR, tier-1, and tier-2 switches respectively (Figure A.1). Also let $\mathcal{T}_0^s$ and $\mathcal{T}_1^s$, $s = [n_{\text{pod}}]$, denote the tier-0 and tier-1 switches in pod s respectively. Note that $\mathcal{T}_0 = \mathcal{T}_0^1 \cup \cdots \cup \mathcal{T}_0^{n_{\text{pod}}}$ and $\mathcal{T}_1 = \mathcal{T}_1^1 \cup \cdots \cup \mathcal{T}_1^{n_{\text{pod}}}$. Note that we use subscripts to denote the switch

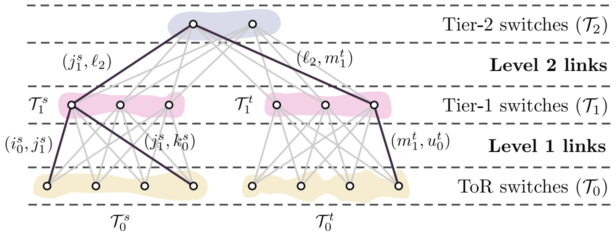


Figure A.1: Illustration of notation for Clos topology used in the proof of Lemma 9

tier and superscripts to denote its pod. To clarify the derivations, we maintain this notation for indices. For instance, i_0^s is the i-th tier-0 switch from pod s, i.e., $i_0^s \in \mathcal{T}_0^s$, and ℓ_2 is the ℓ-th tier-2 switch. Note that tier-2 switches do not belong to specific pods. We write (i_0^s, j_1^s) to denote the level 1 link that connects i_0^s to j_1^s (as in Figure A.1) and use $r(i_0^s, j_1^s) = r(j_1^s, i_0^s)$ to refer to the probability of link (i_0^s, j_1^s) causing a retransmission. Note that r is also a function of the number of packets in a connection, but we omit this dependence for clarity.

The bounds in (A.8) are obtained by decomposing the events that Vigil votes "bad" for a level 1 or level 2 link into a union of simpler events. Before proceeding, note that each connection only goes through one link in each level and in each direction, so that events such as "going through a ToR to tier-1 link" are disjoint.

Starting with level 1, let $\boldsymbol{A_0}$ be the event that a connection goes through link (i_0^s, j_1^s). This event happens with probability

$$\mathbb{P}\left[\boldsymbol{A_0}\right] = \frac{1}{n_0 n_1 n_{\text{pod}}}, \tag{A.12a}$$

given that there are $n_0 n_1 n_{\text{pod}}$ level 1 links and that connections occur uniformly at random. Therefore, a link (i_0^s, j_1^s) receives a bad vote if a connection goes through it (event $\boldsymbol{A_0}$) *and* either of the following occurs:

- event $\boldsymbol{A_1}$: (i_0^s, j_1^s) causes a retransmission, i.e.,

$$\mathbb{P}\left[\boldsymbol{A_1}\right] = r(i_0^s, j_1^s) \tag{A.12b}$$

- event $\boldsymbol{A_2}$: the connection also goes through some (j_1^s, k_0^s), $k_0^s \neq i_0^s$, and (j_1^s, k_0^s) causes a retransmission. Therefore,

$$\mathbb{P}\left[\boldsymbol{A_2}\right] = \underbrace{\frac{1}{n_0 n_{\text{pod}} - 1}}_{\text{connect to } k_0^s} \sum_{k_0^s \in \mathcal{T}_0^s \setminus \{i_0^s\}} r(j_1^s, k_0^s) \tag{A.12c}$$

- event $\boldsymbol{A_3}$: the connection also goes through some (j_1^s, ℓ_2) and (j_1^s, ℓ_2) causes a retransmission, which occurs with probability

$$\mathbb{P}\left[\boldsymbol{A_3}\right] = \underbrace{\frac{n_0(n_{\text{pod}} - 1)}{n_0 n_{\text{pod}} - 1}}_{\text{leave pod } s} \underbrace{\frac{1}{n_2}}_{\substack{\text{go through} \\ \ell_2}} \sum_{\ell_2 \in \mathcal{T}_2} r(j_1^s, \ell_2) \tag{A.12d}$$

- event $\boldsymbol{A_4}$: the connection also goes through some (ℓ_2, m_1^t), $t \neq s$, and (ℓ_2, m_1^t) causes a retransmission, so that

$$\mathbb{P}\left[\boldsymbol{A_4}\right] = \underbrace{\frac{n_0}{n_0 n_{\text{pod}} - 1}}_{\text{go to pod } t} \underbrace{\frac{1}{n_1 n_2}}_{\substack{\text{go through} \\ (\ell_2, m_1^t)}} \sum_{\substack{\ell_2 \in \mathcal{T}_2, \\ m_1^t \in \mathcal{T}_1^t, \\ t \in [n_{\text{pod}}] \setminus s}} r(\ell_2, m_1^t) \tag{A.12e}$$

- event $\boldsymbol{A_5}$: the connection also goes through some (m_1^t, u_0^t), $t \neq s$, and (m_1^t, u_0^t) causes a retransmission. Thus,

$$\mathbb{P}\left[\boldsymbol{A_5}\right] = \underbrace{\frac{1}{n_0 n_{\text{pod}} - 1}}_{\text{go to pod } t} \underbrace{\frac{1}{n_1}}_{\substack{\text{go through} \\ m_1^t}} \sum_{\substack{m_1^t \in \mathcal{T}_1^t, \\ u_0^t \in \mathcal{T}_0^t, \\ t \in [n_{\text{pod}}] \setminus s}} r(m_1^t, u_0^t) \tag{A.12f}$$

Similarly for level 2, let $\boldsymbol{B_0}$ be the event that a connection goes through link (j_1^s, ℓ_2), so that its probability is

$$\mathbb{P}\left[\boldsymbol{B_0}\right] = \underbrace{\frac{1}{n_{\text{pod}}}}_{\text{start in pod } s} \underbrace{\frac{n_0(n_{\text{pod}} - 1)}{n_0 n_{\text{pod}} - 1}}_{\text{leave pod } s} \underbrace{\frac{1}{n_1 n_2}}_{\substack{\text{go through} \\ (j_1^s, \ell_2)}} \tag{A.13a}$$

Then, link (j_1^s, ℓ_2) gets a bad vote if a connection goes through (j_1^s, ℓ_2) (event $\boldsymbol{B_0}$) *and* either of the following occurs:

- event $\boldsymbol{B_1}$: (j_1^s, ℓ_2) causes a retransmission, i.e.,

$$\mathbb{P}\left[\boldsymbol{B_1}\right] = r(j_1^s, \ell_2) \tag{A.13b}$$

- event $\boldsymbol{B_2}$: the connection also goes through some (i_0^s, j_1^s) and (i_0^s, j_1^s) causes a retransmission. Then,

$$\mathbb{P}\left[\boldsymbol{B_2}\right] = \underbrace{\frac{1}{n_0}}_{\text{start in } i_0^s} \sum_{i_0^s \in \mathcal{T}_0^s} r(i_0^s, j_1^s) \tag{A.13c}$$

- event $\boldsymbol{B_3}$: the connection also goes through some (ℓ_2, m_1^t), $t \neq s$, and (ℓ_2, m_1^t) causes a retransmission, which yields

$$\mathbb{P}\left[\boldsymbol{B_3}\right] = \underbrace{\frac{1}{n_1(n_{\text{pod}} - 1)}}_{\text{go through } m_1^t} \sum_{\substack{m_1^t \in \mathcal{T}_1^t, \\ t \in [n_{\text{pod}}] \setminus s}} r(\ell_2, m_1^t) \tag{A.13d}$$

- event $\boldsymbol{B_4}$: the connection also goes through some (m_1^t, u_0^t), $t \neq s$, and (m_1^t, u_0^t) causes a retransmission. Therefore,

$$\mathbb{P}\left[\boldsymbol{B_4}\right] = \underbrace{\frac{1}{n_0 n_1(n_{\text{pod}} - 1)}}_{\substack{\text{go through} \\ (m_1^t, n_0^t),\ t \neq s}} \sum_{\substack{m_1^t \in \mathcal{T}_1^t, \\ u_0^t \in \mathcal{T}_0^t, \\ t \in [n_{\text{pod}}] \setminus s}} r(m_1^t, u_0^t) \tag{A.13e}$$

To obtain the lower bound in (A.8a), note that a bad link receives at least as many bad votes as retransmissions it causes. Therefore, the probability of Vigil voting "bad" for a bad link is larger than the probability of that link causing a retransmission. Explicitly, using the fact that failure and routing are independent and $r = r_b$, (A.12) and (A.13) give

$$\begin{aligned} s_b &\geq \min\left[\mathbb{P}(\boldsymbol{A}_0 \cap \boldsymbol{A}_1), \mathbb{P}(\boldsymbol{B}_0 \cap \boldsymbol{B}_1)\right] \\ &= \min\left[\frac{1}{n_0 n_1 n_{\text{pod}}}, \frac{1}{n_1 n_2 n_{\text{pod}}} \frac{n_0(n_{\text{pod}} - 1)}{n_0 n_{\text{pod}} - 1}\right] r_b. \end{aligned}$$

The assumption that $n_{\text{pod}} \geq 1 + \frac{n_2(n_0-1)}{n_0(n_0-n_2)}$ makes the first term smaller than the second and yields (A.8a).

In contrast, the upper bound in (A.8b) is obtained by applying the union bound [53] to (A.12) and (A.13). Indeed, this leads to the following inequalities for the probability of Vigil voting "bad" for a good level 1 and level 2 link:

$$\begin{aligned} v_{g,1} &= \mathbb{P}\left[\boldsymbol{A}_0 \cap (\boldsymbol{A}_1 \cup \boldsymbol{A}_2 \cup \boldsymbol{A}_3 \cup \boldsymbol{A}_4 \cup \boldsymbol{A}_5)\right] \\ &\leq \mathbb{P}[\boldsymbol{A}_0] \left(\sum_{i=1}^{5} \mathbb{P}[\boldsymbol{A}_i]\right) \qquad \text{(A.14a)} \\ v_{g,2} &= \mathbb{P}\left[\boldsymbol{B}_0 \cap (\boldsymbol{B}_1 \cup \boldsymbol{B}_2 \cup \boldsymbol{B}_3 \cup \boldsymbol{B}_4)\right] \\ &\leq \mathbb{P}[\boldsymbol{B}_0] \left(\sum_{i=1}^{4} \mathbb{P}[\boldsymbol{B}_i]\right) \qquad \text{(A.14b)} \end{aligned}$$

where $v_{g,1}$ and $v_{g,2}$ denote the probability of a good level 1 and level 2 link being voted bad, respectively. Note that once again used the independence between failures and routing. From (A.14), it is straightforward to see that $v_g \leq \max\left[v_{g,1}, v_{g,2}\right]$.

To obtain (A.8b), we first bound (A.14) by assuming that all k bad links belong to the event $\boldsymbol{A}_i$ and $\boldsymbol{B}_i$, $i \geq 2$, that maximize $v_{g,1}$ and $v_{g,2}$. For a good level 1 link, it is straightforward to see from (A.12) that since $n_0 \geq n_2$, event $\boldsymbol{A}_3$ has the largest coefficient. Thus, taking all links to be good except for k bad links satisfying $\boldsymbol{A}_3$ one has

$$v_{g,1} \leq \frac{1}{n_0 n_1 n_{\text{pod}}} \frac{n_0(n_{\text{pod}}-1)}{n_0 n_{\text{pod}}-1} \times \left[\left(4 - \frac{k}{n_2} + \frac{2(n_0-1)}{n_0(n_{\text{pod}}-1)}\right) r_g + \frac{k}{n_2} r_b\right], \qquad \text{(A.15)}$$

which holds for $k \leq n_2$. Similarly for a good level 2 link, since $n_{\text{pod}} \geq \frac{n_0}{n_1} + 1$ it holds from (A.13) that event $\boldsymbol{B}_2$ has the largest coefficient. Therefore,

$$v_{g,2} \leq \frac{1}{n_1 n_2 n_{\text{pod}}} \frac{n_0(n_{\text{pod}}-1)}{n_0 n_{\text{pod}}-1} \times \left[\left(4 - \frac{k}{n_0}\right) r_g + \frac{k}{n_0} r_b\right], \qquad \text{(A.16)}$$

which holds for $k \leq n_0$. Straightforward algebra shows that for $n_{\text{pod}} \geq 2$, $v_{g,2} \geq v_{g,1}$, from which (A.8a) yields.

□

□

Appendix B

Video Delivery And QoS

B.1 Mechanical Turk Experiment

The motivation was to further validate the QoE metrics (stalls and skips) we selected to evaluate video quality. For that purpose, we used a high quality (HD) documentary about Buckingham Palace[1]. The video was divided into equal sized segments of 1 min each, and different types of impairments were introduced in those segments. Due to of logistics constraints, only results for stalls are available. Specifically, we considered: 1) a single stall of variable duration at a random location in the video; 2) multiple stalls of small (≈ 0.5 sec), medium (≈ 1 sec), and long (≈ 1.5 sec) durations, evenly distributed in the segment; 3) multiple stalls with the same distribution in duration, but now closely spaced (0.1 sec) in a burst. In 2) and 3) we varied the number of stalls. The quality of the video segments was evaluated on a $0-5$ scale (0 being the lowest quality) by 20 users recruited through Amazon's Mechanical Turk market. For calibration purposes, users were first presented with an unimpaired video segment, and told to assign it a rating of 5.

Results of the study are presented in Figure **??**, which confirms a strong cor-

[1] `https://www.youtube.com/watch?v=jffKwoWjXtg`.

relation (error bars again correspond to 95 percent confidence intervals) between stalls, both number and duration, and video quality. The limited size of the study is clearly insufficient for broad conclusions, but it further confirms previous QoE studies [18, 69, 50] and the impact of stalls on video quality. Hence, fewer/shorter stalls do translate into higher video quality.

B.2 Server Selection Algorithm

Server selection can be viewed as a Stackelberg game between the clients and the provider, with the provider as the leader and clients as the followers. Once assigned servers, clients seek to maximize their performance by scheduling requests to servers accordingly. Given this behavior, the provider's goal is to assign servers so as to minimize the 95^{th} percentile cost. This non-convex cost function together with the online nature of the game make computing the optimal assignment strategy hard.

We therefore propose a semi-online greedy optimization that is run every 5 mins and uses the current estimate of the 95^{th} percentile cost to assign client's to servers in a way that meets their rate guarantees while minimizing cost. Specifically, the optimization maintains an estimate of the number of client's expected to arrive from each region (clients in a region have similar bandwidth profiles and share the same connections to servers). Given these estimates, it seeks to identify which assignment of servers for each group of client results in the smallest increase in the current 95^{th} percentile cost. Furthermore, while the optimization's goal is to minimize peering costs, it acknowledges that this should not be at the expense of poor performance for the clients. Thus, it also includes two additional constraints:

$$E\left[\sum_j \alpha_{ij} R_{ij} - T\right] \geq 0 \tag{B.1}$$

$$E\left[\left(\sum_j \alpha_{ij} R_{ij} - T\right)^2\right] \leq \gamma \tag{B.2}$$

where α_{ij} is the number of requests client i sends to server j and R_{ij} is the rate in chunks per second from that client's region to server j.

Note that, reusing the notation of Section **??**, B.2 can be written as $a_i T Q a_i + b^T a_i + c \leq \gamma$ where Q is a matrix with $Q_{ii} = Var(R_i) + \widehat{R}_i^2$ and $Q_{ij} = \widehat{R}_i \widehat{R}_j$, b is a vector where $b_i = -2T\widehat{R}_i$, and $c = T^2$.

Take $\mathcal{F}_j$ as the current 95^{th} percentile cost on peering link j, $\mathcal{L}_j$ the current load on peering link j, and m_i the expected number of clients arriving from region i in the current decision period. We aim to solve the following optimization:

$$\begin{aligned}
\min_{\alpha_{ij}} \quad & \max_{B_j}(B_j + \mathcal{L}_j - \mathcal{F}_j, 0) \\
s.t. \quad & B_j = \sum_i m_i \sum_j \alpha_{ij} \widehat{R}_{ij} \\
& \sum_j \alpha_{ij} \widehat{R}_{ij} \geq T_i \quad \forall i \\
& \alpha_{ij} \leq w_{max} \\
& \alpha_i^T Q_i \alpha_i + b^T \alpha_i + T^2 \leq \gamma^2 \\
& \mathcal{B}_j + \mathcal{L}_j \leq C_j
\end{aligned}$$

where w_{max} is the maximum window size allowed on the clients, and the server selection algorithm assigns all servers with $\alpha_{ij} > 0$ to region i. It is straightforward to show that Q in the above equations is positive semidefinite. Therefore, the optimization is convex and can be solved efficiently.

Bibliography

[1] Closing the network diagnostics gap with vigil (extended version). https://www.dropbox.com/s/vupgssrby3z40xu/techreport.pdf.

[2] Pc3000. emulab. https://wiki.emulab.net/wiki/pc3000.

[3] RFC 791: Internet Protocol, 1981. DARPA.

[4] Soft Perfect Connection Emulator. https://www.softperfect.com/, 2012. [Online].

[5] Kristin L Adair, Alan P Levis, and Susan I Hruska. Expert network development environment for automating machine fault diagnosis. In *Aerospace/Defense Sensing and Controls*, pages 506–515. International Society for Optics and Photonics, 1996.

[6] Bhavish Agarwal, Ranjita Bhagwan, Tathagata Das, Siddharth Eswaran, Venkata N Padmanabhan, and Geoffrey M Voelker. Netprints: Diagnosing home network misconfigurations using shared knowledge. In *NSDI*, volume 9, pages 349–364, 2009.

[7] Marcos K Aguilera, Jeffrey C Mogul, Janet L Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance debugging for distributed systems of black boxes. *ACM SIGOPS Operating Systems Review*, 37(5):74–89, 2003.

[8] Akamai state of the Internet report, q2, 2016. `http://www.akamai.com/dl/akamai/akamai-soti-q114.pdf`.

[9] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Francis Matus, Rong Pan, Navindra Yadav, George Varghese, et al. CONGA: Distributed congestion-aware load balancing for datacenters. *ACM SIGCOMM Computer Communication Review*, 44(4):503–514, 2014.

[10] J.G. Apostolopoulos and M.D. Trott. Path diversity for enhanced media streaming. *IEEE Comm. Mag.*, 42(8), August 2004.

[11] R. Arratia and L. Gordon. Tutorial on large deviations for the binomial distribution. *Bulletin of Mathematical Biology*, 51(1):125–131, 1989.

[12] B. Arzani, R. Guerin, and A. Ribeiro. A distributed routing protocol for predictable rates in wireless mesh networks. In *Proc. IEEE ICNP*, Austin, TX, October 2012.

[13] Behnaz Arzani, Alexander Gurney, Shuotian Cheng, Roch Guerin, and Boon Thau Loo. Impact of path characteristics and scheduling policies on MPTCP performance. In *Proc. PAMS*, 2014.

[14] Behnaz Arzani, Alexander Gurney, Sitian Cheng, Roch Guerin, and Boon Thau Loo. Deconstructing MPTCP performance. In *ICNP*, 2014.

[15] Behnaz Arzani, Alexander Gurney, Bo Li, Xianglong Han, Roch Guerin, and Boon Thau Loo. Fixroute: A unified logic and numerical tool for provably safe internet traffic engineering. *arXiv preprint arXiv:1511.08791*, 2015.

[16] Ender Ayanoglu, I Chih-Lin, Richard D Gitlin, and James E Mazo. Diversity coding for transparent self-healing and fault-tolerant communication networks. *IEEE Transactions on communications*, 41(11):1677–1686, 1993.

[17] Paramvir Bahl, Ranveer Chandra, Albert Greenberg, Srikanth Kandula, David A Maltz, and Ming Zhang. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *ACM SIGCOMM Computer Communication Review*, volume 37, pages 13–24. ACM, 2007.

[18] Athula Balachandran, Vyas Sekar, Aditya Akella, Srinivasan Seshan, Ion Stoica, and Hui Zhang. Developing a predictive model of quality of experience for Internet video. In *Proc. ACM SIGCOMM*, 2013.

[19] M. Ball. The state and future of Netflix v. HBO in 2015. REDEF Originals, May 2015. `https://redef.com/original/the-state-and-future-of-netflix-v-hbo-in-2015`.

[20] Dipyaman Banerjee, Venkateswara Madduri, and Mudhakar Srivatsa. A framework for distributed monitoring and root cause analysis for large IP networks. In *IEEE SRDS*, pages 246–255, 2009.

[21] Boaz Barak, Sharon Goldberg, and David Xiao. Protocols and lower bounds for failure localization in the internet. In *EUROCRYPT*, pages 341–360, 2008.

[22] Dimitris Bertsimas and John N. Tsitsiklis. *Introduction to linear optimization*. Athena Scientific, 1997.

[23] Peter Bodík, Ishai Menache, Mosharaf Chowdhury, Pradeepkumar Mani, David A Maltz, and Ion Stoica. Surviving failures in bandwidth-constrained datacenters. In *ACM SIGCOMM*, pages 431–442, 2012.

[24] J. Bonte. Online video: What do media companies *Really* want? White paper, Cisco Internet Business Solutions Group, July 2010.

[25] Youmna Borghol, Sebastien Ardon, Niklas Carlsson, Derek Eager, and Anirban Mahanti. The untold story of the clones: content-agnostic factors that impact YouTube video popularity. In *Proc. SIGKDD*, 2012.

[26] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[27] Mark Burgess. Probabilistic anomaly detection in distributed computer networks. *Science of Computer Programming*, 60(1):1–26, 2006.

[28] Prabir Burman. A comparative study of ordinary cross-validation, v-fold cross-validation and the repeated learning-testing methods. *Biometrika*, 76(3):503–514, 1989.

[29] M. Carbone and L. Rizzo. Dummynet revisited. *ACM SIGCOMM Computer Communication Review*, 40(2), March 2010.

[30] Content delivery summit 2013. `http://www.contentdeliverysummit.com/2013/`.

[31] Content delivery summit 2014. `http://www.contentdeliverysummit.com/2014/`.

[32] Content delivery summit 2015. `http://www.contentdeliverysummit.com/2015/`.

[33] Content delivery summit 2016. `http://www.contentdeliverysummit.com/2016/`.

[34] C. Chen, S. Inguva, A. Rankin, and A. Kokaram. A subjective study for the design of multi-resolution ABR video streams with the VP9 codec. In *Proc. Intl. Symp. Electronic Imaging: Human Visual Perception*, San Francisco, CA, February 2016.

[35] J. Chen, S.H.G. Chan, and V.O.K. Li. Multipath routing for video delivery over bandwidth-limited networks. *IEEE J. Select. Areas Commun*, 22(10), 2004.

[36] Mike Chen, Alice X Zheng, Jim Lloyd, Michael Jordan, Eric Brewer, et al. Failure diagnosis using decision trees. In *Autonomic Computing, 2004. Proceedings. International Conference on*, pages 36–43. IEEE, 2004.

[37] X. Chen, M. Chamania, A. Jukan., A.C. Drummond, and N.L.S Da Fonseca. On the benefits of multipath routing for distributed data-intensive applications with high bandwidth requirements and multidomain reach. In *Proc. Comm. Netw. & Serv. Research Conf. (CNSR'09)*, Moncton, NB, May 2009.

[38] Yan Chen, David Bindel, Hanhee Song, and Randy H Katz. An algebraic approach to practical and scalable overlay network monitoring. *ACM SIGCOMM Computer Communication Review*, 34(4):55–66, 2004.

[39] Yen-Yang Michael Chen, Anthony Accardi, Emre Kiciman, David A Patterson, Armando Fox, and Eric A Brewer. Path-based failure and evolution management. In *USENIX NSDI*, 2004.

[40] Yung-Chih Chen, Don Towsley, and Ramin Khalili. MSPlayer: Multi-source and multi-path leveraged YoutubER. In *CoNEXT*, 2014.

[41] Xu Cheng, Jiangchuan Liu, and Cameron Dale. Understanding the characteristics of Internet short video sharing: A YouTube-based measurement study. *IEEE Transactions on Multimedia*, 2013.

[42] Conviva. 2014 viewer experience report 2014. `http://lp.conviva.com/rs/901-ZND-194/images/2014%20Conviva%20Viewer%20Experience%20Report.pdf/`,.

[43] Conviva. 2015 end of year report and 2016 predictions.

[44] Conviva. 2015 viewer experience report. `http://lp.conviva.com/rs/901-ZND-194/images/Conviva_Viewer_Experience_Report_2015_Final.pdf`.

[45] Xavier Corbillon, Ramon Aparicio-Pardo, Nicolas Kuhn, Géraldine Texier, and Gwendal Simon. Cross-layer scheduler for video streaming over MPTCP. In *Proceedings of the 7th International Conference on Multimedia Systems*. ACM, 2016.

[46] Cox's OTT service shows the way to revive US cable TV industry, July 2013. `http://press.ihs.com/press-release/design-supply-chain-media/coxs-ott-service-shows-way-revive-us-cable-tv-industry`.

[47] Mark Crovella and Anukool Lakhina. Method and apparatus for whole-network anomaly diagnosis and method to detect and classify network anomalies using traffic feature distributions, 2014. US Patent 8,869,276.

[48] A. Dhamdhere, R. Teixeira, C. Dovrolis, and C. Diot. NetDiagnoser: Troubleshooting network unreachabilities using end-to-end probes and routing data. In *ACM CoNEXT*, 2007.

[49] Giorgos Dimopoulos, Ilias Leontiadis, Pere Barlet-Ros, Konstantina Papagiannaki, and Peter Steenkiste. Identifying the root cause of video streaming issues on mobile devices.

[50] Florin Dobrian, Vyas Sekar, Asad Awan, Ion Stoica, Dilip Joseph, Aditya Ganjam, Jibin Zhan, and Hui Zhang. Understanding the impact of video quality on user engagement. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 362–373. ACM, 2011.

[51] Nick Duffield. Network tomography of binary network performance characteristics. *IEEE Transactions on Information Theory*, 52(12):5373–5388, 2006.

[52] Nick G. Duffield, Vijay Arya, Rémy Bellino, Timur Friedman, Joseph Horowitz, D. Towsley, and Thierry Turletti. Network tomography from aggregate loss reports. *Performance Evaluation*, 62(1):147–163, 2005.

[53] William Feller. *An Introduction to Probability Theory and Its Applications*, volume 1. Wiley, 1968.

[54] Bernard Fortz and Mikkel Thorup. Internet traffic engineering by optimizing ospf weights. In *INFOCOM 2000. Nineteenth annual joint conference of the IEEE computer and communications societies. Proceedings. IEEE*, volume 2, pages 519–528. IEEE, 2000.

[55] Qiang Fu, Jian-Guang Lou, Yi Wang, and Jiang Li. Execution anomaly detection in distributed systems through unstructured log analysis. In *Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on*, pages 149–158. IEEE, 2009.

[56] Jr. G. D. Forney, L. Brown, M. V. Eyuboglu, and J. L. Moran III. The V.34 high-speed modem standard. *IEEE Comm. Mag.*, 54(12):28–33, December 1996.

[57] Moshe Gabel, Kento Sato, Daniel Keren, Satoshi Matsuoka, and Assaf Schuster. Latent fault detection with unbalanced workloads. Technical report, Lawrence Livermore National Laboratory (LLNL), Livermore, CA, 2014.

[58] Deepak Ganesan, Ramesh Govindan, Scott Shenker, and Deborah Estrin. Highly-resilient, energy-efficient multipath routing in wireless sensor networks. *ACM SIGMOBILE Mobile Computing and Communications Review*, 5(4), 2001.

[59] M.-N. Garcia, F. De Simone, S. Tavakoli, N. Staelens, S. Egger, K. Brunnstróm, and A. Raake. Quality of experience and HTTP adaptive streaming: A re-

view of subjective studies. In *Proc. 6th IEEE Intl. Workshop on Quality of Multimedia Experience (QoMEX)*, Singapore, Singapore, September 2014.

[60] Mojgan Ghasemi, Theophilus Benson, and Jennifer Rexford. RINC: Real-time Inference-based Network diagnosis in the Cloud. Technical report, Princeton University, 2015. `https://www.cs.princeton.edu/research/techreps/TR-975-14`.

[61] Phillipa Gill, Martin Arlitt, Zongpeng Li, and Anirban Mahanti. Youtube traffic characterization: a view from the edge. In *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, pages 15–28. ACM, 2007.

[62] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding network failures in data centers: Measurement, analysis, and implications. *ACM SIGCOMM Computer Communication Review*, 41(4):350–361, 2011.

[63] L. Golubchik, J.C.S. Lui, T.F. Tung, A.L.H. Chow, W.-J. Lee, G. Franceschinis, and C. Anglano. Multi-path continuous media streaming: what are the benefits? *Performance Evaluation*, 49(1–4), September 2002.

[64] Mehmet H. Gunes and Kamil Sarac. Resolving IP aliases in building traceroute-based internet maps. *IEEE/ACM Transactions on Networking*, 17(6):1738–1751, 2009.

[65] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, et al. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 139–152. ACM, 2015.

[66] Brandon Heller, Colin Scott, Nick McKeown, Scott Shenker, Andreas Wundsam, Hongyi Zeng, Sam Whitlock, Vimalkumar Jeyakumar, Nikhil Hand-

igol, James McCauley, et al. Leveraging SDN layering to systematically troubleshoot networks. In *ACM SIGCOMM HotSDN*, pages 37–42, 2013.

[67] Tom Henderson, Sally Floyd, Andrei Gurtov, and Yoshifumi Nishida. The NewReno modification to TCP's fast recovery algorithm. *RFC6582*, 2012.

[68] Christian E. Hopps. RFC 2992: Analysis of an Equal-Cost Multi-Path algorithm, 2000.

[69] Tobias Hoßfeld, Raimund Schatz, Ernst Biersack, and Louis Plissonneau. Internet video delivery in YouTube: From traffic measurements to quality of experience. In *Data Traffic Monitoring and Analysis*. Springer, 2013.

[70] Ling Huang, XuanLong Nguyen, Minos Garofalakis, Michael I Jordan, Anthony Joseph, and Nina Taft. In-network pca and anomaly detection. In *Advances in Neural Information Processing Systems*, pages 617–624, 2006.

[71] Yiyi Huang, Nick Feamster, and Renata Teixeira. Practical issues with using network tomography for fault diagnosis. *ACM SIGCOMM Computer Communication Review*, 38(5):53–58, 2008.

[72] Yiyi Huang, Nick Feamster, and Renata Teixeira. Practical issues with using network tomography for fault diagnosis. *ACM SIGCOMM Computer Communication Review*, 38(5):53–58, 2008.

[73] Olumuyiwa Ibidunmoye, Francisco Hernández-Rodriguez, and Erik Elmroth. Performance anomaly detection and bottleneck identification. *ACM Computing Surveys (CSUR)*, 48(1):4, 2015.

[74] U. Javed, M. Suchara, J. he, and J. Rexford. Multipath protocol for delay-sensitive traffic. In *Proc. COMSNETS'09*, Bangalore, India, January 2009.

[75] Brian Weatherred Johnson, Steve HS Kim, Edward James Leo Jr, and Dennis Lee. Link aggregation path selection method, 2003. US Patent 6,535,504.

[76] Srikanth Kandula, Dina Katabi, and Jean-Philippe Vasseur. Shrink: A tool for failure diagnosis in IP networks. In *ACM SIGCOMM MineNet*, pages 173–178, 2005.

[77] Frank Kelly. Fairness and stability of end-to-end congestion contro. *European journal of control*, 2003.

[78] Ramin Khalili, Nicolas Gast, Miroslav Popovic, Utkarsh Upadhyay, and Jean-Yves Le Boudec. MPTCP is not Pareto-optimal: performance issues and a possible solution. In *Proc. ACM CoNEXT*, 2012.

[79] Andreas Kind, Marc Ph Stoecklin, and Xenofontas Dimitropoulos. Histogram-based traffic anomaly detection. *IEEE Transactions on Network and Service Management*, 6(2), 2009.

[80] Ramana Rao Kompella, Jennifer Yates, Albert Greenberg, and Alex C. Snoeren. IP fault localization via risk modeling. In *USENIX NSDI*, pages 57–70, 2005.

[81] S. S. Krishnan and R. K. Sitaraman. Video stream quality impacts viewer behavior: Inferring causality using quasi-experimental designs. In *Proc. ACM IMC'12*, Boston, MA, November 2012.

[82] J. Kufa and T. Kratochvil. Comparison of H.265 and VP9 coding efficiency for full HDTV and ultra HDTV applications. In *Radioelektronika (RADIOELEKTRONIKA), 2015 25th International Conference*, pages 168–171, April 2015.

[83] Max Kuhn and Kjell Johnson. *Applied predictive modeling*. Springer, 2013.

[84] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. FlowRadar: A better NetFlow for data centers. In *USENIX NSDI*, pages 311–324, 2016.

[85] Chang Liu, Ting He, Ananthram Swami, Don Towsley, Theodoros Salonidis, and Kin K Leung. Measurement design framework for network tomography using fisher information. *ITA AFM*, 2013.

[86] Hongqiang Harry Liu, Ye Wang, Yang Richard Yang, Hao Wang, and Chen Tian. Optimizing cost and performance for content multihoming. In *Proc. ACM SIGCOMM*, 2012.

[87] Yuliang Liú, Rui Miao, Changhoon Kim, and Minlan Yuú. LossRadar: Fast detection of lost packets in data center networks. In *ACM CoNEXT*, pages 481–495, 2016.

[88] Ratul Mahajan, Neil Spring, David Wetherall, and Thomas Anderson. User-level internet path diagnosis. *ACM SIGOPS Operating Systems Review*, 37(5):106–119, 2003.

[89] Pietro Marchetta, Antonio Montieri, Valerio Persico, Antonio Pescapé, Ítalo Cunha, and Ethan Katz-Bassett. How and how much traceroute confuses our understanding of network paths. In *IEEE LANMAN*, pages 1–7, 2016.

[90] Matt Mathis, John Heffner, Peter O?Neil, and Pete Siemsen. Pathdiag: automated tcp diagnosis. In *Passive and Active Network Measurement*, pages 152–161. Springer, 2008.

[91] Microsoft. Windows ETW. `https://msdn.microsoft.com/en-us/library/windows/desktop/bb968803(v=vs.85).aspx`, 2000. [Online].

[92] Microsoft. SQLIO. `http://www.microsoft.com/en-us/download/details.aspx?id=20163`, 2012. [Online].

[93] Microsoft. TestLimit. `http://blogs.msdn.com/b/vijaysk/archive/2012/10/27/tools-to-simulate-cpu-memory-disk-load.aspx`, 2012. [Online].

[94] Microsoft. BWorld Robot Control Software. `https://chocolatey.org/packages/newt`, 2013. [Online].

[95] Mininet. `http://mininet.org/`.

[96] APS Mosek. The MOSEK optimization software. *Online at http://www. mosek. com*, 54, 2010.

[97] Matthew K Mukerjee, Ilker Nadi Bozkurt, Bruce Maggs, Srinivasan Seshan, and Hui Zhang. The impact of brokers on the future of content delivery. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, pages 127–133. ACM, 2016.

[98] Matthew K Mukerjee, David Naylor, Junchen Jiang, Dongsu Han, Srinivasan Seshan, and Hui Zhang. Practical, real-time centralized control for CDN-based live video delivery. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. ACM, 2015.

[99] Radhika Niranjan Mysore, Ratul Mahajan, Amin Vahdat, and George Varghese. Gestalt: Fast, unified fault localization for networked systems. In *USENIX ATC*, pages 255–267, 2014.

[100] Srinivas Narayana, Wenjie Jiang, Jennifer Rexford, and Mung Chiang. Joint server selection and routing for geo-replicated services. In *Proc. UCC*, 2013.

[101] Netflix, openconnect. `https://openconnect.itp.netflix.com/`.

[102] Ashkan Nikravesh, David R Choffnes, Ethan Katz-Bassett, Z Morley Mao, and Matt Welsh. Mobile network performance from user devices: A longi-

tudinal, multidimensional analysis. In *International Conference on Passive and Active Network Measurement*, pages 12–22. Springer, 2014.

[103] Nagao Ogino, Takeshi Kitahara, Shin'ichi Arakawa, Go Hasegawa, and Masayuki Murata. Decentralized boolean network tomography based on network partitioning. In *IEEE/IFIP NOMS*, pages 162–170, 2016.

[104] J. Ozer. Netflix finds x265 20% more efficient than VP9. streaming media, September 2016. Available at `http://www.streamingmedia.com/Articles/Editorial/Featured-Articles/Netflix-Finds-x265-20-More-Efficient-than-VP9-113346.aspx`,.

[105] Christoph Paasch and Olivier Bonaventure. Multipath TCP. *Communications of the ACM*, 57(4):51–57, 2014.

[106] Christoph Paasch, Simone Ferlin, Ozgu Alay, and Olivier Bonaventure. Experimental evaluation of multipath TCP schedulers. In *Proceedings of the 2014 ACM SIGCOMM workshop on Capacity sharing workshop*. ACM, 2014.

[107] Christoph Paasch, Ramin Khalili, and Olivier Bonaventure. On the benefits of applying experimental design to improve Multipath TCP. In *Proc. ACM CoNEXT*, 2013.

[108] Venkata N Padmanabhan, Sriram Ramabhadran, and Jitendra Padhye. Netprofiler: Profiling wide-area networks using peer cooperation. In *Peer-to-Peer Systems IV*, pages 80–92. Springer, 2005.

[109] Sinno Jialin Pan and Qiang Yang. A survey on transfer learning. *Knowledge and Data Engineering, IEEE Transactions on*, 22(10):1345–1359, 2010.

[110] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu

Wu, et al. Ananta: Cloud scale load balancing. *ACM SIGCOMM Computer Communication Review*, 43(4):207–218, 2013.

[111] Karl Pearson. Liii. on lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 2(11):559–572, 1901.

[112] Qiuyu Peng, Anwar Walid, and Steven H Low. Multipath TCP: Analysis and design. In *Proc. ACM SIGMETRICS*, Pittsburgh, PA, June 2013.

[113] Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr. Sharma, and Arvind Krishnamurthy. Designing distributed systems using approximate synchrony in data center networks. In *USENIX NSDI*, pages 43–57, 2015.

[114] 6 key factors to consider when choosing a transparent caching solution. `http://qwilt.com/downloads/Qwilt-TransparentCaching-6KeyFactors.pdf`.

[115] Marjan Radi, Behnam Dezfouli, Kamalrulnizam Abu Bakar, and Malrey Lee. Multipath routing in wireless sensor networks: Survey and research challenges. *Sensors*, 12(1), January 2012.

[116] Costin Raiciu, Christoph Paasch, Sebastien Barre, Alan Ford, Michio Honda, Fabien Duchene, Olivier Bonaventure, and Mark Handley. How hard can it be? designing and implementing a deployable Multipath TCP. In *Proc. NSDI*, 2012.

[117] Ashwin Rao, Arnaud Legout, Yeon-sup Lim, Don Towsley, Chadi Barakat, and Walid Dabbous. Network characteristics of video streaming traffic. In *Proc. ACM CoNEXT*, 2011.

[118] Pablo Rodriguez and Ernst W Biersack. Dynamic parallel access to replicated content in the Internet. *IEEE/ACM Transactions on Networking*, 2002.

[119] Arjun Roy, Jasmeet Bagga, Hongyi Zeng, and Alex Sneoren. Passive realtime datacenter fault detection. In *ACM NSDI*, 2017.

[120] K. Russel. What the Netflix-Comcast deal really means in plain english. `http://www.businessinsider.com/netflix-comcast-deal-explained-2014-2`.

[121] Sandvine. *2016 Global Internet Phenomena – Latin America & North America*, 2016.

[122] M. Seufert, S. Egger, M. Slanina, T. Zinner, and T. Hossfeld. A survey on qualitty of experience of HTTP adaptive streaming. *IEEE Communications Surveys & Tutorials*, 17(1), April 2014.

[123] Amanpreet Singh, Mei Xiang, Andreas Könsgen, Carmelita Goerg, and Yasir Zaki. Enhancing fairness and congestion control in Multipath TCP. In *Proc. IEEE WMNC*, 2013.

[124] V Singh, T Karkkainen, J Ott, S Ahsan, and L Eggert. Multipath rtp (mprtp). Technical report, IETF Internet-Draft, 2012.

[125] J. Søgaard, S. Tavakoli, K. Brunnström, and N. Garcia. Subjective analysis and objective characterization of adaptive bitrate videos. In *Proc. Intl. Symp. Electronic Imaging: Human Vision and Electroninc Imaging*, San Francisco, CA, February 2016.

[126] Rade Stanojevic, Nikolaos Laoutaris, and Pablo Rodriguez. On economic heavy hitters: shapley value analysis of 95th-percentile pricing. In *SIGCOMM IMC*, 2010.

[127] Srikanth Sundaresan, Nick Feamster, and Renata Teixeira. Locating throughput bottlenecks in home networks. *ACM SIGCOMM Computer Communication Review*, 44(4):351–352, 2015.

[128] Srikanth Sundaresan, Yan Grunenberger, Nick Feamster, Dina Papagiannaki, Dave Levin, and Renata Teixeira. WTF? Locating performance problems in home networks. Technical report, Georgia Institute of Technology, 2013. `http://hdl.handle.net/1853/46991`.

[129] Ariel Tseitlin. The antifragile organization. *Communications of the ACM*, 56(8):40–44, 2013.

[130] M. Uhrina, L. Sevcik, J. Frnda, and M. Vaculik. Impact of H.265 and VP9 compression standards on the video quality for 4k resolution. In *Telecommunications Forum Telfor (TELFOR), 2014 22nd*, pages 905–908, November 2014.

[131] Kirk Webb, Mike Hibler, Robert Ricci, Austin Clements, and Jay Lepreau. Implementing the Emulab-PlanetLab portal: Experience and lessons learned. In *Proc. WORLDS*, 2004.

[132] Philip M Wells, Koushik Chakraborty, and Gurindar S Sohi. Adapting to intermittent faults in multicore systems. *ACM SIGOPS Operating Systems Review*, 42(2):255–264, 2008.

[133] Patrick Wendell, Joe Wenjie Jiang, Michael J Freedman, and Jennifer Rexford. Donar: decentralized server selection for cloud services. In *Proc. ACM SIGCOMM*, 2010.

[134] Chathuranga Widanapathirana, Jonathan Li, Y Ahmet Sekercioglu, Milosh Ivanovich, and Paul Fitzpatrick. Intelligent automated diagnosis of client device bottlenecks in private clouds. In *Utility and Cloud Computing (UCC), 2011 Fourth IEEE International Conference on*, pages 261–266. IEEE, 2011.

[135] Damon Wischik, Costin Raiciu, Adam Greenhalgh, and Mark Handley. Design, implementation and evaluation of congestion control for Multipath TCP. In *Proc. NSDI*, 2011.

[136] Xin Wu, Daniel Turner, Chao-Chih Chen, David A Maltz, Xiaowei Yang, Lihua Yuan, and Ming Zhang. NetPilot: Automating datacenter network failure mitigation. *ACM SIGCOMM Computer Communication Review*, 42(4):419–430, 2012.

[137] Hao Yin, Xuening Liu, Tongyu Zhan, Vyas Sekar, Feng Qiu, Chuang Lin, Hui Zhang, and Bo Li. Design and deployment of a hybrid CDN-P2P system for live video streaming: experiences with LiveSky. In *ACMMM*, 2009.

[138] Minlan Yu, Albert G Greenberg, David A Maltz, Jennifer Rexford, Lihua Yuan, Srikanth Kandula, and Changhoon Kim. Profiling network performance for multi-tier data center applications. In *NSDI*, 2011.

[139] Kyriakos Zarifis, Tobias Flach, Srikanth Nori, David Choffnes, Ramesh Govindan, Ethan Katz-Bassett, Z Morley Mao, and Matt Welsh. Diagnosing path inflation of mobile client traffic. In *International Conference on Passive and Active Network Measurement*, pages 23–33, 2014.

[140] Yin Zhang, Lee Breslau, Vern Paxson, and Scott Shenker. On the characteristics and origins of internet flow rates. In *ACM SIGCOMM Computer Communication Review*, volume 32, pages 309–322. ACM, 2002.

[141] Yin Zhang, Zihui Ge, Albert Greenberg, and Matthew Roughan. Network anomography. In *ACM SIGCOMM IMC*, 2005.

[142] Mingchen Zhao, Paarijaat Aditya, Ang Chen, Yin Lin, Andreas Haeberlen, Peter Druschel, Bruce Maggs, Bill Wishon, and Miroslav Ponec. Peer-assisted content distribution in Akamai Netsession. In *Proc. IMC*, 2013.

[143] Yao Zhao, Yan Chen, and David Bindel. Towards unbiased end-to-end network diagnosis. In *ACM SIGCOMM Computer Communication Review*, volume 36, pages 219–230. ACM, 2006.

[144] Yao Zhao, Yan Chen, and David Bindel. Towards unbiased end-to-end network diagnosis. *ACM SIGCOMM Computer Communication Review*, 36(4):219–230, 2006.

[145] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y Zhao, et al. Packet-level telemetry in large datacenter networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 479–491. ACM, 2015.

[146] Danyang Zhuo, Monia Ghobadi, Ratul Mahajan, Amar Phanishayee, Xuan Kelvin Zou, Hang Guan, Arvind Krishnamurthy, and Tom Anderson. RAIL: A case for Redundant Arrays of Inexpensive Links in data center networks. In *USENIX NSDI*, 2017.