Failure Diagnosis for Datacenter Applications

Qiao Zhang

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2018

Reading Committee:

Thomas E. Anderson, Chair

Arvind Krishnamurthy, Chair

Xi Wang

Program Authorized to Offer Degree:
Computer Science and Engineering

University of Washington

Abstract

Failure Diagnosis for Datacenter Applications

Qiao Zhang

Co-Chairs of the Supervisory Committee:
Professor Thomas E. Anderson
Computer Science and Engineering

Professor Arvind Krishnamurthy
Computer Science and Engineering

Fast and accurate failure diagnosis remains a major challenge for datacenter operators. Current datacenter applications are increasingly architected around loosely-coupled modular components: each component can scale and evolve independently. However, when application failures occur, they become much harder to detect and localize. The challenges are three-fold: complex component dependency, gray failures, and unpredictable component behaviors.

My thesis is that fast and accurate failure diagnosis for datacenter applications is possible using three key ideas: (1) a global view of component interactions and dependencies, (2) a penalized-regression-based failure localization algorithm that localizes both fail-stop and gray failures, and (3) a network architecture that produces predictable routes, simplifying failure localization without sacrificing load balancing and other network features.

I present two complementary systems to demonstrate this. The first, Deepview, is a system that can localize virtual hard disk (VHD) failures in Infrastructure-as-a-Service clouds. I show that Deepview localizes VHD failures accurately and quickly to compute, storage and network components in production at Microsoft Azure. The second, Volur, is a network architecture that makes in-network routing predictable to the end-hosts. I show that Volur accurately localizes non-fail-stop link or switch failures and approximates state-of-the-art dynamic load balancing schemes.

Table of Contents

List of Figures

List of Tables

Acknowledgments

When I was six, someone asked me what I want to be when I grow up. I said, I want to be a PhD. Little did I know that that premature conviction predisposed the future me to a path more adventurous than usual. And that has made all the difference.

Many people have made grad school a truly unique experience for me. I would like to first thank my advisors Tom and Arvind. Tom is old school, yet extremely refreshing in this fast-changing world of computer science research. He showed me by example what matters in research and how to stay true to that. Arvind is a perennial optimist with one of the most enviable traits for research and mentoring. He inspired me to have high hopes for research and keep them alive no matter what.

I would also like to thank my friends and labmates from grad school: Luheng, Danyang, Haichen, Vincent, Seungyeop, Will, Ray, Colin, Ming, Shumo, Tianqi, Jialin, Naveen, Yuchen, Antoine, Irene, Dan, Adriana, Niel, Kaiyuan, Helgi, Nacho, Eunsol, Max, Dhruv, Xi, Mehrdad, Lakshmi, Vikram, Ali and many others. I am honored to have collaborated with Danyang, Vincent, Simon, Haichen, Tianqi, Chuanxiong, the late Ben, and Shyam. I really value all the whiteboard discussions, late-night hacking and hallway conversations that we have had. I also want to thank Sandy for editing my dissertation manuscript.

Finally, I want to thank my family. My mother Yan implanted the early thought of getting a PhD in me, and perhaps has enjoyed the journey vicariously through me. My father Shuguang brought home a 80286 PC in 1995 and later a whiny modem, and taught me QBasic and FoxPro. My girlfriend Luheng wowed me with the prowess of a woman computer scientist. She convinced me that this wonderful world of computer science should be enjoyed equally between men and women.

As I will enter the software industry soon, I hope to keep the curiosity, the willingness to experiment, and most importantly the sense of adventure alive.

Dedication

To my family

Chapter 1

Introduction

Datacenter applications are server-side applications hosted in remote large-scale computing facilities. This includes Internet services such as search engines, social networks, or online shopping. It also includes cloud computing services such as Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS), or Software-as-a-Service (SaaS).

To meet the growing demand, datacenter applications are increasingly architected around loosely-coupled modular components. Application program flow often involves requesting data from or offloading computation to remote components through remote procedure calls (RPCs) [22]. In normal operation, components only depend on a public interface while hiding implementation details from their users. Shared components include storage, cache, network, naming services, concurrency control, consensus and resource orchestration [18, 27, 26, 60, 98, 80]. Each component may iteself comprise many smaller components.

This architecture has allowed datacenter operators to make enormous progress toward achieving a multitude of design goals, including higher scalability, performance, reliability, availability and lower cost. One of the main advantages is that, with modularity and explicit service contracts between different software teams, a team can scale, evolve, optimize, and harden individual services and components independently on their own.

However, when failures occur, datacenter operators may face significant challenges in diagnosing and locating those failures. While redundancy and other failure-masking techniques can help tolerate failures, they are not universally applicable or effective. In consequence, the inability to localize and handle failures quickly can adversely impact the availability of datacenter applications, impacting external customers, as exemplified in many publicly known datacenter application out-

ages (a collection of those incidents are curated by Bailis et al. [15]). This makes failure diagnosis one of the most important open problems in the area of datacenter systems.

In the rest of this chapter, I will first introduce three challenges for failure diagnosis for datacenter applications. Then, I will propose the principal hypothesis that this thesis will demonstrate, and summarize the contributions.

1.1 Challenges

1.1.1 Complex Component Dependency

Today's datacenter applications are complex. It is not rare to see an application depending on tens to hundreds of underlying components/services. It is also common for each component to be a complex distributed system on its own. How does the application depend on its underlying components? How do those components depend on each other? Such questions are critical for correctly assigning blame when an application experiences a failure. However, the software stack is so deep and complex that the component dependencies are sometimes not fully understood or properly taken into account. This deficiency in understanding can prevent fast and accurate failure diagnosis.

A clear illustration of this problem is the prevailing industry practice for failure diagnosis. Many cloud providers dedicate a team of engineers to the development and operation of each service. That way implementation details that matter for troubleshooting stay within each team. Because applications involve many services, when outages happen, multiple teams from dependent services can become involved in locating and mitigating the failures. This industry practice can be described as a *component-local view*—service owners each check their service-specific monitoring systems for anomalies that are likely correlated with the ongoing incident, and through manual troubleshooting, they try to come to a consensus on the root cause so that the culprit service can take mitigatory actions.

While the component-local view is the most natural approach due to the way engineering organizations are structured around the division of services, it fails to properly take into account the complex component dependency, and thus it is not always effective. First, services are often inter-

connected in complex ways, and even service owners themselves often only have incomplete knowledge of the full set of dependencies with other services. It becomes difficult to come up with the right hypotheses when troubleshooting failures. Second, it can be difficult to distinguish cause from effects looking at components individually and locally. A single root cause often has multiple symptoms at different downstream components. As a result, more than one team can see anomalies in their service monitoring systems, making it time-consuming to come to a consensus on where the true failure is.

1.1.2 Gray Failures

Another consequence of the scale and complexity of cloud services is that when failures happen, they can manifest in a variety of ways.

The simplest way that a component can fail is fail-stop. Either due to a bug or some other unexpected program behavior, the service shuts down and stops taking requests. This is akin to putting "ASSERT" or "PANIC" in the code to handle unexpected behavior. With the right kind of monitoring, locating fail-stop failures is often straightforward. Moreover, software or hardware restart typically restores service availability.

However, there is another class of failures that are harder to detect and whose effects are more insidious, called "gray failures" [58, 76]. This includes performance degradation, random packet drops, non-fatal exceptions and many others. From a local perspective, an engineering team will often think it is better to fail to perform a single request than to crash the entire service. These show up in all kinds of places: compute, storage, network and others. They are sometimes silent and evade detection by monitoring systems, or partial and only affect a subset of client requests, or even input-dependent and only affect requests matching specific patterns. Their symptoms are diverse, with the potential for causing service outages. Detecting and localizing such "gray failures" takes time-consuming manual effort, making fast and accurate failure diagnosis difficult.

1.1.3 Unpredictable Component Behaviors

Due to either non-determinism or information hiding, some component behaviors are unpredictable from the user's perspective. Unpredictable component behaviors make it hard to understand how requests are handled and routed in the system, rendering failure localization techniques that are based on end-to-end signals ineffective. One important case in point is the opaque routing behavior in today's datacenter networks.

Almost all datacenter applications rely on the underlying network to function. Failures in the network can have wide-ranging impact on applications. However, fast and accurate failure localization for datacenter network remains a challenge. Modern datacenter networks delegate route selection to switches in the network, and hide the routing logic from end-hosts. For example, while switches have well-defined routing tables for visibility and control, often there are many equivalent routes with load balancing modules that remain opaque and unpredictable. While this provides a simple way for the network to scale, when the network experiences "gray failures" and is unable to correct the problem itself, opaque routing can significantly hinder failure diagnosis and recovery. It is almost impossible to predict how packets are routed in a network in many cases, so end-hosts cannot localize and recover from network device failures.

1.2 Thesis Statement

My thesis is that fast and accurate failure diagnosis for datacenter applications is possible. The above-mentioned challenges can be met with three key innovations.

1.2.1 A Global View

First, rather than taking a component-local view, datacenter operators should take a global view in diagnosing failures. Specifically, this means explicitly modeling component dependencies, collecting global signals that relate application health to component health in addition to component-local signals, and using algorithms that leverage global signals and knowledge of component dependency to assign blame in a fast and accurate manner.

1.2.2 A Penalized-Regression-Based Algorithm

Second, I propose a set of statistical algorithms to localize both fail-stop and various types of gray failures in an accurate, principled and interpretable way. These algorithms leverage global signals and use penalized regression to infer likely failure locations. The use of hypothesis testing on the regression coefficients allows a principled and accurate way to localize gray failures, compared to the use of manually tuned thresholds.

1.2.3 A Predictable Component Architecture

Third, I propose a general design principle that requires component behaviors to be predictable to aide failure localization. In particular, I show how to apply this principle for datacenter network by proposing a predictable network architecture where we require switch routing behavior to be externally predictable by the trusted network-layer software at the endpoint. In contrast to source routing, our architecture allows for imperfect predictions—in-network routing decisions are fine as long as they are predictable and/or infrequently changed. My solution makes effective failure localization and avoidance possible while also allowing for the existence and continuing evolution of in-network features.

1.3 Summary of Contributions

My thesis is that *fast and accurate failure diagnosis for datacenter applications is possible*. I start by demonstrating it in the real-world context of diagnosing virtual hard disk (VHD) failures in IaaS clouds. IaaS cloud platform is one of the most important and largest datacenter applications today. It is a prime example of a datacenter application that is a complex distributed system built on top of modular components, with failures that are challenging to detect and localize.

The main contribution of this dissertation is in demonstrating how the three key innovations proposed in the thesis, namely, *a global view*, *a penalized-regression-based algorithm* and *a predictable component architecture*, make fast and accurate failure diagnosis of datacenter applications possible, in particular in the context of diagnosing VHD failures in IaaS clouds and diagnosing net-

work component faults.

Specifically, we design, implement and evaluate two systems.

- **Deepview**: a system that can localize VHD failures to compute cluster or storage clusters or network tiers. Deepview composes a global view of the IaaS stack and uses an algorithm which integrates Lasso regression and hypothesis testing. Deepview is deployed at Microsoft Azure—one of the largest IaaS providers. I show it can localize VHD failures to compute, storage and network components in an accurate and timely fashion.

- **Volur**: a prototype predictable network. It complements Deepview by proposing a design principle for datacenter network that makes in-network routing predictable to the end-hosts for failure diagnosis and load balancing. In simulation and testbed experiments, Volur is shown to accurately localize non-fail-stop link or switch failures as well as to recover from them quickly

1.4 Organization

The remainder of this dissertation is organized as follows. Chapter 2 reviews background information about datacenter applications and datacenter networks, and then compares and contrasts this work with prior approaches for failure diagnosis. Chapter 3 and Chapter 4 describe the design, implementation and evaluation of Deepview and Volur, and these chapters are primarily based on previously published work [107, 108]. Chapter 5 concludes with limitations and future work.

Chapter 2
Background and Prior Approaches

The past decade saw an increasing trend towards deploying server-side computing in datacenters. Today, a diverse array of applications run in the cloud, ranging from backends to support smartphone location-based services to web-based business software that produces financial reporting for Fortune 500 companies. As a result, systems designers must increasingly address the critical issues of performance, availability and reliability of datacenter applications.

In this chapter, I first provide some background on datacenter applications, using Infrastructure-as-a-Service (IaaS) as an example application, and briefly review common datacenter network architectures. Then, I will survey prior approaches for failure diagnosis of large-scale networked systems, and compare and contrast them with ideas proposed in this dissertation. I will discuss more problem-specific related work in later chapters.

2.1 Background on Datacenter Applications

System designers have largely adopted the component-based architecture instead of the monolithic architecture. IaaS serves as a prime example application of this architecture.

2.1.1 Component-based Software Architecture

The component-based software architecture is now commonplace in today's datacenter. The cloud offers the key benefits of shared physical resources, such as physical space, servers and networks. A datacenter's low-latency network also facilitates the reuse and sharing of software components. Common functionalities are bundled into services for use by multiple tenants. Typical datacenter services include load balancers [43], event delivery [7], configuration management [26], task

scheduling [55, 88], cache [80], databases [40, 36], various types of storage (object/block/disk, warm/cold) [47, 93, 27], and, recently, machine learning models [39, 5]. These services form a hierarchy: some components provide services to others yet are themselves applications atop more basic services. Developers then structure their applications to offload computation to these services.

Services typically expose an Application Programming Interface (API) for client applications to query using Remote Procedure Calls (RPCs). RPCs act like local function calls, but the function logic they access is implemented on remote servers. The datacenter network routes requests to servers performing the actual computations.

2.1.2 Example Application: IaaS

IaaS, one of the most important cloud services today, allows its customers to rent virtual machines configured with customizable computing resources (e.g., CPU/memory, storage and network bandwidth). These resources are acquired and released elastically on-demand. Often, customers pay for what they use, sometimes on a per-minute basis, similar to how we pay for utilities.

The key enabler of IaaS is *virtualization* [17, 101]. While it appears to customers that they get direct and exclusive access to some physical machines, they actually get access to a set of virtualized resources through a narrow interface of virtual machines. Through a layer of indirection provided by hypervisors, cloud providers virtualize their underlying hardware resources and provide them to customers in fungible and resizable units. Unmodified or virtualization-aware guest OSes then run on top of hypervisors.

2.2 Background on Datacenter Network

Datacenter network underpins datacenter applications. Modern datacenter network dictate the following design requirements:

High scalability. Modern datacenters must scale from tens to hundreds of thousands of hosts. A more scalable network lets more applications run in the cloud. It enables different types of applications that are more distributed in nature and coordinate among themselves via service-oriented

architectures.

High bisection bandwidth and uniform bandwidth. To be profitable, cloud providers must achieve high server utilization. One approach to doing so is to require computing resources to be fungible and their assignments to be liquid regardless of their physical locations inside a datacenter. The intent is that any spare computing cycles in some parts of the datacenter can be immediately put to use for applications that require it. Moreover, any applications communicating between different physical servers would not suffer from performance degradations due to network congestion or hotspots. The network is essentially "flat" in the eyes of the applications. To realize this blueprint [50, 79], cloud providers need a network that provides high and uniform bandwidth across servers.

High availability and reliability. Users today expect cloud applications to be available 24/7 and to function reliably. Any downtime can directly lead to revenue loss. Although applications are designed to be somewhat tolerant of outages of underlying components, in most cases, this can be achieved only by highly availability and reliability of all of its constituent services including the underlying network infrastructure.

Low networking cost. Keeping the networking cost low is an important part of cloud providers' monetization strategy of providing the maximum amount of general-purpose computing at the lowest possible cost. The conventional practice is to oversubscribe the network, but this can fail to provide high and uniform bandwidth for modern highly connected applications. Moreover, it becomes increasingly costly and technologically difficult to scale to higher port-counts and higher bandwidth.

How to provide higher scalability, bandwidth, availability and reliability while keeping networking costs low is the core design challenge for modern datacenter networks. Key to meeting these challenges is *scaling out*, rather than scaling up, similar to how large-scale, low-cost computing became possible via commodity PCs. While specifics differ [50, 8, 91], most modern clouds use low-cost commodity switches connected with a Clos topology and simple routing protocols, such as ECMP [95]. We now discuss these common features.

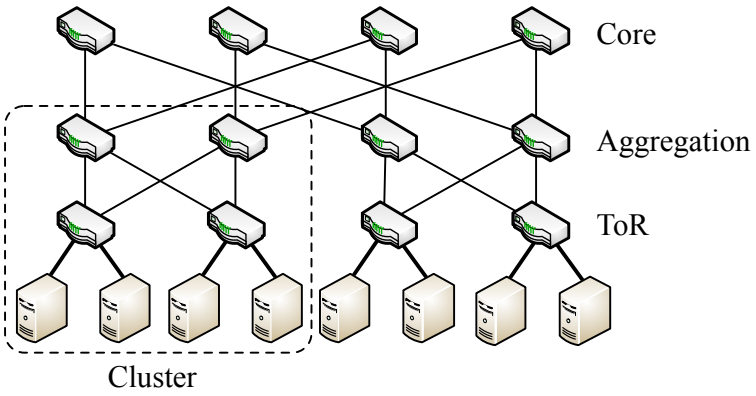


Figure 2.1: A canonical 3-level Folded-Clos topology. The three levels of switches are termed ToR switches, aggregation switches, and core switches.

2.2.1 *Clos Topology*

To connect tens to hundreds of thousands of hosts without needing high-port-count switches while keeping networking costs low, datacenter operators have adapted variants of the design of a Clos network [34, 9, 50, 8, 91] and deployed its key ideas using available commodity hardware.

Figure 2.1 shows the modern incarnation of Clos networks as typically deployed. First, servers are aggregated in racks and connected to a single ToR using a star topology. Singly homed servers in a rack would suffer from disconnectivity if the ToR goes down [72], but this design is simple and easy to wire. There may still be moderate amounts of oversubscription at the ToR to further reduce cost. The core of the network provides the abstraction of one big switch with close to uniform bandwidth between any pair of ingress ports at the edge. This is implemented as a multi-level topology using commodity switches. To provide redundancy and high aggregate bandwidth, each level has multiple switches, and consecutive levels are connected by striping uplinks in a Clos-like fashion.

2.2.2 *ECMP Routing*

Clos topology introduces multiple equal-length network paths between any server pairs. The routing protocol must provide high utilization of all available paths by balancing load appropriately, and the protocol must handle occasional switch failures.

Equal-Cost Multipath (ECMP) routing is a commonly used protocol that attempts to balance load in a transparent fashion, at each switch locally, on a per-flow granularity. When a packet comes in to a switch, the switch calculates a hash value using the packet header, typically the flow 5-tuple, and then maps the hash value to one of many equivalent outgoing ports, based on the current network topology, e.g., the number of uplinks to the next layer of switches. If there is symmetry in the network, all paths would be used equally.

2.3 Background on Failure Diagnosis

Keeping the system up and running is arguably one of the most important goals for a cloud operator. System availability can be quantified by calculating the fraction of uptime during a time span that we are measuring. To emulate the high availability systems from telecommunications, current cloud systems strive to meet five-nines (99.999%) availability standard which allows for a total of 5.26 minutes' downtime in a year.

Systems designers find multiple ways to achieve high availability, for example, by coming up with better large-scale system designs or imposing better engineering practices. However, once a system is up and serving customers, the system operators need to deal with the more short-term concerns of handling failures. When failures occur, operators need to detect their existence, localize to the right subsystem or component, and then mitigate their impacts or repair them.

While failure detection, localization and mitigation/repair are all important parts of running a highly available system. Often, one of them might be the bottleneck. For example, operators may be able to quickly detect and fix failures but are slow to localize the failures to the right component.

2.4 Prior Approaches for Failure Diagnosis

The literature for failure diagnosis of large-scale networked systems is vast. We can classify them by mulitple dimensions, e.g., by *the type of applications* (Internet services [32, 109, 92, 77], datacenter applications [104, 13], IP networks [63, 67], enterprise networks [14, 64, 77], datacenter networks [87, 53, 53, 52]), by *the extent of visiblity and control* (ranging from full transparency of system internals and freedom of instrumentation [104, 87, 52] to complete blackbox and observational roles [75]), or

by *the type of algorithms* (minimum-set-cover tomography [67, 77], statistical anomaly detection [75, 64, 87], classification-based machine learning [32, 13], and Bayesian network [14, 63, 83, 92]).

The general approach from the prior work is to develop a model that describes system behavior, collect health signals from the system either passively or actively, and then apply a set of algorithms that leverage those signals to localize failures in an automated fashion. I borrow this general approach for the failure diagnosis of datacenter applications, but I innovate in all three areas: model, signal and algorithm.

The closest related work to my thesis is the path-based failure management approach proposed by Chen et al. [75]. Like this work, they target large-scale component-based multi-tiered networked system and similarly recognize the shortcomings of a local view of components for failure diagnosis. They even try to tackle non-fail-stop failures. Their approach is three-fold. First, they propose a path-based blackbox model of the system. Second, they instrument the RPC middleware layer and trace requests through the system. Relevant local information (e.g. timestamp, component version) is recorded as the request traverses through the system treating each component as blackbox. Third, they use a classical statistical anomaly detection technique including two-sample test and analysis of variance to distinguish anomalous paths and uses them to infer failure location via either classification-based machine learning or association-based data mining.

In comparison, the global view proposed in my thesis prescribes a more general approach. The path-based measurement proposed by Chen et al. is one particular mechanism to obtain the component interactions as well as to collect signals to relate the health of the system to its components. There are many alternative mechanisms. For example, in Deepview, through operator knowledge, I am able to model how virtual hard disk (VHD) requests from VMs depend on the underlying components of compute, storage and network, without having to explicitly trace a VHD request. As another example, in Volur, tracing every single packet is infeasible. Instead, I rely on the predictability of routing behavior to directly compute the paths based on only packet headers and infrequently changing network state. Rather than emphasizing any particular mechanism (although for datacenter networks, our mechanism is novel), my thesis argues that the right approach to failure diagnosis for large-scale component-based datacenter applications is (i) to consider all components

globally, (ii) to explicitly model their dependencies and interactions, (iii) to collect global signals, using mechanisms appropriate for the problem.

Furthermore, compared to Chen et. al and other prior work, I propose a novel and principled algorithm based on statistical techniques. I use a linear model to consider all paths and all signals in an integrated fashion. This is the first to use a penalized linear model to infer failure locations with hypothesis testing to handle gray failures.

Another class of work that is closely related to my thesis is the use of tomography techniques to localize network failures. In network tomography, end-to-end signals are collected either through active probes or passive measurements, and then a set of heuristics are used to infer the failure locations [30, 67]. The use of end-to-end measurements for failure diagnosis is a mechanism in line with our global view approach. I will show in Deepview evaluation (Section 3.6.2) that our penalized regression algorithm achieves higher localization accuracy than traditional tomography algorithms. The other key differentiator is that past network tomography approaches do not handle the issue of opaque routing in modern datacenter networks. In Volur, I propose a novel predicable network architecture and design a prototype network that makes routing predictable and enables accurate failure localization using tomography or other algorithms.

2.5 Summary

A diverse array of applications run in today's datacenters. Increasingly, these applications are built using a component-based architecture, where cloud providers expose a myriad of software services whose APIs applications can query via RPCs. IaaS is one key datacenter application that benefits from this component-based architecture. Moreover, modern datacenter networks have been designed to meet scalability, availability, reliability and bandwidth requirements needed to support datacenter applications. Finally, while there are many prior work in the area of failure diagnosis, I propose novel ideas in model, signal and algorithm.

Chapter 3
Deepview

This chapter describes a system called Deepview that localizes virtual hard disk (VHD) failures in Infrastructure-as-a-Service (IaaS) clouds. IaaS cloud platform, one of the most important datacenter applications today, is a prime example of a datacenter application that is a complex distributed system built on top of modular components. For this application to meet high availability goals, we show that it is critical to quickly and accurately localize a new type of failure, called *virtual hard disk* (VHD) failures. However, this is in fact very challenging in the presence of multiple components, gray failures, and unpredictable component behaviors. We demonstrate how a global view and penalized-regression-based algorithms can localize both fail-stop and gray failures for VHD failures to the component level in an accurate and speedy manner in real-world deployment.

3.1 Introduction

Infrastructure-as-a-Service (IaaS) is one of the largest cloud services today. Customers rent virtual machines (VMs) hosted in large-scale datacenter instead of managing their own physical servers. Hosted in compute clusters, VMs mount OS and data VHDs from remote storage clusters. Resources can be scaled up or down elastically since compute and storage are separated by design.

Achieving high availability is arguably the most important goal for IaaS. Recently, large-scale system design [38, 76, 58], failure detection and mitigation techniques [49, 104, 52, 14, 13, 64, 87], and better engineering practices [20] helped operators improve cloud system availability. Yet, attaining the standard of five-nines (99.999%) VM availability remains a challenge [69, 23].

Microsoft Azure experience on the order of thousands of VM down events daily. The biggest category of down events (52%) comes from VHD failures. Due to compute-storage separation, when

a VM cannot access its remote VHDs, the hypervisor crashes the VM, causing a VHD failure. Such VHD failures are caused by various failures in the IaaS stack and constitute the biggest obstacle towards achieving five-nines availability for our IaaS [1].

Compute-storage separation brings unique challenges to locating VHD failures. First, it is hard to find in a timely fashion the failing component from the large number of interconnected compute, storage, and network components. The current practice of monitoring individual components is not sufficient. Due to complex dependencies and interactions among components in our IaaS, a single root cause can have multiple symptoms at different places: network or storage failure may cascade through many other component, affecting many VMs and applications. It becomes hard to distinguish causes from effects, resulting in a lengthy troubleshooting process as reported incidents bounce back and forth across different teams.

Second, many component failures in the IaaS stack are gray in nature and hard to detect [58]. For failures such as intermittent packet drops and storage performance degradation, some VHD requests that pass through the component can fail, but others may not. Depending on the length and severity, some of these can cause VHD problems while others only cause performance issues. In these case, failure signals may be weak or sporadic in time and space, complicating fast and accurate detection.

To address these challenges, we designed and deployed a system called Deepview. Deepview takes a global view: it gathers as input both VHD failure events and VHD paths between VMs and their storage to construct a model that connects compute, storage, and network components. We further introduce an algorithm that integrates Lasso regression [96] and hypothesis testing [29] for principled and accurate failure localization.

We implement the Deepview system for near-real-time VHD failure localization atop a high-performance log analytics engine. To meet the requirement to localize failurs in near-real-time, we add streaming support to the engine. Our implementation can run the Deepview failure localization algorithm in seconds, at a scale of thousands of compute and storage clusters, tens of thousands of

[1] Azure had 34 regions with 99.9979% uptime in 2016. [100]

network switches, and millions of servers and VMs.

Currently deployed at Azure, Deepview helped us identify many new VHD failure root causes that were previously unknown, such as gray storage cluster failures and unplanned Top of Rack switch (ToR) reboot. Using Deepview, unclassified VHD failure events dropped from several thousands per day to less than 500, and the Time to Detection (TTD) for incidents was reduced from tens of minutes and sometimes hours to under 10 minutes.

Contributions. We identify VHD failures as the greatest obstacle to five-nines VM availability for our IaaS cloud, and we propose a system to quickly detect and localize them. In particular, we:

- Introduce a global-view-based algorithm that accurately localizes VHD failures, even for gray failures
- Build and deploy a near-real-time system that localizes VHD failures in a timely manner
- Quantify the implications of key IaaS architectural design decisions, including ToR as a single-point-of-failure and compute-storage separation.

3.2 Background and Motivation

This section first provides background on Azure's IaaS architecture. We explain how compute-storage separation can result in a new type of failure—VHD failures. Then, we introduce the state-of-the-art industry practice for localizing VHD failures and explain its drawbacks in terms of speed and accuracy. Finally, we motivate the approach Deepview takes and explain the challenges for putting the system into production use.

3.2.1 Compute-Storage Separation in IaaS

Figure 3.1 shows Azure's IaaS architecture. A similar architecture seems to be used at Amazon for instances backed by the elastic block store [1]. Every VM has one OS VHD and one or more data VHDs attached. One key design decision is to separate compute and storage physically—**VMs and their VHDs are provisioned from different physical clusters**.

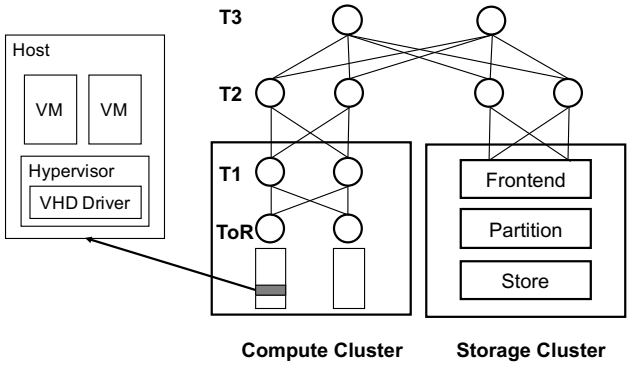


Figure 3.1: Azure's IaaS architecture. A region has tens to hundreds of compute/storage clusters. Each Tier2 (T2) switch connects some subset of clusters, while Tier3 (T3) switches connect the T2 switches. T3 switches are connected by inter-region network (not drawn).

The main benefit of this separation is to keep customer data available when their VMs become unavailable, e.g., due to a localized power failure. As a result, VM migration becomes easy as we only need to create a new VM (possibly on a different host or cluster) and attach the same VHDs.

In our datacenters, VHDs are provisioned and served from a highly available, distributed storage service [27, 45]. Azure's storage service is deployed in self-contained units of clusters with their own Clos-like network [8, 50, 27], software load balancers, frontend machines and disk/SSD-equipped servers. Similarly, VMs are hosted on physical servers grouped in what we call compute clusters. Each metro region typically has tens to hundreds of compute clusters and storage clusters, interconnected by a datacenter network.

Another benefit is load-balancing. A VM in a compute cluster can remotely mount VHDs from many different storage clusters. A compute cluster therefore uses VHDs from multiple storage clusters, and a storage cluster can serve many VMs from different compute clusters. As we will see later in section 3.3, this many-to-many relationship is leveraged by Deepview.

VHD Access is Remote. Compute-storage separation requires all VMs to access their VHDs over the network. When a VM accesses its disks, it is unaware that they are remotely mounted. The VHD driver in the host hypervisor provides the needed disk virtualization layer. The driver intercepts VM disk accesses, and turns them into VHD remote procedure call (RPC) requests to the remote stor-

VHD Failure	SW Failure	HW Failure	Unknown
52%	41%	6%	1%

Table 3.1: Breakdown of the causes of VM downtime. VHD failures cause the majority of VM downtime.

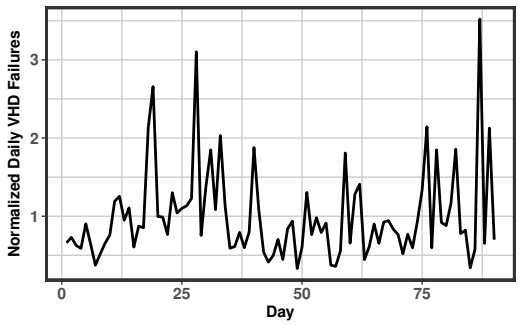


Figure 3.2: Daily VHD failures normalized by the 3-month average. Every day had at least one failure. On the worst day there were 3.5x more failures than average.

age service. The VHD requests and responses traverse over multiple system components (e.g., the VHD driver and the remote storage stack) and through multiple network hops (e.g., ToR/T1/T2/T3 switches).

3.2.2 A New Type of Failure: VHD Failure

Compute-storage separation causes a new type of failure. In our datacenter, whenever VHD accesses are too slow to respond (the default timeout is 2 minutes), the hypervisor crashes the guest OS. In order to protect data integrity, when VHDs do not respond, the guest OS must be paused. But the pause cannot be indefinite—an unresponsive VM can cause customers to fail their own application level SLAs. After some wait, one reasonable option is to surface the underlying VHD access failure to the customer by crashing the guest OS. We call this **VHD failure caused by Compute-Storage Separation** or VHD failure for short.

VHD failure is the biggest cause of unplanned VM downtime. We analyzed an entire year's of

IaaS VM down events, including their durations and causes (an internal team finds root causes for VM down events). Table 3.1 shows that 52% of VM downtime is due to VHD Failures, 41% due to Software Failures (data-plane software and host OS), and 6% due to Hardware Failures, and 1% due to unknown causes.

Figure 3.2 further shows the daily number of VHD failures normalized by the 3-month average across tens of regions. VHD failures happen daily. Occassionally, they are particularly numerous. The worst day over the 3-month period saw a 3.5x spike in volume.

To minimize the impact of VHD failures and improve VM availability, the most direct approach is to quickly localize and mitigate these failures. Next, we explain the prior VHD failure handling approach and its drawbacks.

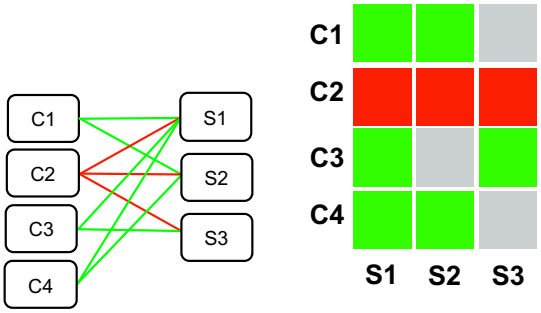
3.2.3 State-of-the-Art: Component View

Our datacenter operators prioritize by the impact of each incident. A large rise in VHD failure events would automatically trigger incident tickets and set off an investigation.

The site reliability engineers (SREs) look at system components individually and locally, to see if any local component anomaly coincides in time with the VHD failure incident. The Compute team might look for missed heartbeats to see if the impacted physical machines have failed. The Storage team might look at performance counters to see if the storage system is experiencing an overload. The Network team might look at network latency and link packet discard rates to determine if some network devices/links could be at fault. Once the failure location is confirmed, the responsible team often has standard procedures for quick mitigation.

Prior to Deepview, failure localization was slow. It was common that Azure needed tens of minutes, sometimes more than one hour, to localize and mitigate big incidents, and hours to tens of hours to detect and localize gray failures. When a big incident happened, often more than one component had an anomaly because a single root cause could cascade to other services. For example, one big network incident caused as many as 363 related incidents from different services! As a result, the incident ticket would sometimes get ping-ponged among the teams.

Further, localization for gray failures [58] was often inaccurate and slow. For example, while



(a) Bipartite model (b) Matrix view

Figure 3.3: The bipartite model and the corresponding matrix view of a downtime event.

we know ToR uplink packet discards can cause VHD requests to fail, it was unclear how severe the discard rate has to be. Setting a threshold to catch those failures became an art: too low generated too many false positives, while too high delayed diagnosis or missed the issue.

3.3 Our Approach: Global View

Our key insight is that rather than looking at the components individually and locally, we should take a global view. The intuition can be illustrated by the bipartite model in Figure 3.3a. In this model, we put compute clusters on the left side and storage clusters on the right. We draw an edge from a compute cluster to a storage cluster if it has VMs that mount VHDs from the storage cluster. We also assign an edge weight equal to the fraction of VMs that have experienced VHD failures.

For a compute cluster issue such as an unplanned ToR reboot that causes all VMs under the ToR to crash regardless of what storage clusters they use, we see the edges (highlighted in red) from the impacted compute cluster with high VHD failure rates, as in Figure 3.3a. When a storage cluster fails, causing all VMs using that storage cluster to experience VHD failures, we see edges with high VHD failure rates coming to the impacted storage cluster.

If we put the compute clusters along the y-axis and the storage clusters along the x-axis, we get a matrix view as shown in Figure 3.3b. In this matrix view, a horizontal pattern points the incident to the computer cluster, while a vertical pattern points to the storage cluster.

Challenges. Though the bipartite model looks intuitive and promising, there are several challenges to use that insight in a production setting. First, since the bipartite model cannot be easily extended to model the multi-tier network layers, we cannot use it to diagnose failures in the network. Second, while we can use some voting/scoring heuristics to automate the visual pattern recognition, they work well only when the failures are fail-stop. For gray failures [58], fewer VMs would crash so the VHD failure signals are often weaker, and the VHD failure patterns are less clear cut. Third, when big incidents happen, many customers may feel the impact, making timely failure localization imperative. Our system must therefore operate in near-real-time.

Problem Statement. Our goal is to localize VHD failures for both fail-stop and gray failures to component failures in compute, storage or network, at the finest granularity possible (clusters, ToRs and network tiers), all within a TTD target of 15 minutes, in line with our availability objectives.

3.4 Deepview Algorithm

In this section, we explain how the Deepview algorithm solves the first two challenges—handling network and gray failures. We first describe our new model, a generalization of the bipartite model to include network devices. Then, we introduce our inference algorithm with two main techniques: 1) **Lasso regression** [96] to select a small subset of components as candidates to blame; 2) **hypothesis testing** [29] as a principled and interpretable way to handle strong and weak signals and decide on the components to flag to operators. There are other failure localization algorithms that can be adapted for our problem. We compare Deepview with them in Section 3.6.2, and show that our approach has better recall and precision.

3.4.1 Model

In Section 3.3, we introduced a bipartite model that takes a global view of compute and storage clusters. Here we generalize the model to include network devices.

In this new model, we have three types of components: compute clusters, network devices and storage clusters. Figure 3.1 shows that compute clusters and storage clusters are interconnected by

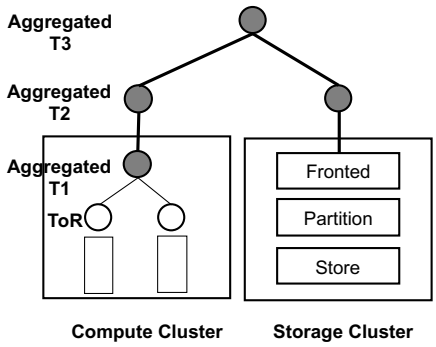


Figure 3.4: Transforming the Clos network to a tree. Not shown: each aggregated T2 switch connects to many compute/storage clusters and each aggregated T3 switch connects to many aggregated T2 switches.

a number of Tier-2 (T2 for short) and Tier-3 (T3) switches in a Clos topology. ToR and Tier-1 (T1) switches are within the clusters, and are part of the clusters. To model the network, we replace each edge in the bipartite model with a path through the network that connects a compute cluster to a storage cluster. Here we describe the model at the level of clusters (which we call Cluster View in Section 3.6). We have also extended the model to the granularity of ToRs inside compute clusters (ToR View). For this work, we keep the storage cluster as a blackbox due to its complexity. As future work, we plan to apply our approach to the host level and the storage clusters internals.

3.4.1.1 Simplify the Clos Network to a Tree

One complication in modeling the network is that each compute/storage cluster pair is connected by many paths. Due to Equal-Cost Multi-path (ECMP) routing [50, 95], we do not know precisely which path a VHD request takes, and therefore, we do not know which path to blame when the request fails.

Our solution is to transform the Clos topology (Figure 3.1) to a tree topology (Figure 3.4) so that there is a unique shortest path between each cluster pair. We start from the bottom and go up for each cluster and aggregate the network devices by tiers, and then use shortest path routing to find the lowest overlap between each cluster pair.

The detailed procedure is as follows. First, we start with ToR switches in a cluster and find the

T1 switches they connect to. Then, we group those T1 switches as an aggregated T1 group for that cluster. Similarly, we can find the connected T2 switches for those T1 switches and group them as an aggregated T2 group for that cluster. We repeat this procedure to find the aggregated T3 groups. At the end of the aggregation, we have determined the aggregated T1, T2, T3 groups for each cluster in a region. The next step is to find the shortest path for each compute-storage pair . If the aggregated T2 groups of a cluster pair overlap, the midpoint is that overlapped aggregated T2 group; if their aggregated T2 groups do not overlap but their aggregated T3 groups do, the midpoint is the T3 group.

Due to the simplification, we cannot pinpoint to a specific network device, but only to within a network tier. In practice, Deepview is mainly used to decide which SRE teams to notify when VMs crash. Upon notification, network teams have other tools (e.g. Traceroute) to further narrow down to a device for mitigation.

3.4.1.2 From Paths to Components

Next, we use our observations of VHD failure occurrences to pinpoint which component has failed. We assume that components fail independently, which is a practical and reasonable approximation of the real world. For example, a compute cluster failure is unlikely to be correlated with a storage cluster failure. We can write down a simple probabilistic equation for a path consisting of compute, storage and network components:

$$\mathbb{P}(\text{path i is fine}) = \prod_{j \in \text{path}(i)} \mathbb{P}\left(\text{component j is fine}\right) \tag{3.1}$$

We approximate 1 – $\mathbb{P}$(path i is fine) using the rate of VHD failures observed for that path:

$$\frac{n_i - e_i}{n_i} \approx \prod_{j \in \text{path}(i)} p_j \tag{3.2}$$

where n_i is the total number of VMs, e_i is the number of VMs that have VHD failures for a given time window, and p_j is the probability that component j is fine. We get a system of equations by

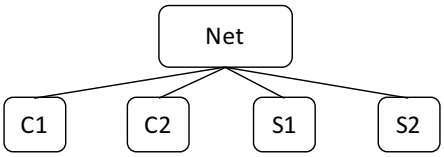


Figure 3.5: Example where multiple solutions may exist.

writing down (3.2) for every path. Next, we infer the values of p_j for all components.

We know there is noise in our measurement, so we cannot directly solve the system of equations and would need to explicitly model the noise. Specifically, after taking log on both sides of equation (3.2) and adding a noise term ε_i, we get a set of linear models:

$$y_i = \sum_{j=1}^{N} \beta_j x_{ij} + \varepsilon_i, \qquad \varepsilon_i \overset{i.i.d.}{\sim} N(0, \sigma^2) \tag{3.3}$$

where $y_i = \log\left(\frac{n_i - e_i}{n_i}\right)$, $\beta_j = \log p_j$, and the binary variable $x_{ij} = 1$ iff i-th path goes through the j-th component.

Interpretation of β_j: Once we get estimates for β_j, the probability that component j is fine can be computed from β_j because $p_j := \exp(\beta_j)$. If β_j is close to 0, we can clear component j from blame. Otherwise, if β_j is unusually negative, we have strong evidence to blame component j (see Section 3.4.3). We would ensure $\beta \leq 0$.

Next, we answer the following two questions: (1) how to get fast, accurate, and interpretable estimates for β_j; (2) given the estimates, how to decide which component to blame in a principled and interpretable manner?

3.4.2 Prefer Simpler Explanation

In practice, the number of unknown variables (β's) can be larger than the number of equations. We illustrate this in a simple example shown in Figure 3.5. We can list 4 equations with 5 free variables

(the βs):

$$
\begin{aligned}
y_1 &= \beta_{c1} + \beta_{net} + \beta_{s1} + \varepsilon_1 \\
y_2 &= \beta_{c1} + \beta_{net} + \beta_{s2} + \varepsilon_2 \\
y_3 &= \beta_{c2} + \beta_{net} + \beta_{s1} + \varepsilon_3 \\
y_4 &= \beta_{c2} + \beta_{net} + \beta_{s2} + \varepsilon_4.
\end{aligned} \tag{3.4}
$$

Suppose all four paths saw equal probability of VHD failures. The blame can be pushed to the compute clusters C1 and C2, or the storage clusters S1 and S2, or the network, or a mix of those. Traditional least-square regression cannot give a solution in this case. But our experience tells us that multiple simultaneous failures are rare for a short window of time (e.g., 1 hour) because individual incidents are rare and failures are (mostly) independent. How do we encode this domain knowledge into our model to help us identify the most likely solution?

To prefer a small number of failures is mathematically equivalent to prefer the estimates $\beta = (\beta_1, \cdots, \beta_N)$ to be sparse (mostly zeros). We express this preference by imposing a constraint on model parameters β. By asking the sum of absolute values of β, i.e., $\|\beta\|_1$ to be small, we can force most of the components of β to zero, leaving only a small number of components of β remaining. This technique of adding a L1-norm constraint is known as Lasso [96], a computationally efficient technique widely used when sparse solutions are needed. We also ensure $\beta \leq 0$ to get valid probabilities. The estimate procedure that encodes all our beliefs in our model is thus the following convex program,

$$
\hat{\beta} = \underset{\beta \in \mathbb{R}^N, \beta \leq \mathbf{0}}{\arg\min} \|\mathbf{y} - \mathbf{X}\beta\|_2^2 + \lambda \|\beta\|_1 . \tag{3.5}
$$

Simplicity vs. Goodness-of-fit via λ: This loss function tries to strike a balance between goodness-of-fit in the first term (i.e., how well the model explains the observation) and sparsity in the second term (i.e., fewer failing components are more likely). The regularization parameter λ is the knob. Larger λ prefers fewer components to be blamed at potentially worse goodness-of-fit. The optimal

value of λ is set by an automatic (data-adaptive) cross-validation procedure [54].

3.4.3 Decide Who To Blame

While big incidents are relatively easy to localize with a fixed threshold, it is much harder to find a threshold that can discriminate gray failures from normal components when there are random measurement errors. The estimated failure probabilities for gray failures can be very close to zero (see Section 3.6.1). The challenge then becomes how big an estimated failure probability is for a gray failure versus just measurement error. Setting such a threshold manually requires laborious data-fitting and is often based on some vague notions of anomaly. In practice, it can be difficult and fragile.

Can we use data to find the decision threshold in a principled and automatic way? Intuitively, the larger the magnitude a (negative) Lasso estimate has, the higher its estimated failure probability, and correspondingly more likely the component has failed. We had a painful experience manually tuning the threshold, but the process gave us some experience in distinguishing true failures (big incidents and gray failures) from measurement noise. We found that if a component's Lasso estimate is much worse than the average, then it is likely a real failure and should be flagged. The further from average, the more confident we are that the component has failed.

This decision can be automated in a hypothesis testing framework. Consider the following one-sided test:

$$H_o(j) : \beta_j = \bar{\beta} \quad \text{v.s.} \quad H_A(j) : \beta_j < \bar{\beta} \tag{3.6}$$

The null hypothesis $H_o(j)$ says the true probability that component j is fine is no different from the grand average of all components. We then use the data to tell us if we can reject $H_o(j)$ or not. If the data allow us to reject $H_o(j)$ in favor of the alternative hypothesis $H_A(j)$, then we can blame component j. Otherwise, we do not blame component j. The hypothesis test has three steps.

Step 1: Compute Test Statistic. Given Lasso estimates for components in a region, we find the mean

$\bar{\hat{\beta}}$ and standard deviation $\sigma_{\hat{\beta}}$. Then we compute a modified Z-score for each component j,

$$z_j = \frac{\hat{\beta}_j - \bar{\hat{\beta}}}{\sigma_{\hat{\beta}}/\sqrt{N}}. \tag{3.7}$$

Under the assumptions that the measurement error is Gaussian, and other caveats,[2] we approximate the distribution of z_j as a Gaussian distribution with mean zero (under $H_o(j)$) and certain variance.

Step 2: Compute p-value. We then compute the p-value [29] for each component j. The p-value is the probability of seeing a failure probability for component j as extreme as currently observed simply by chance assuming that it is no different from the average. If the p-value is really small, then we do not believe the failure probability for component j is just about average. See the Appendix for more discussion on p-value.

Step 3: Make a Decision. Finally, we apply a standard threshold of 1% on p-value.[3] It expresses our tolerance for false positive rate. For example, if the p-value for component j is less 1%, we blame the component with at most 1% false positive rate. Otherwise, we have insufficient evidence to blame component j.

Avoid the Pitfalls in Multiple Testing. We test every component in a region and flag them based on p-values. For every test, we may falsely blame a normal component with a small chance. But with a large number of components in every region, we are bound to commit an actual false positive if not careful. This is called the multiple testing problem. We use the Benjamini-Hochberg procedure [19] to control the False Discovery Rate. See Appendix for details.

3.5 Deepview Design and Implementation

We have two main system requirements:

[2] Testing on Lasso estimates is an active research area. We fit a Lasso model to obtain a set of nonzero variables, and refit these variables with least squares. See [110].

[3] Another common threshold is 5%, but it generates too many false positives for testing multiple hypotheses in our setting. See Appendix.

- **Near-real-time (NRT) processing**: VHD failures result in customer VM downtime, so failure localization must be speedy and accurate. We have the requirement that the time-to-detection (TTD) be within 15 minutes.

- **Speedy iteration**: VHD failures are the biggest obstacle to higher VM availability, so there is an immediate need by the operations team for better diagnosis. Our system is designed for quick iteration.

Our system requires two types of input data: non-real-time structural data and real-time event data. The former include the compute and storage clusters information, all the VMs and their VHD storage account information and related context, the paths for all the compute-storage pairs, and the network topology. Taking periodic snapshots of those every few hours suffices for our purposes. The latter are the VHD failure signals from servers. To meet near-real-time requirements, our algorithm needs to see VHD failure signals within minutes, ideally through a streaming system.

We need to scale to thousands of compute and storage clusters, tens of thousands of VHD failures per day, tens of thousands of network switches, hundreds of thousands paths, and millions of VMs.

The non-real-time information is either already in our in-house log analytics engine called Azure Kusto [3, 2], or can be generated and ingested into Kusto. Kusto stores data as tables but the tables are append-only, and it supports a SQL-like declarative query language. Kusto is backed by reliable persistent storage from a distributed storage service, using memory and SSD for a read-only cache. By default, it builds indices for all columns to improve query speed.

VHD failure events are generated by hypervisors. They are collected by a real-time pipeline. Since most of our data is already in Kusto, and Kusto provides highly expressive declarative language and fast data analysis, we ingest the VHD failure events into Kusto and build Deepview system on top of it.

System Architecture: The resulting system architecture is shown in Figure 3.6. It has four components. The real-time path and non-real-time path are for the input data ingestion for the Deepview algorithm. The Kusto platform provides both data analysis and storage for input, intermediate, and

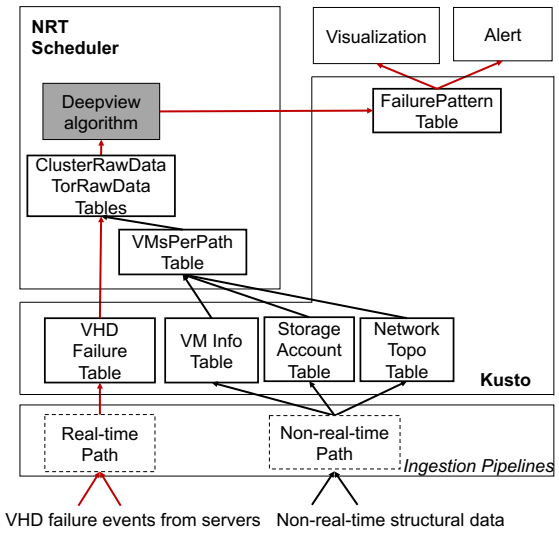


Figure 3.6: Deepview system architecture. Data schema is given in Table 3.2.

Table Name	Schema
VHDFailure	(ts, vm_id, vhd, str_account)
VMInfo	(ts, vm_id, comp_cluster, tor)
StorageAccount	(ts, str_account, str_cluster)
NetworkTopo	(ts, cluster, tor_list, t1_list, t2_list, t3_list)
VMsPerPath	(tstart, tend, num_vms)
ClusterRawData	(tstart, tend, comp_cluster, str_cluster, num_vms, num_failed_vms)
TorRawData	(tstart, tend, comp_cluser, tor, str_cluster, num_vms, num_failed_vms)
FailurePattern	(tstart, tend, region, type, loc, pval, visual_url)

Table 3.2: Kusto schemas for the Deepview data.

output data. The visualization and alert are tools for the consumption of Deepview results. The NRT scheduler is what we build on top of Kusto to support stream processing for the Deepview algorithm.

3.5.1 Stream Processing

We build our own stream processing system on top of Kusto because most of our data are already there; a few additions to satisfy our needs. We do not claim novelty compared to existing research and commercial streaming systems [105, 16, 4].

To support stream processing on top of Kusto tables, we use two abstractions:

- A computation directed-acyclic-graph (DAG) declared as a set of SQL-like queries with their output tables.

- A scheduler that runs each query at a given frequency.

We store the DAG and its scheduling policy as tables, since tables are Kusto's only supported data structure.

Computation DAG. The computation DAG consists of a set of queries that read from input tables and produce one or multiple output tables. The queries are the "edges" and the input/output tables are the "nodes". To maintain the DAG in Kusto, we give each query a name and store the query definition and the query output table name in yet another table.

NRT Scheduler. To provide a streaming window abstraction, we use a schedule to describe when each query in the DAG should be executed. The schedule describes how often it should run and how many times to retry. To meet availability requirements, we use a one-hour sliding window that moves forward every 5 minutes.

3.5.2 Algorithm Implementation

The algorithm implementation has three parts: first, construct the model—instantiate the design matrix x_{ij} and observation y_i based on the Deepview raw data tables, then run Lasso regression to infer β, and finally carry out hypothesis testing to pinpoint the failures.

Sparse Matrix and Region Filtering. The scale of our data poses some challenges for algorithm running time and memory footprint. Constructing a full design matrix requires filling in entries for every path and every component with either zero or one. This can be slow and has high memory usage. However, x_{ij} are mostly zeros since each path has at most tens of components, so we only need to store the non-zero entries. Another simple technique is to only get data from Kusto for regions with non-zero VHD failure occurrences. Since simultaneous failures are rare, region filtering can avoid running the algorithm for some regions without hurting accuracy.

Coordinate Descent. Lasso regression has no closed form solution. Coordinate descent [46] is

one of the fastest algorithms to solve the Lasso regression. We minimize the loss function as in Equation 3.5 with respect to each coordinate β_j while holding all others constant. We cycle among the coordinates until all coefficients stabilize. In practice, with warm start, we found that coordinate descent almost always converges in only a few rounds.

Cross-Validation with Warm Start to Set λ. We set the regularization parameter λ for Lasso using a data-adaptive method, i.e., cross-validation [54]. We use 5-fold cross validation where we split the data by paths into 5 partitions, and use any four of them to fit β for a given choice of λ and then compute the mean squared error (MSE) on the holdout partition using the fitted β. The optimal λ is the one that minimizes the average MSE. We speed up cross validation using a warm start technique [46]. Recall that a larger λ meant fewer non-zero β_j. We start with the smallest λ that turns off all β_j, and then we gradually decrease λ. Since β tends to change only slightly for a small change in λ, we reduce the number of rounds for coordinate descent by reusing $\beta(\lambda_{k-1})$ as the initial values for $\beta(\lambda_k)$.

3.6 Evaluation

We have deployed Deepview in production at Azure. Here, we first evaluate how well Deepview localizes VHD failures using production case studies. Then, we compare Deepview's accuracy with other algorithms. Next, we analyze various techniques proposed for Deepview and ask how useful each is. Finally, we evaluate how Deepview's runtime efficiency.

3.6.1 Deepview Case Studies

In this subsection, we ask how effective Deepview is at detecting and diagnosing incidents in production use.

3.6.1.1 Statistics

We examined the Deepview results for one month. The number of VHD failures generated per day can be up to tens of thousands. For this month, Deepview detected 100 patterns, and reduced the number of unclassified VHD failure events to less than 500 per day. We also tried to associate the

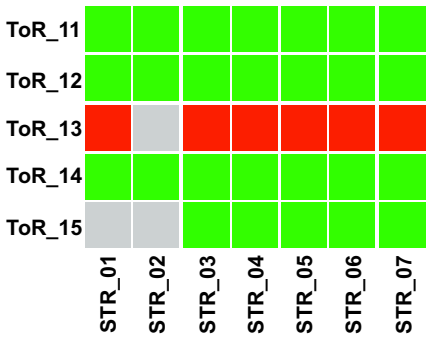


Figure 3.7: Deepview pattern for an unplanned ToR reboot.

detected patterns with incident tickets: 70 of the patterns were directly associated with incident tickets. The other 30 patterns were not associated with tickets. These 30 patterns turned out to be generated by weak VHD failure signals. They were all real underlying component failures that escaped the previous alerting system, either because of their smaller impact (e.g., unplanned ToR reboot) or their gray failure nature (e.g., gray storage failure).

Next we examine some of the representative patterns we found and discuss the insight we learn from them.

3.6.1.2 Unplanned ToR Reboot

From time to time, ToRs undergo scheduled downtime for firmware upgrade or other maintenance operations. Impacted customers are notified in advance, with their VMs safely migrated to other places. However, occasionally, a ToR may experience an unplanned reboot due to a hardware or software bug. Since each server connects to only one ToR, the VMs under the ToR will not be able to access their VHDs. We get VHD failures as a result. To detect unplanned ToR reboots, Deepview first estimates the failure probability and p-value for the ToR, and then checks the following conditions for confirmation: all the VMs under the ToR get VHD failures, the ToR OS boot time matches the failure time detected by Deepview, and the neighboring ToRs are working fine.

Figure 3.7 shows one such unplanned ToR reboot detected by Deepview in a small region.[4] It

[4]Readers may wonder how VHD failure events can be identified when the ToRs are single point of failure. They

shows a portion of the Deepview UI, which we call ToR view. It clearly shows a horizontal pattern. The ToR switches in the compute cluster are listed on the y-axis and the storage clusters are listed on the x-axis. Each cell in the figure shows the status of the ToR and storage cluster pair. Gray means the VMs under the ToR do not use the corresponding storage cluster; green means the VMs do not have VHD failures; red means the VMs are experiencing VHD failures.

Deepview blamed the right ToR among 288 components in the region (ToRs, T1/T2/T3 switch groups and compute/storage clusters). Deepview estimated the failure probability for the failed ToR to be 100% with a p-value of 1.84E−64, which is much less than 0.01.

Deepview therefore makes it possible to study how often ToRs cause downtime. We discuss this in detail in Section 3.7.1.

3.6.1.3 Storage Cluster Gray Failure

Our storage cluster runs a full storage stack including load balancer/frontend, meta-data management, storage layer, etc. VHD failures can happen due to a variety of failure modes in the storage stack. When storage cluster failures are non-fail-stop, the VHD signals can be weak and noisy. For example, the load balancer could discard VHD requests to shed load, and in other cases, software bugs could cause some VHDs to become unavailable, impacting only a subset of VMs.

We next discuss such a storage gray failure case. A new storage cluster was brought online, but with a misconfiguration that allowed a test feature in the caching subsystem to be enabled. This bug mistakenly put some VHDs in negative cache (denoting deletion), rendering them "invisible" and unavailable for VM access.

Based on the VHD failure events at hour 0 in Figure 3.8, Deepview found three non-zero failure probability entities in the region, 0.34 for storage cluster S0, 0.002 and 0.047 for compute clusters C0 and C1. Notice that because this storage cluster failure only affected a small number of VMs, we did not get a failure probability of 1 for S0. Further, the two compute clusters saw non-zero failure probabilities because they also saw VHD failure events. However, despite the weak signal,

---

are in fact stored locally in the servers and are retrieved once network connectivity is restored (typically within 10 minutes for software failures).

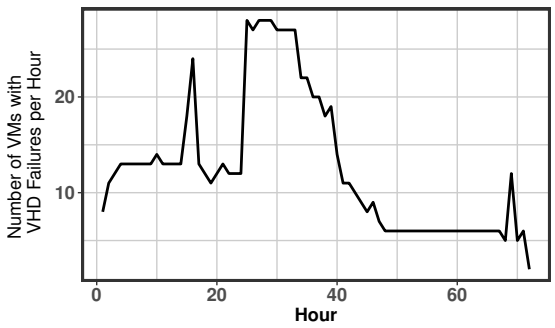


Figure 3.8: The number of VMs with VHD failures per hour during a storage cluster gray failure.

our algorithm was able to correctly pinpoint the failure to S0. Our hypothesis testing procedure computed a p-value of 3.9E−34 for S0, identifying it as a failed cluster with very high confidence. On the other hand, C0 and C1 had p-values 0.51 and 0.54, respectively, and signifying a lack of evidence. Using our prior threshold method for detection, we would have delayed the detection by 22 hours. As shown in Figure 3.8, the signal is weak: the number of VMs affected per hour in the beginning was only around 10, and the peak number was only 28.

3.6.1.4 Network Failure

In our datacenter, switches other than ToRs have replicas. Single switch failures thus seldom lead to wide impact outages. However, in rare cases, a combination of capacity loss and traffic surge can cause network failures.

In one region, we have over 100 compute clusters and 50 storage clusters. They are connected by four T2 aggregated switches (numbered T2_0 to T2_3) with a T3 aggregated switch (T3_0) on top, as annotated along the axes in Figure 3.9. Each aggregated switch contains multiple switches. One day, a T3_0 switch underwent a major maintenance event, which triggered some T2 switches in T2_0 to mistakenly detect Frame Check Sequence (FCS) errors on the links to T3_0. Our automatic network service then kicked in and shut down most of links between the T3_0 switch and T2_0 except for three links saved by a built-in safety mechanism.

This loss in capacity together with a surge in storage replication traffic caused significant con-

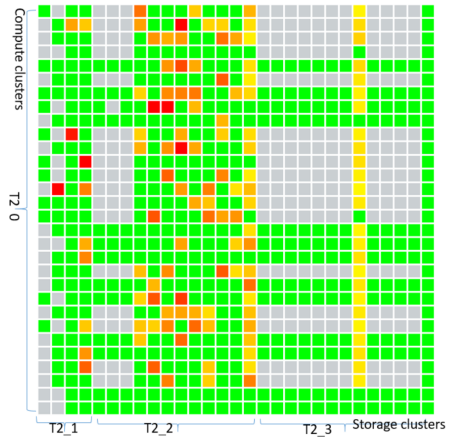


Figure 3.9: Deepview pattern for a network incident.

gestion between T2_0 and T3_0. As a consequence, we saw a significant increase in VHD failures experienced by customer VMs.

Figure 3.9 shows the pattern in the Deepview UI (partial Cluster View) with compute clusters on the y-axis and storage clusters on the x-axis. The switch aggregated per cluster is annotated on each axis. Yellow cells have a VHD failure rate at most 5%. The VHD failure rates are moderate because the VHD failures in this case were caused by network congestion; most of the time network connectivity was still working.

Deepview identified three aggregated switches with non-zero failure probabilities: 0.21%, 0.11%, 0.03% for T3_0, T2_0, and T2_2, respectively. Their corresponding p-values are 9.91E−12, 4.25E−04, 0.221. We point to T3_0 and T2_0 as the faulty network layers. The failure location is correct, as the root cause is the link congestion between these two network layers. We note Deepview gave small failure probabilities because the VHD failure signals are weak: only a very small percentage of affected VMs crashed. But since T3_0 and T2_0 are high in the network hierarchy, they impact a large number of VMs.

We also experienced network incidents where network connectivity for many VMs were lost. They were easy for Deepview to detect and localize, as the signals were strong: many VMs died at the same time. We present this gray failure case to show the strength of Deepview.

To summarize, we have shown that Deepview can localize various incidents in which the signals

can be weak or strong. Deepview has also deepened our understanding of VHD failures by identifying various patterns including horizontal patterns caused by incidents including unplanned ToR reboot, vertical patterns caused by storage outages, and network failure patterns.

3.6.2 Algorithm Comparison

Several algorithms that have been previously used to localize failures in the network can be extended to localize VHD failures. We compare with two tomography algorithms and a Bayesian network algorithm:

- **Boolean-Tomo [42, 41]:** Classify paths into good and bad paths based on a threshold (bad if at least γ VHD failures). Iteratively find the component on the largest number of unexplained bad paths, as the top suspect until all bad paths are explained. For the threshold γ, we tried γ = 1, 2, 3, 4, 5, and picked γ = 1 to maximize its recall and then precision.

- **SCORE [67]:** Classify paths into good and bad paths based on a threshold (γ). Iteratively compute for each component its hit ratio $\frac{numBadPaths(c)}{numPaths(c)}$ and coverage ratio $\frac{numUnexplainedBadPaths(c)}{totalNumUnexplainedBadPaths}$. Only consider components above a hit ratio threshold (η). Take the component with the highest coverage ratio as the top suspect. For the threshold γ and η, we tried $\gamma = 1, 2, 3, 4, 5$ and η = 0.001, 0.01, 0.1, and picked γ = 1 and η = 0.01 to maximize its recall and then precision.

- **Approximate Bayesian Network [83]:** The runtime to compute exact Bayesian network is exponential in the number of components, and thus is infeasible for us. We tried an approximation [83]. It uses mean-field variational inference to approximate the Bayesian network with a Noisy-OR model, and estimates the component j's failure rate as the posterior mean of a Beta distribution $B(\alpha_j, \beta_j)$. A component is blamed if $\frac{\hat{\alpha}_j}{\hat{\alpha}_j+\hat{\beta}_j}$ is above certain threshold. We do not include its accuracy numbers, because we are unable to make it give meaningful results on our data. The estimated posterior means of component failure rate allows us to apply a threshold. The computation takes 10 minutes for a single region, so this approach is not fast enough for our problem.

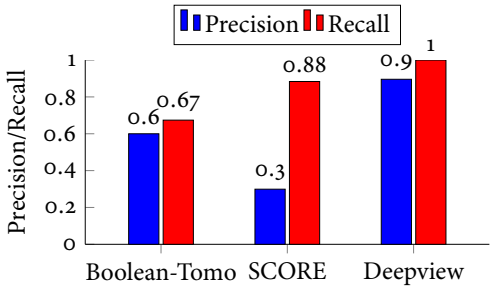


Figure 3.10: Precision/Recall comparison.

	Compute	Storage	Net	ToR
Precision	0.85	0.875	1.0	1.0
Recall	1.0	1.0	1.0	1.0

Table 3.3: Precision/Recall by failure type for Deepview.

Dataset. As we cannot run the other algorithms in production, we use trace data to compare algorithms. We had already hand-curated 42 incidents from a detailed study of tickets, so we use trace data from those incidents. They consist of 16 compute cluster issues (not ToR-related), 14 storage cluster issues, 10 unplanned ToR reboots, and 2 network issues. Only time periods when there is an incident are considered because a random sample is too sparse. Thus, we may overestimate the precision. But our comparison is fair since all algorithms use the same baseline ground truth.

Metrics. We compare each algorithm on recall and precision. Recall is the percentage of true failures that have been localized and precision is the percentage of localizations that are correct. In other words, high recall means we can localize most real failures, while high precision means we have few false positives.

Figure 3.10 summarizes the precision and recall for the 42 incidents. SCORE achieves a recall of 0.88, beating Boolean-Tomo, but it gives many false positives. Deepview, achieves both a high precision of 0.90 and a high recall of 1.0, beating both alternatives. Table 3.3 shows a breakdown of the precision and recall by failure types for Deepview. Overall, Deepview handles cases with a strong failure signal (Compute/ToR) and those with a weak failure signal (Storage/Network) well.

Deepview also does well for unplanned ToR reboots and Network incidents. However, there were fewer of these incidents, so the estimates are to be taken with a grain of salt.

The other advantage of Deepview is that its parameters needs no manual tuning. Parameters are set by cross-validation (for λ) or using a standard interpretable criterion (false positive tolerance of 1% for p-value). Boolean-Tomo and SCORE, instead, need careful tuning of their thresholds. In fact, we find that their precision and recall are sensitive to the thresholds. We picked those that maximize recall (as recall is typically more important than precision in production), while keeping precision as high as possible. We note that Deepview beats the performance of Boolean-Tomo and SCORE for all combinations of thresholds (omitted for lack of space).

3.6.3 Deepview Algorithm Analysis

We have introduced a set of techniques for our algorithm. Here, we analyze how useful each technique is.

Cross-validation and λ in Lasso Regression. The regularization parameter λ is set by cross-validation for each region. The optimal values found for incidents in Section 3.6.2 span three orders of magnitude with a minimum of 0.00012 and a maximum of 0.48. In fact, it is well known in statistical literature that choosing a universally optimal λ for all problems is impossible. The theoretical optimal [21] depends on the number of paths, the number of components, the structure of the network, and the error variance (i.e., how stable are VHD failures among different paths). When cross-validation is fast, it is preferred to a manual threshold.

Hypothesis Testing and Gray Failures. We use hypothesis testing to find a decision threshold to localize both big incidents and gray failures in the presence of random noise. The gray failure case studies in Section 3.6.1 show that hypothesis testing is essential. For the storage case, the failure probabilities are 0.34 for the truly failed storage cluster S0, and 0.002 and 0.047 for two normal compute clusters. Their p-values 3.9E−34 and 0.51 and 0.54 are needed to accentuate the difference and allow us to pick only S0. Similarly, for the network case, looking at p-values allow us to filter out T2_2.

3.6.4 Deepview Running Time

Algorithm Running Time. We measure the running time for Deepview algorithm in production. The worst-case running time is 18.3 seconds on a single server. It includes the time to read input data from Kusto, execute the algorithm and write the output data to Kusto.

Time to Detection (TTD) TTD is defined as the time between when an incident happens and when the failure is localized. The average time from a VHD failure event to its appearance in Kusto is 3.5 minutes. Adding the 5 minutes windowing time and the processing time, Kusto achieves a TTD under 10 minutes. This is a significant improvement over the previous TTD which typically lasted from tens of minutes to hours.

3.7 Discussion

Several architectural decisions were made when our IaaS was built. One is that a server connects to only a single ToR via a single NIC. While this makes ToR a single-point-of-failure (SPOF), the decision dramatically reduces networking cost. Another decision is that a VM can host its VHDs in any storage cluster in the same region. This makes load-balancing for storage clusters easy, but with potentially higher network latency and lower throughput. Further, both decisions may adversely impact VM availability. Using the data collected from Deepview, we can now study the impact of these decisions quantitatively.

3.7.1 ToR as a Single-Point-of-Failure

As we have described in Section 3.6.1, Deepview can detect unplanned ToR reboots. From the failure patterns, we find that there are two types of ToR failures: soft failures and hard failures. Soft failures can be recovered by rebooting the ToR, while hard failures cannot.

Our data shows that: (1) less than 0.1% switches experience unplanned reboots in a month; (2) 90% of the failures are soft failures, with the rest hard failures. The hard failure rate agrees with our ToR Return Merchandise Authorization (RMA) rate, which indicates that 0.1% switches need to be RMAed in one year. These numbers are obtained from a fleet of tens of thousands of ToRs.

The impact of a soft failure typically lasts for less than 20 minutes: 10 minutes for the ToRs to

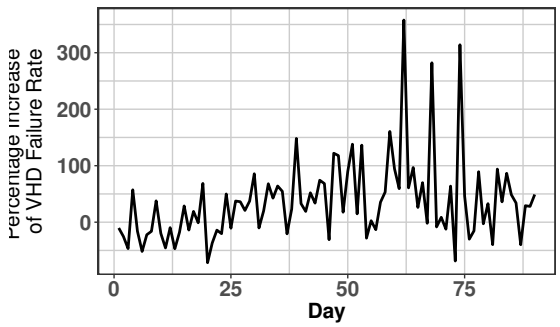


Figure 3.11: Daily percentage increase in VHD failure rate for VMs crossing T3 and above compared to those that only cross T2 for a 3-month period.

come up and 10 minutes for the VMs to recover. The impact of a hard failure lasts longer as the failed switch needs to be physically replaced. The impact to VMs can be shorter though as the VMs can be migrated to other hosts due to the separation of compute and storage. We conservatively use 2 hours as the impact period for hard failures.

If the ToR is the only failure source for VMs on that rack, the availability of our IaaS is no better than

$$1 - \frac{0.9 \times 20 + 0.1 \times 120}{1000 \times 30 \times 24 \times 60} = 99.99993\%$$

Even with ToR as the single point of failure, the service can achieve six-nines. This meets the rule of thumb that critical dependencies need to offer one additional 9 relative to the target service [97].

Thanks to Deepview data, for the first time, we are able to show that ToR as a single point of failure is an acceptable design choice for IaaS as it is not on the critical path for five-nines availability.

Note that simply examining ToR logs would not have given us these numbers, as many ToR reboots are planned, with no impact on VM availability.

3.7.2 Co-locate or Disaggregate?

A VM can use VHDs from any storage cluster in the same region, due to the separation of compute and storage. We look at the network distance between VMs and the storage clusters for their VHDs. We find that some 51.8% of VHD paths go through T2, 41.0% need to go through T3 and the rest go

above T3 in Azure.

A longer network path may result in higher network latency and packet drop rate. However, it is not clear whether it will also negatively affect VM availability.

Here we use the Deepview data to answer this quantitatively. We look at our data for three months. For each day, we first compute the VHD failure rates r_0 and r_1 for VMs crossing T2 only and VMs crossing T3 and above, respectively. Then, we find the percentage increase $(r_1 - r_0)/r_0$.

Figure 3.11 shows the daily percentage increase over a 3-month period. VMs whose network paths cross T3 network layer or above see a higher VHD failure rate than those that only need to cross T2 on most days. There is a 11.4% increase $((\overline{r_1} - \overline{r_0})/\overline{r_0})$ in the VHD failure rate if the VHD access needs to cross T3 or above.

One possible explanation is that as VHD requests go up the network tiers, they traverse more switches which may become oversubscribed. Thus VHD requests may become more likely to fail when network path lengths get longer. An implication of this study is that there is some benefit to colocating VMs and their VHDs in nearby clusters for availability.

3.8 Related Work

Machine Learning. Machine learning techniques have been used for failure localization, such as decision trees [13, 32], Naive Bayes [109], SVM [102], correlations [104], clustering [31], and outlier detection [87]. They allow domain knowledge to be encoded as features, but in general require a rich set of signals to discriminate different failure cases and may rely on assumptions about traffic that are not generally applicable. The most relevant work is NetPoirot [13], which targets a similar scenario as ours, but with a very different approach. NetPoirot is a single node solution where end-hosts independently run pre-trained classification models on local TCP statistics to infer failure locations. We believe NetPoirot and Deepview are complementary—TCP metrics from IaaS VMs may provide a useful signal to Deepview.

Tomography. There has been a large body work in network tomography (see [30] for a survey), and specifically binary tomography and its variants [42, 41, 67] for network failure localization. Typically, greedy heuristics are used to select among multiple solutions that all explain the observations.

Various thresholds are often needed to tradeoff between precision and recall ratios. Compared with those approaches, Deepview avoids manual threshold tuning and achieves both higher recall and precision as shown in section 3.6.2.

Bayesian Network. Bayesian network [81] is a principled probabilistic approach to failure localization. It can model complex system behaviors [14] and handle measurement errors [63]. While exact inference is intractable [66], there are various approximation techniques such as using noisy-or to simplify conditional probability calculation [83, 14, 92], considering k-subset root-causes to shortcut marginalization [63, 14], using a simple factored form for joint posterior [83], or using message passing for faster inference [92]. For our problem, we find that using a combination of approximation techniques (we tried two [83]) was essential. It is future work to compare Deepview with some practical Bayesian network approach.

3.9 Summary

In this chapter, we have demonstrated that taking a global view of failures and using a penalized-regression-based algorithm can handle the challenges of complex component interaction, gray failures and unpredictable component behaviors for diagnosing VHD failures for IaaS service. While we have modeled compute cluster internals in detail, it would be future work to open up storage cluster internals too for finer-grained failure localization.

Due to opaque network routing, Deepview is unable to localize network failures to finer granularity, e.g., to individual links or switches within the network, rather than network tiers. The next chapter will introduce another key idea to deal with this challenge.

Chapter 4

Volur

In Chapter 3, we demonstrated that we can localize VHD failures to components in compute, storage and network subsystem for IaaS clouds. One limitation is that our localization granularity for network components is coarse-grained—we can pinpoint failure to a network tier rather than a specific network device. The reason is that the datacenter network has many redundant paths and the routing is opaque to datacenter operators. We cannot predict on which path a request of a packet is routed, and thus we needed to aggregate network devices and simplify the network topology in our model.

Switch routing is unpredictable because details of its load balancing module are hidden from endhost software and hardware. Normally, modularization is a positive, allowing a separation of concern. However, we show that this makes failure localization very challenging when the network fails to detect the problem itself. While there are many workarounds to deal with the difficulty of not knowing how a packet is routed by a switch, we believe that the right approach is to require the switching behavior of switches to be externally predictable. We prescribe this as an architectural design principle for datacenter networks and show what is required to make it work.

In this chapter, we describe Volur, a prototype predictable network architecture with three components: a predictable switch, a network state service and an end-host path module. In this network, end-hosts can predict and control the paths packet take in the network as long as switches route only based on packet headers and infrequently changing network state. This allows us to build a failure localization application that can localize gray failures down to the granularity of a single link or switch or routing table entry. A limitation of the approach is that it prevents switches from making decisions based on purely local information, unless that information is visible to endpoints. For the

case of load balancing, we show that we can build a load balancing application that can emulate dynamic in-network load balancing feature just from the end-hosts.

In our evaluation, we verified the feasibility of Volur networks in a large production datacenter. We also show in a testbed that the failure localization application can achieve accurate localization for single failure and multiple concurrent failures, even in the presence of fast in-network failover and inaccurate path prediction. Finally, we show with ns-3 simulation that our end-host load balancing emulation technique can approximate the performance of the state-of-art dynamic in-network feature.

4.1 Introduction

To tolerate faults, to balance load, and to scale, today's datacenter networks provide tens to hundreds of independent paths between any two servers. The selection of which path to use for any particular packet—routing—is often complex and opaque.

A canonical example of this effect, Equal-Cost MultiPath (ECMP) [95], sees switches selecting among equivalent paths via a pseudo-random but deterministic function of the packet header and local switch state. This routing function is, by default, unknown to endpoints and network operators. Many other aspects of route selection exhibit similar opacity.

An unintended side effect of the intelligence we have built into the network is that failure localization is more difficult whenever the network is unable to detect and correct the problem itself. For example, while heartbeats and switch counters will catch many problems, some failures are silent or intermittent [112]. With opaque routing, network operators have no way to determine where these packet drops are occurring, or why. Unpredictable routing also limits options for rapid failure recovery—even if end hosts could determine the location of the failure, they have few options to avoid it.

While a number of solutions have been proposed to improve diagnosis and repair [70, 53, 112, 87, 94], there is something of an arms race with advanced network features that add complexity to switch routing. For example, attempts to address the inherent load imbalance of asymmetric networks [10, 48] invalidate many assumptions made by diagnostic tools. Programmable switches [24] further

add to the unpredictability of the network. How do we continue evolving our networks without making them harder to debug?

OpenFlow argued successfully that switches should provide explicit external control over their forwarding tables [74, 79]. In this paper, we propose to extend and relax that design principle. Specifically, we argue that switch routing behavior should be externally predictable by the trusted network-layer software running on the endpoint—failure handling and network features need not be in an arms race. In contrast to source routing, which advocates for perfect control and predictability of a *dumb* network using a *smart* edge, we allow for imperfect predictions; in-network routing decisions are fine as long as they are predictable and/or infrequent. For instance, switches can do immediate adaptation for local failure recovery as long as end hosts eventually become informed of the new network state—that is sufficient for fault localization and avoidance. Thus, our approach can be seen as a way for *smart* networks and *smart* end hosts to coexist.

The primary challenges of our approach are threefold. First, what information do end hosts need to predict network behavior and how can they acquire/use that information efficiently? Second, for the features that make today's networks unpredictable, can we replicate some of those features in a predictable way? Finally, given a predictable network, how do we prevent applications from maliciously directing traffic, e.g., to overload parts of the network?

In order to answer the above questions, we designed and implemented a prototype network architecture called Volur that is composed of three components: (1) switches that route using predictable functions of the packet's header and switch configuration state, (2) a network state service that disseminates any required information to end hosts, and (3) a per-end host path choice module that models network behavior. On top of Volur, we build two applications that utilize the predictability of the network to locate failures and balance load.

We build Volur network predictors for two different deployments to verify the feasibility of predictable networks: a large production datacenter with upwards of 100 thousand devices and a smaller testbed. These deployments span multiple switch ASICs in switches from multiple vendors.

To evaluate our architecture as a whole, we utilized the aforementioned large production datacenter deployment; a second, modestly-sized Cloudlab testbed; and an ns-3 packet-level simulated

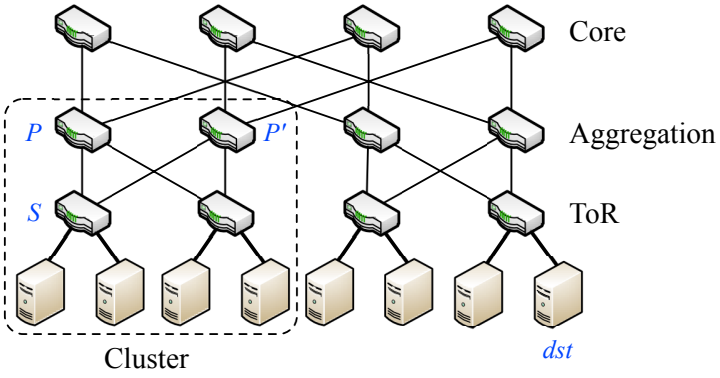


Figure 4.1: A canonical 3-level Folded-Clos topology. The three levels of switches are termed ToR switches, aggregation switches, and core switches. Folded-Clos networks have inherently high path diversity; switch *S*, for instance, has 4 distinct paths to *dst*.

network. Our results show that:

- With ECMP specifics from switch vendors, an end host can already accurately predict and control path choice on unmodified switches in a large scale production datacenter.
- Volur can locate non-fail-stop failures with over 0.95 precision and recall for single failure, and over 0.90 precision and over 0.50 recall for multiple, diverse failures.
- Volur can recover from failures within a fraction of a second by steering flows.
- Even in unbalanced networks, Volur-based load balancing can approach the performance CONGA, a state-of-the-art in-network design, without giving up routing predictability.

4.2 Motivation and Related Work

Datacenters can vary greatly from one deployment to another, sometimes even within the same company. Even so, two features are common to most deployments: (a) a tree-like network topology and (b) opaque, ECMP-based routing over the tree.

As described in Chapter 2, datacenter networks are typically multi-level, multi-rooted trees of switches like the one in Figure 4.1 [8, 50]. The canonical version has three levels: (1) the ToR switches,

which connect a single *rack* of servers, (2) the aggregation switches, which connect the ToRs of a single *cluster*, and (3) the core switches, which connect all of the clusters of the *datacenter network* together.

A multi-level, multi-rooted tree offers many equally long paths between any two end hosts. With ECMP, switches can be configured to use any alternate next hops instead of just one. As an example, consider Figure 4.1, where switch S has a total of four paths to *dst*—two through each of its two uplinks. When a packet for *dst* arrives, S will use ECMP to compute a pseudo-random function on fields in the packet header. This result determines the next hop and is configured so that packets on the same flow will be assigned the same path. In turn, S's parents, P and P', will choose one of their two possible next hops with equal probability. By default, end hosts and operators have no visibility into the pseudo-random function or its configuration.

Other routing elements are similarly opaque to operators. Examples include Link Aggregation Groups, which operate similarly to ECMP, but among point-to-point links rather than paths; Resilient Hashing [25], which dictates ECMP behavior so that flow assignment is stable even as links are brought up or taken down; and in-network load balancing like LocalFlow [89], DRILL [48], DLB [73], and CONGA [10], which route based on transient workload statistics.

4.2.1 Failures in Data Center Networks

That modern datacenter networks have many paths is a great boon to performance, cost, and fault tolerance. The opaque *use* of that diversity, however, presents a significant challenge to the identification and isolation of network failures, particularly when the failures evade traditional debugging tools.

In this paper, we focus on failures that manifest in the form of packet drops. In real networks, these can be a result of anything from loss of power to an errant routing table entry.

Some types of failures are easy to detect. For instance, if a link is severed, operators can discover the issue in many ways. In addition to physical-layer detection, heartbeat protocols like BFD [65] or BGP keepalive messages will eventually time out on either end of the link. Similarly, drop counters on both switches will reflect a high number of drops on the affected interfaces. Even traceroutes can

pinpoint the failure.

Unfortunately, when failures are not strictly *fail-stop*, the above techniques can fail. There are many such non-fail-stop behaviors, and we discuss some of them below.

- *Partial failures:* Some non-fail-stop errors are stochastic in nature. Here, a flaky link or a faulty switch may drop a small fraction of packets [112]. The occasional heartbeat or traceroute packet can get (un)lucky and miss the problem while application traffic will continue to experience packet drops. Increasing the rate of probing increases overheads and is still not guaranteed to detect the problem.

- *Silent failures:* While operators can sometimes detect partial failures by examining switch counters, some of them are also silent. With silent failures, buggy software or faulty hardware cause switches to drop packets without incrementing any switch drop counters. When a failure is both partial and silent, it will evade detection from heartbeats and traceroutes in addition to switch counters. These are known to occur in practice [68, 112, 6].

- *Input-dependent failures:* Still other failures only affect particular flows [112, 6]. For example, a routing table entry corruption can cause packets with a specific header to get blackholed. Heartbeats and traceroute probes may not match the same entry as the problematic traffic. Post-hoc localization of a known failure is possible, but can be very time consuming [112]. As above, when a failure is both silent and input-dependent, traditional mechanisms such as heartbeats, switch counters, and traceroutes are all ineffective.

In addition to the difficulty of detecting such failures, non-fail-stop events also present challenges to mitigation [71, 103]. For instance, in the case of input-dependent failures, network operators are forced to make a choice between two extremes. They can either take the component down while they repair/replace it, or they can leave the component up while they diagnose the problem. The former negatively impacts unaffected flows; the latter negatively impacts the victim flows.

4.2.2 Related Work in Failure Handling

Motivated by the above issues, researchers have proposed various solutions to better locate and route around failures. Broadly speaking, these take one of two approaches. They either (a) introduce new network features to better track/control the routes of packets, or (b) work around uncertainty in the network with clever end-host-based techniques. While both approaches have resulted in interesting and impactful work, they can be fragile to the constantly increasing complexity of datacenter networks.

4.2.2.1 Network-based Approaches

While switches have long provided features to help track packets and their drops [28, 33, 82, 44], these features have traditionally been very course-grained, e.g., per-queue granularity. Recent work has thus tried to improve measurement granularity; however, these systems all have important drawbacks, both with and without in-network load balancing.

Sampling. One approach to improving network debugging is to improve the flexibility and overhead for switches to sample traversing packets. NetSight [53] and Everflow [112], for instance, allow on-demand insertion of rules to forward either the packet itself or a summary of the packet to a separate collector. FlowRadar tracks statistics for every flow by optimizing the collection mechanism at the switches [70]. Although these approaches are powerful, there is a hardware cost to collecting packets, and forwarding or analyzing them at the switch.

Tagging. A lightweight alternative is to tag packets as they pass the switch [87, 94]. End hosts can use the tag to map flows to their paths. A drawback is that locating failures with tags requires successful delivery; if none of the target packets make it through to the destination, the failure will remain opaque. Thus, accuracy can suffer for input-dependent failures or workloads with very small flows. Making matters worse, in-network load balancing proposals break flows into small flowlets of just a few packets each [10, 99]. Smaller number of packets makes it harder to estimate component drop rates accurately.

4.2.2.2 End-host-based Approaches

Another class of prior work attempts to locate and route around failures without modifying the network.

Active probing. Common forms of active probing include pings and traceroutes. Pingmesh [52] and NetNORAD [6] ask servers to send probes at a controllable rate in order to determine the latency and packet loss rate of the network. Because of the unpredictability of ECMP, both are limited to the granularity of groups of switches in the same tier; traceroutes are needed to further localize the error. In a network with load-dependent routes [78], this second localization step becomes impossible.

Source rerouting. There are also proposals for end-host-based rerouting. An end host using FlowBender [61], for example, will change its packet header when it detects congestion. The hope is that a different packet header will result in a different ECMP path through the network. MPTCP [84] uses a similar technique to obtain multiple paths through the network. End hosts then send packets along those multiple paths simultaneously, choosing each path's rate based on congestion signals. This technique does not help in localizing the fault. However, our approach is compatible with end-host solutions like FlowBender [61] and MPTCP [84] by making it possible to explicitly select an end to end route.

4.3 Volur: A Predictable Network

In this paper, we explore an alternative design choice. Rather than continue the arms race by working around an increasingly complex network, we investigate the design of a predictable network made of predictable switches. With a predictable network, end hosts can assist in handling failures: they can deduce the path of dropped packets and pick packet headers that are predicted to avoid those paths.

An obvious solution to achieving predictability is source routing using schemes like those proposed in [57] or [86]. However, in traditional source routing, end hosts have full control over how their packets are routed. In essence, a *dumb network* is controlled by a *smart edge*. While this would make failures easier to locate and route around, a naive application of source routing to datacenters

Challenges	Solution	Section(s)
Is it possible and/or practical to predict network behavior?	Some deployed networks are already predictable. More generally, we anticipate that the OpenFlow model may also apply here—if customers value predictability, vendors will provide it as a feature.	4.3.1, 4.5.1
Is predictability compatible with dynamic switch behavior?	For infrequently-changing behavior (e.g., failover), Volur disseminates versioned network state to end hosts.	4.3.2
	For frequently-changing behavior (e.g., load balancing), end hosts can approximate current switch features.	4.4.2
How do we deal with unpredictable failures and other inaccuracies in prediction?	End-hosts use versioned topology to sieve out reliable drop statistics. Common-case consistent hashing limit routing changes.	4.4.1.1, 4.5.2.3
How do we defend against DDoS attacks that might be enabled by end host path prediction/control?	Only convey topology to the end host trusted computing base. A NAT can be used if extra protection is needed.	4.3.3

Table 4.1: A roadmap of key challenges in creating a predictable network and their solutions.

surrenders at least two crucial features:

- *Fast failover:* Easily-detectable failures like signal loss on a link are simpler and faster for the network to handle. In these cases, switches adjacent to the failure should perform detection and rerouting as they are able do so at timescales much less than the RTT of the network. This fast failover is essential for achieving high network availability.
- *Backward compatibility:* Most current applications and operating systems are designed to be agnostic to the routing decisions of the network. While it is possible to convert all applications, OSes, and/or hypervisors to use source routing, an ideal solution would permit the use of legacy software.

Rather than shunt all routing responsibility to end hosts, our only requirement is that network routing be predictable. Thus, one way to view our work is as a framework for a *smart network* and a *smart edge* to coexist.

We restrict switches to route based only on the packet header and infrequently changing configuration state. Compared to pure source routing, prediction in this model is not always accurate. The

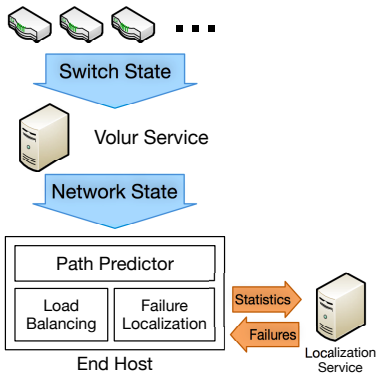


Figure 4.2: The primary components in Volur, our instantiation of a predictable network. A logically centralized Volur Service collects the current configuration of the network and disseminates it to end hosts, which use it to predict paths. Predictions can be used for many purposes, from locating failures to balancing load.

point of fast failover, for instance, is that the switches know about and can react to failures faster than end hosts. Immediately after a failure, the network may not operate as end hosts expect. Instead, we tolerate a small amount of inaccuracy in return for these features.

There are several challenges in making such a system practical, which we list in Table 4.1. Is our approach compatible with network features that are currently implemented using dynamic switch behavior? How do we deal with fundamentally unpredictable behavior like failures? Can we defend against DDoS attacks that might arise from end host path prediction/control?

The primary contribution of our work is to characterize what it takes to design and implement a predictable network and to detail its benefits/limitations. Our architecture is called Volur and it consists of three primary components: *switches* that are predictable, a *Volur service* that gathers and distributes the current state of switches, and *end hosts* that use that state to predict routes. The overall architecture is illustrated in Figure 4.2.

4.3.1 A Predictable Switch

Our primary design principle is simple to state: switches should route only on the packet's header and infrequently changing configuration state. Note that this restriction only applies to functions

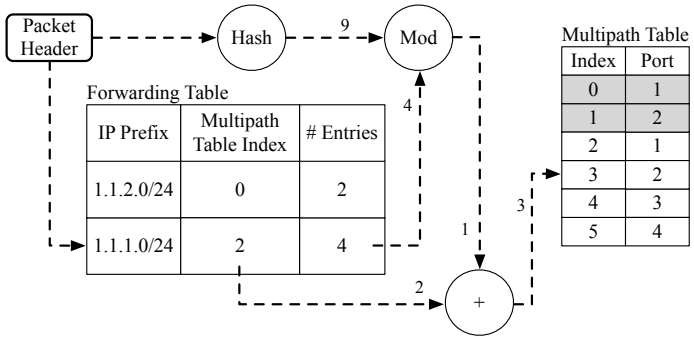


Figure 4.3: A conceptual diagram of how the L3 processing stage in the presence of ECMP. This diagram is adapted from [111].

that affect the packet's path—features such as management, monitoring, QoS, and queuing are all orthogonal.

4.3.1.1 A Simplified Packet Pipeline

To explore the operation of a typical switch and see why our design principle is congruent with common-case network features, we describe the packet processing pipeline of a simple predictable switch. We also detail the network state that each pipeline stage relies upon. Due to space constraints, we limit our exposition to the subset of the pipeline necessary for forwarding a simple Ethernet and IPv4 packet without VLANs or tunneling. The switch we describe allows for both fast failover and backward compatibility. A surprising result is that our simple model maps well to the current configuration of switches in a large production datacenter (see Section 4.5.1).

L2 processing. When a switch receives the packet, it first looks at its L2 Ethernet header. If the destination MAC of the packet matches the switch's MAC address, the packet will continue to L3 processing. Otherwise, the packet will be switched as a raw Ethernet frame (we omit those details). *Depends on: packet header and switch's MAC address.*

L3 processing. In L3 processing, the switch begins by looking up the destination IP in its forwarding table. The resulting entry may either point to an egress port, point to multiple egress ports, or indicate that the packet should be dropped. When there are multiple possible next hops, as is often

the case in Clos networks, the switch will calculate a hash function over several subfields of each packet's header. The result of the hash function (modulo the number of possible next hops) is used as an index into the next hop table. Figure 4.3 depicts this process.

Traditionally, ECMP has been considered unpredictable, but for efficiency, modern ECMP implementations are typically deterministic. A typical ECMP implementation will hash on a packet's 5-tuple (source address, destination address, source port, destination port, and protocol ID) using simple hash functions like XOR, CRC, or table lookups [62, 56, 37]. Hash functions can be combined with hash seeds, preprocessing, bit shifting, masks, and resilient hashing techniques to improve results in various situations [25]. All of the above functions are approximately predictable as long as changes are relatively infrequent.

Depends on: packet header, L3 forwarding table, multipath table, and ECMP hash configuration.

Buffering. In a predictable switch, the queuing/scheduling should not affect the choice of next hop.

Egress modifications. Finally, before the packet is sent back out on the wire, the switch will update the src and dst MAC addresses to correspond to the next L2 hop. In addition, it will recalculate the TTL field and IP and Ethernet checksums.

Depends on: switch's MAC address, neighbor's MAC address, and packet header.

4.3.1.2 Applicability to More Complex Pipelines

Modern networks sometimes expect many more features than described above. Some of these simply add more fields upon which to route, e.g., tunneling, VLANs, and QoS. With regards to routing decisions, these can be made predictable as they rely on the packet and infrequently changing state. Even recent programmable switch designs like P4 [24] can be made predictable if switches' match-action tables are required to be deterministic on the packet header and infrequently changing state.

Other features depend on dynamic switch-local state and are more difficult to predict, e.g., load balancing that relies on instantaneous queue length or utilization. There are several alternative ways to implement such protocols in a predictable fashion. For instance, load balancing decisions can be emulated by propagating the decision back to end hosts, who are then responsible for routing. In this

way, we can trade reaction time for predictability. In Section 4.5.3, we examine some state-of-the-art in-network load balancing solutions and show how they can be approximated in a predictable network; however, we note that it is out of the scope of this paper to enumerate *all* possible dynamic network behavior and their predictable alternatives.

4.3.2 The Volur Service

Predicting the route of a packet requires both the packet's header and elements of the switch's current state. For the sender of the packet, obtaining its header is simple. For the other piece of information—switch state—we introduce a simple aggregation service that gathers up-to-date state from every switch and disseminates it to every end host. This dissemination must be performed on any switch state change including link failures/recoveries and control plane routing updates. Replication and sharding of such a service is straightforward; for ease of explanation, we assume a logically centralized Volur service.

The primary goal of the Volur service is to disseminate switch state updates as quickly as possible. There are two steps:

Switches to the Volur service. As state updates may occur at irregular intervals and must be disseminated quickly, switches mostly operate on a push model. When a state change occurs (e.g., a BGP update or link failure), switches will immediately send a `diff` of their state to the Volur service. The service also periodically pings each switch for a hash of their current state to ensure that it is still alive and correctly synchronized. In systems with an existing centralized SDN infrastructure, the Volur service is a natural extension to the SDN controller.

Volur service to end hosts. The second step is to disseminate the state changes to end hosts. There are two channels for state dissemination in our system. The end hosts periodically pull a full snapshot of the current network state. In addition, the Volur service broadcasts versioned, perishable state updates to end hosts. These updates are sent using UDP to ensure time bounds.

1. When a switch updates its state, it sends a `diff` of the state change to the Volur service. Let the maximum propagation delay of this message be t_1.

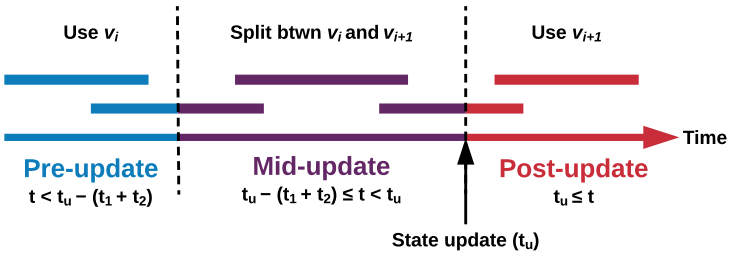


Figure 4.4: End hosts classify traffic into three classes given a state update at time t_u: pre-update, mid-update, and post-update.

2. Upon receipt of the update, Volur increments its version number $v \rightarrow v + 1$ and distributes the update to all affected hosts. Let the maximum delay of this message be t_2.
3. Upon receipt of an update from the Volur service, hosts send back an acknowledgment.
4. If the ACK is not received after some predetermined timeout, inform the end host during its next checkpoint.

Given the above protocol, we can quantify the length of three distinct phases of prediction accuracy. Assume that the end host receives an update at time t, as shown in Figure 4.4. *Pre-change* predictions (before $t - (t_1 + t_2)$) are correct. *Mid-update* predictions are slightly uncertain in that they can follow either state v or $v + 1$. This period lasts from $t - (t_1 + t_2)$ to t. *Post-update* predictions starting from time t should all follow version $v + 1$. If the end host, during a checkpoint, finds out that it failed to acknowledge an update, all predictions between the current checkpoint and the last one are potentially inaccurate. All inaccuracies are handled by higher-level network applications.

4.3.3 End Hosts

Given a predictable network, end host operation is relatively straightforward. We provide to each end host a switch predictor that takes a switch state object and an input packet header. The output of the predictor is a next hop and output packet header.

Predicting a packet's path. For every packet, path prediction is just a matter of iteratively chaining

the next hop and packet header predictions of each intervening switch. The switch state object is obtained from the Volur service as described above; the end hosts already have the initial packet header.

Controlling a packet's path. To control a packet's path, end hosts only need to find a packet header that maps onto a target/acceptable round-trip path. As switch operations are typically not cryptographically secure, it is often possible to create an efficient inverse for them. In Section 4.5.1, we show that such techniques can be used to generate headers for specific paths in our large production datacenter network in under 12 µs. Solutions are not guaranteed to exist, but operators work hard to ensure that hash functions spread packets over the entire network evenly, and to prevent hash polarization.

End hosts have at least a few options when trying to craft a header to hit one of n paths. In general, they need $\sim \lceil \log n \rceil$ bits that are otherwise unused by the network. They can combine several bit regions to obtain a larger 'address space':

- *IPv6 flow labels:* A 20-bit-wide field that indicates packets belong to the same flow. It is intended for purposes like ours.
- *Port numbers:* The source and destination ports provide up to 32 bits. For the source port, this may require a small modification to the way port numbers are allocated in the OS.
- *IP addresses:* IP addresses are also possible. For instance, we can create a path address space of 8 bits by giving each server a /24 or /120 for IPv4 and IPv6, respectively.

For legacy hosts, they can send packets without choosing any particular header field. Those packets will be load balanced with ECMP just as they are today.

Preventing malicious control of paths. As a corollary, our architecture allows for efficient defenses against DDoS attacks. A potential concern with our system is that it may allow malicious users in multi-tenant datacenters to launch a targeted DDoS attack against individual network elements. To that end, we note that without up-to-date switch state, the network is not more predictable than it is today—the configuration state space is very large and constantly changing. Further, because cluster

and fabric switches and links have extremely high capacity, it would be difficult for the attacker to determine whether any particular trial succeeded at steering to a particular path without access to datacenter internal traceroute. Thus, the Volur service only distributes state to the trusted computing base, and not untrusted applications/VMs.

Even so, if more security is necessary, the VM layer can pick a random source port or flow label for each connection, similar to the NATs that many VMMs already use. If even a small part of the header is randomized, steering is difficult.

Changing paths mid-connection. Beyond controlling a single packet's path is controlling the path of an entire TCP connection. For new connections, this is just a matter of choosing a suitable 5-tuple for the connection. To reroute existing connections to avoid a failed or congested network component, Volur must change the packet headers without disrupting TCP's ability to demultiplex traffic. IPv6 flow labels are a good candidate for this. Otherwise, e.g., in the case of TCP source ports, the VMM/OS may need to rewrite the packet headers.

To be more concrete about this second option, when Alice wishes to change the path of a connection with Bob, she might decide on a TCP source port, s, that results in the target forward and reverse paths. Alice will send the new source port to Bob asynchronously in a separate connection. This must be done out-of-band because in the case of a failed path, the original connection may not be usable. When Bob acknowledges the new source port, Alice installs rewrite rules as follows:

- For *outgoing packets*, Alice overwrites the source port number with s.
- For *incoming packets*, Alice remembers the original source port number in a hash table so that when a response comes in, she can insert the original port transparently.

Bob installs similar rewrite rules:

- For *outgoing packets*, Bob overwrites the destination port number with s.
- For *incoming packets*, Bob remembers the original destination port number in a hash table so that when a response comes in, he can insert the original port transparently.

Both ends of the connection can initiate such a path change, but to prevent flapping, we designate the client that called `connect()` to be responsible for most path changes.

4.4 Applications of Volur

We show two applications of Volur's predictability. The first is a fault localization service that addresses our original motivation—to find/handle faults that the network misses. The second is a load balancing mechanism that simultaneously demonstrates the power of our approach and shows how to emulate state-of-the-art dynamic network behavior predictably.

4.4.1 Volur-FL: Fault Localization

Using predictability, end hosts can attribute packet drops to a path (or a set of paths during a topology change). Thus, the two key challenges in locating failures are to determine: (1) whether the drop was due to congestion or a failure, and (2) if the drop was due to a failure, which component on the path is responsible. We use the penalized regression algorithm proposed in Deepview [107].

4.4.1.1 Collecting Drop Statistics

Volur-FL first collects drop statistics for each path. For TCP traffic (the majority of datacenter traffic), drop information is already readily available in the form of retransmission statistics. Volur-FL uses Linux eBPF (Extended Berkeley Packet Filters) [35] to gather these statistics on a per-connection basis.

Specifically, we track two TCP variables: *pktsSent*, the number of packets sent and *pktsRetrans*, the number of packets retransmitted. Hosts poll these statistics every 10 s. Note that these variables track control packets that are ACKed (e.g., SYN/FIN packets) in addition to data packets. It is important to track control packets since, for black holes, no data packets will be sent, only SYNs. These statistics are approximations of the ground truth as in-flight packets, cumulative ACKs, and spurious retransmits can affect these numbers; however, our evaluations show that this approximation is effective in practice.

Non-TCP traffic is slightly more complex as not all protocols acknowledge packet receipt (e.g., UDP). For them to be used in fault localization, they must be extended with simple ACK packets or some other type of coordination to detect when traffic is dropped; the ACKs do not need to be used for any other purpose. The rest of this paper assumes TCP traffic.

End hosts attribute the drops to paths as described in the preceding section. To handle uncertainty during the mid-update period, they evenly attribute drops to all applicable predictions. For example, suppose that a single TCP connection has 100 packets. If there are two possible versions, we attribute 50 packets to each path. If there are four, we attribute 25.

4.4.1.2 Implicating Components

Volur-FL uses path drop statistics to then implicate faulty components. We model server-to-server network paths and various network components along the paths. We focus on links, switches, and routing table entries (RTEs), which are among the most common network failure granularities. Our goal is to find which network components to blame given observed packet drops.

There are two steps in our algorithm as described in the previous chapter. First, we use Lasso regression to infer the component loss rates for each component based on path loss rates from all paths. Then, we use hypothesis testing to decide which components to blame. Here we introduce the variables in the context of Volur, and refer the reader to the previous chapter for the justification for the algorithm.

Each path i is observed to have transmitted T_i and dropped D_i packets, and each component j has an unknown loss rate $(1 - p_j)$ we want to infer. We assume drops are independent for simplicity. We can derive the Lasso regression equations as we have done for Deepview in the previous chapter:

$$\begin{aligned}
\mathbb{P}(\text{path i is fine}) &= \prod_{j \in \text{path}(i)} \mathbb{P}\left(\text{component j is fine}\right) \\
\frac{T_i - D_i}{T_i} &\approx \prod_{j \in \text{path}(i)} p_j \\
y_i &= \sum_{j=1}^{N} \beta_j x_{ij} + \varepsilon_i, \qquad \varepsilon_i \overset{i.i.d.}{\sim} N(0, \sigma^2)
\end{aligned} \tag{4.1}$$

where $y_i = \log\left(\frac{T_i - D_i}{T_i}\right)$, $\beta_j = \log p_j$, and the binary variable $x_{ij} = 1$ iff i-th path goes through the j-th component.

After solving these regression equations, we use the following hypothesis test to decide which components to blame:

$$H_o(j) : \beta_j = \bar{\beta} \quad \text{v.s.} \quad H_A(j) : \beta_j < \bar{\beta} \tag{4.2}$$

If the data allow us to reject $H_o(j)$ in favor of the alternative hypothesis $H_A(j)$, then we can blame component j. Otherwise, we do not blame component j.

4.4.2 Volur-LB: Load Balancing

The second application we explore, Volur-LB, demonstrates predictable dynamic load balancing based on CONGA [10]. A consequence of our design is to show that it is possible to predictably emulate across a wide range of operating conditions a state-of-the-art dynamic network feature. We note that concurrent work has shown that end host load balancing, with careful design, can outperform current in-network approaches [106], but our focus is on emulation. We also note that our goal is not to predict *all* possible in-network functionality; however, we hope that the design of Volur-LB provides some insight into the design on features that do not require unpredictability.

At a high level, CONGA-enabled networks perform two functions. First, switches tag passing packets with congestion metrics and feed that information back to the source ToR. Second, the source ToR waits for a sufficiently long inter-packet gap, rerouting each *flowlet* toward the least-congested path. In Volur-LB, we separate these two functions explicitly and offload the second (flowlet rerouting) to end hosts.

4.4.2.1 Collecting Congestion Metrics

Like CONGA, Volur-LB gathers congestion metrics via in-band feedback. As a packet travels from the source to the destination ToR, switches tag the packet with their current load if it is larger than previously tagged values (see [10] for details). The destination ToR then feeds these path-level con-

gestion metrics back to the source ToR by piggybacking the information on normal traffic. For every feedback-carrying packet, the destination ToR sends a single path-level metric, choosing amongst them in round-robin fashion.

At the end of the above process, the source ToRs have a lowest-utilized path toward every destination (multiple in the case of ties). In addition, as none of these operations affect routing, they can all be done without losing predictability.

Where we begin to differ from CONGA is with an extra step to transfer the congestion metrics to servers in the source rack. Volur-LB uses two mechanisms. First, the ToR switch uses incoming traffic to the rack to opportunistically piggyback the congestion metrics to its member servers. For every packet sent to a member server, the ToR switch tags it in its egress pipeline with a (destination rack, best path to the rack) tuple. The destination rack is chosen in a round-robin fashion, and if there are multiple best paths, a hash of the packet header is used to break tie. In theory, congestion metrics kept at servers would be less up-to-date compared to what their ToR switches maintain. However, for servers that communicate often with others, their congestion metrics would be refreshed timely by incoming ACKs or data packets. The second mechanism allows servers to query their ToR for the best path to a destination leaf using UDP packets. Servers send those requests to ToRs at connection setup in parallel with their SYN packets. The on-demand query allows servers to steer to good default paths after long silence.

4.4.2.2 Flowlet Steering

In parallel with congestion metric collection, end hosts in Volur-LB monitor inter-packet spacing to detect flowlets [99]. For every new flowlet, the server steers the flowlet toward the destination's last 'best path'. Since end hosts know when and where flowlets are rerouted, predictability is maintained. Extension of Volur-FL to flowlets instead of flows is straightforward.

Our approach maintains the metrics and features of CONGA with minimal extra overhead (some additional header data on ToR-server packets). Pushing the decision to servers increases the latency of feedback and decreases the rate at which feedback arrives at the decision point, but per our evaluation in Section 4.5.3, the effects are negligible.

4.5 Evaluation

We leverage a few evaluation platforms. To evaluate the feasibility of predictable networks we implement one on top of a large commercial datacenter. To test the performance of failure localization in a more controllable environment, we use an 80-machine Cloudlab testbed. Finally, to test the relative performance of Volur-LB and CONGA, we simulate the necessary hardware changes in ns-3. We show that:

- Some of today's networks are already predictable without modifying hardware or nonparticipating end hosts.
- Volur-FL is effective in locating a diverse set of failures and is robust to topology updates.
- End host dynamic load balancing can closely approximate the performance of state-of-the-art in-network approaches.

4.5.1 A Predictable Production Prototype

We begin by demonstrating the feasibility of prediction using an implementation of Volur on a large production datacenter. The datacenter has upwards of one hundred thousand devices and hosts a variety of applications, from frontend web servers and caching to backend storage and data analytics. For the most part, servers are connected into the network with a single 10 Gbps link, while interconnect switches use 40 Gbps links.

All of the switches are based on chipsets from one of the biggest manufacturers of switch ASICs, but span multiple vendors. These switches support a diverse set of configurations for routing. Just for ECMP, the options included flexible field selection, hash seeds, pre- and post-processing steps, and many possible hash functions. Our predictor faithfully reproduced the path computation pipeline of a switch along with the effects of all of these configuration options. It gathered the options from switches in order to perform predictions.

Our prototype did not require any modifications to either switch configurations or OS configurations—the network, as configured, already approximated a predictable network. We also verified the fea-

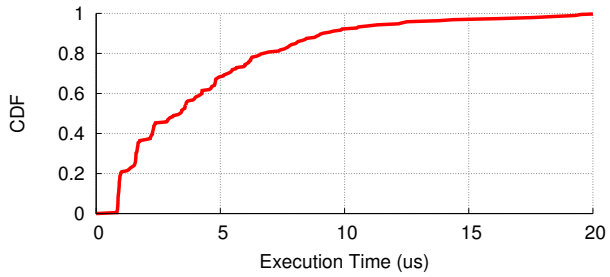


Figure 4.5: CDF of the time it takes to compute a header for a specific path. The topology we use is a fully-deployed version of a recently published datacenter architecture [12].

sibility of our system on top of a testbed of Cavium switches and ASICs, but we omit those results here.

4.5.1.1 Predicting Paths

To test the accuracy of our predictor, we ran a number of UDP traceroutes between several servers within a cluster and compared their results to the results of our path prediction engine. The ToR switch was already configured with an ECMP group. When replicating the relevant configuration options within our path deduction engine, we are able to replicate 100% of the results recorded by the UDP traceroute experiment. We also built the predictor's inverse for the purpose of efficiently generating headers for a target path.

Overhead of prediction. In addition to verifying that our prediction engine can accurately predict paths, we also tested the efficiency of the engine when trying to find a header for a particular network path on a 2.60 GHz Intel Xeon CPU E5-2670. We use a topology based on a recently published datacenter architecture [12]. In it, ToR switches have four uplinks each and aggregation switches have 48. As our inverse function is only able to reverse a single switch's routing function at a time, we repeatedly generate a valid header for one switch, rejecting the result if it does not map to the correct routing choice on the second switch or if it requires a reserved port.

Figure 4.5 shows a CDF of the execution time. We randomly choose a target path and fix every part of the packet header except the UDP source and destination ports. We then track the time it

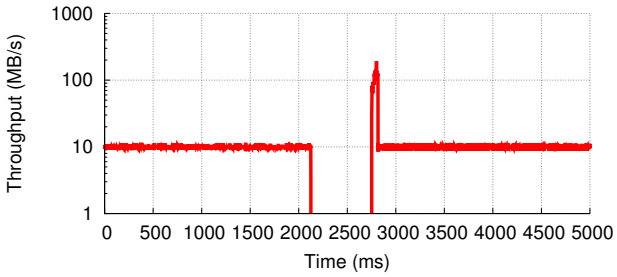


Figure 4.6: Throughput of a constant-rate connection in the presence of a failure. Upon TCP timeout, the sender transparently switches paths. The post-failure spike was due to a rush of ACKs.

takes to find a target header for the given path. We were always able to find a valid header. The median time to find a valid pair of ports is 3.4 µs.

4.5.1.2 Controlling Paths

To demonstrate path control, we implemented an `iptables` user-space application called 'ECMP-interpose' that automatically and transparently modifies connection parameters when a TCP timeout occurs. Modifications are as described in Section 4.3.3.

More concretely, ECMP-interpose installs rules into `iptables` that match on relevant incoming and outgoing TCP traffic and relay packets via the NFQUEUE target to a user-level packet queue. For each connection, we install rules matching the TCP source port into the INPUT and OUTPUT chains in the filter table on both end hosts as described previously. `iptables` allows rules for several connections to be consolidated via range and set matches for performance. After modification, ECMP-interpose computes the new TCP checksum and relays packets back to the kernel. We deployed this prototype to two servers in the production datacenter.

Effect of rerouting. We evaluate our prototype with a simple rerouting experiment. We run a constant-rate TCP connection from a source to a destination in a different cluster in the same datacenter. At ~2 seconds, we fail the connection. When the sender gets a timeout (via `tcp_retransmit_timer()` in the Linux networking stack), ECMP-interpose automatically switches to an alternate path. Switchover was near-instantaneous. We conclude that we can successfully and transparently and selectively

change ECMP routes of live connections by interposing on these connections and modifying their port numbers. Route changes are instant and stable.

4.5.2 Volur-FL Evaluation

We evaluate Volur-FL by asking several questions:

- Can we localize different types of failures and how sensitive is our approach to the failure's drop rate?
- Can we handle multiple, potentially heterogeneous faults?
- How much does a stale view of topology affect results?

Testbed. We answer these questions using an 80-machine Cloudlab [85] testbed. The machines were interconnected via a 10 Gbps network. Each physical machine emulates either a server or a predictable software switch in a 3-level folded Clos topology. The network was implemented using a GRE-tunnel overlay [59]. The resulting topology has 12 racks with 4 servers each. The racks' 12 ToRs are split into 3 clusters with 4 aggregation switches in each. Each aggregation switch connects to two core switches, for a total of 8 core switches.

To avoid perturbation due to congestion on the underlying network, we limit the bandwidth of all emulated links to 1 Gbps using Linux's traffic shaping facilities. The links also emulate RED queues and ECN marking with threshold at 30 KB [11]. We configure Linux to use DCTCP, included in our kernel.

To collect drop statistics, we use Linux eBPF [35] with `bcc` (BPF Compiler Collection) [51]. `bcc` can monitor TCP transmits and retransmits by attaching eBPF programs to existing kernel tracepoints. Linux kernel tracks a number of TCP statistics (RFC4898), so we only need to read off relevant statistics with minimal additional overhead.

Unless stated otherwise, drop statistics were polled every 10 seconds. We use recall and precision, averaged over 50 runs, as metric for fault localization. Recall is the percentage of faults that have been predicted and precision is the percentage of predictions that are correct.

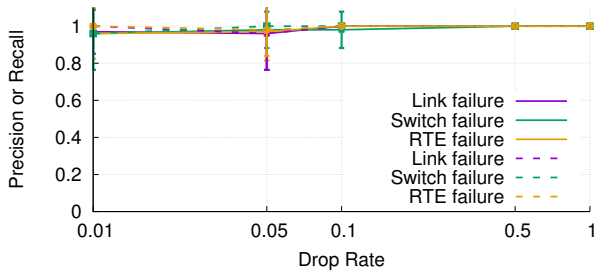


Figure 4.7: Precision and recall for a single failure and various drop rates from 1% to 100%. We considered link failures, switch failures, and failures of individual routing table entries over a window of 10 s.

Workload. We generate traffic according to a realistic workload based on empirically observed traffic patterns in deployed datacenters [11]. The web search workload is heavy-tailed: a small fraction of flows contribute most of the traffic. Flows arrive according to a Poisson process between server pairs evenly. We inject an offered load of 40% of total host access link bandwidth. In practice, ECN was effective at preventing congestion loss regardless of utilization [**?**]. Even for an offered load of 60%, drop rates remained below 0.1%, which we used as our fault localization algorithm *threshold*.

Failures. We injected failures into the network at random time while running our failure localization application in the background. The set of failures we tested were drawn from those emphasized by recent literature [71, 112, 103] and they cover the range of failure behaviors listed in Section 4.2.1. In particular, several types of components can fail silently in our testbed: links, switches, and individual routing table entries. Routing table entries are structured in a PortLand [79]-like fashion, using longest prefix matching. Failures can either be fail-stop or stochastic with some drop rate.

4.5.2.1 Localization of a Single Failure

We first evaluate our localization precision and recall for a single failure. We inject failures at either a link, switch, or routing table entry, at random location. We tested various drop rates ranging from 1% to 100%. For each failure-type, drop-rate pair, end hosts collected statistics using our custom `bcc` program. Then, using the algorithm described in Section 4.4.1.2, we implicate any number of

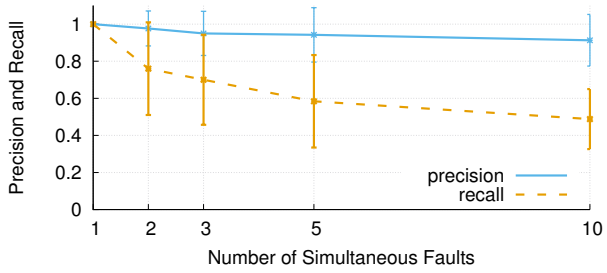


Figure 4.8: Average precision and recall for simultaneous failures in our testbed. The failures are of a random type and rate.

components. We ran the experiment 50 times for each failure-type, drop-rate pair. The mean time from failure to end host notification was 21.78 ± 1.63 seconds (much of this was due to our use of a 10 s aggregation period).

Figure 4.7 shows that, our algorithm achieves above 0.95 precision and recall. This is because Lasso's sparsity assumption is accurate when there is only a single instance of failure.

4.5.2.2 Localization of Multiple, Simultaneous Failures

Volur-FL also extends to multiple simultaneous, possibly heterogeneous, failures. We injected a random mix of failures and look at the precision and recall for our algorithm. The failures are randomly chosen: they can be link failures, switch failures, or routing table corruptions. Their drop rates are sampled uniformly between 1% and 100%.

Figure 4.8 shows the average precision and recall for up to 10 simultaneous failures. We note that for our 3-cluster, 12-rack testbed, 5 to 10 failure counts are high. Across the experiments, our system maintains a precision above 0.90 even when failure count is high. However, as the failure count increases, recall decreases quickly from 1.00 to 0.50. This shows that while our Lasso algorithm can maintain high precision, it is unable to recover all the failures when the underlying failures are no longer sparse. For example, if both a switch and a link have failed on the same path (particularly if they have similar loss rates), Lasso is likely to prefer implicating a single component rather than recovering both failures.

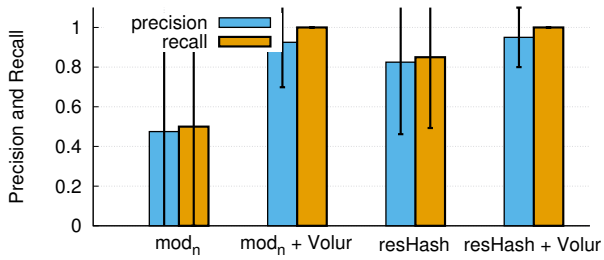


Figure 4.9: The average precision and recall for detecting a 10% switch failure in the presence of a topology change. With Volur state dissemination, precision and recall increase to above 0.92 regardless of failover scheme (mod$_n$ or resHash).

4.5.2.3 Impact of Stale State

Part of Volur's design is that switches can make routing changes on-the-fly, as long as those changes are infrequent. In this subsection, we evaluate the impact of stale state on failure localization. More specifically, we try to locate a single 10% drop rate failure in the presence of a topology-changing switch reboot.

We first configure a random aggregation or core switch (ToRs will never be mispredicted) to silently drop 10% of packets. Later, we reboot a random aggregation switch, which is properly detected/disseminated via the Volur service. Notification took up to 1 ms, but results were similar for other latencies.

For efficiency, we assume that flows are approximately constant rate and divide their traffic into pre-, mid-, and post-update proportionally. Thus, we only need to keep track of the start and end time of every flow (via tracing into `tcp_set_state`) rather than needing to store packet-level timestamps.

We evaluated two different switch failover policies with and without state dissemination. The first policy, 'mod$_n$', remaps all flows using a simple modulo function. The result is that most flows change paths after a failure. The second, resilient hashing ('resHash'), uses a simple, predictable function that limits the number of flows that need to change paths after a single failure.

Figure 4.9 shows that without resilient hashing or topology dissemination, precision and recall

falls to around 0.5, with successes limited to cases where the failures are in separate subtrees and therefore most traffic is predicted correctly. With resilient hashing, both numbers rise to above 0.80 as resilient hashing avoids remapping every flow. Thus, a large number of path predictions are still correct even with stale network state. For both failover strategies, adding topology dissemination brings precision and recall back above 0.92.

4.5.3 Volur-LB Evaluation

In this section, we evaluate the performance of Volur-LB with a 12-switch, 72-host ns-3 simulation. We show that Volur-LB achieves an average flow completion time (FCT) within 1.05x of CONGA at low to moderate load, and within 1.1x at very high load for both symmetric and asymmetric topologies.

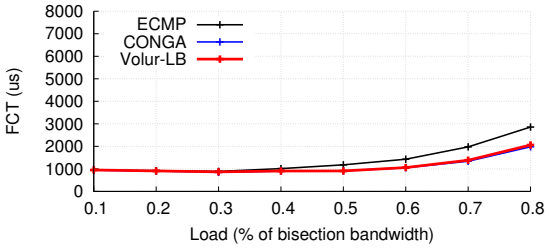
Architecture. We used a 6-leaf 6-spine topology with 10 Gbps links and 2:1 leaf oversubscription ratio. In the *symmetric* topology, all links have 10 Gbps capacity. In the *asymmetric* topology, each leaf has 2 randomly picked uplinks out of 6 uplinks with half capacity. In the worst case, a leaf to leaf path can have 4 paths out of 6 paths with only 5 Gbps capacity. The degree of asymmetry is high.

End hosts use DCTCP with no delayed ACK and 1 ms minRTO. Switch queues use RED with ECN marking, with a threshold of 65 MTU as recommended by [11]. All switch queues have 700 KB (467 MTU) buffers [11].

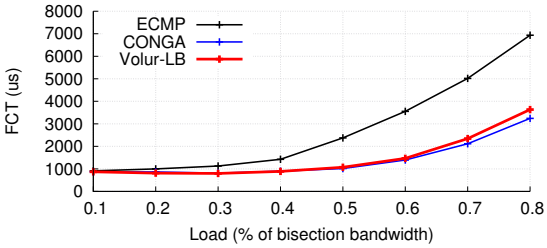
Workload. We generated flows according to the enterprise workload in [10, 99] with arrival rate to match different offered traffic load. Traffic were generated using a simple client-server program at each host. All traffic went through the spine to stress the load balancing properties of the fabric. Each client established 6 persistent TCP connections with every server.

Methodology. We use flow completion time (FCT) as evaluation metric. We average over 5 runs. We compare:

- *ECMP:* Our baseline is ECMP, in which each switch makes local, uniform-random load balancing decisions.



(a) Avg. FCT on symmetric topology



(b) Avg. FCT on asymmetric topology

Figure 4.10: Volur-LB achieves almost identical FCT as CONGA for symm topology and within 1.1x for asymm topology.

- *CONGA:* We simulate CONGA with default parameters: $Q = 3$, $\tau = 160$ µs, and flowlet timeout of 500 µs. We validated our implementation with testbed results in [10].
- *Volur-LB:* Finally, we implement Volur-LB as described in Section 4.4.2. Where applicable, we use the same configuration parameters as our CONGA implementation.

Results. Figure 4.10a shows that Volur-LB achieves almost identical average FCT to CONGA on symmetric topology for all load. Figure 4.10b shows that Volur-LB achieves FCT within 1.1x of CONGA on asymmetric topology even for load as high as 80%.

4.6 Discussion

For simplicity, we have thus far focused on typical Clos topologies and simple switch configurations. Real datacenters come in many shapes and sizes—too many to cover in detail. In this section, we point out a few interesting variations.

4.6.1 Redundant Downward Paths

One way to increase network fault tolerance at the cost of scalability is to add redundant links between root switches and their subtrees. In VL2 [50], fabric switches connect to more than one cluster switch in each subtree. This removes a fundamental weakness in traditional Clos networks: that

there is only a single path from every root to every leaf.

In Volur, this redundancy implies that we must be able to steer traffic in the downward direction—something that the design of the previous section cannot accomplish because with the Trident 2, switches use the same hash keys and hash functions on the upward direction and the downward direction. This clashes with our goal of allowing independent control over every ECMP decision in the network.

To enable switches to hash two different packets differently, we use encapsulation. The original header is used to control the downward path while the outer header is used to control the upward path. The last switch on the upward path is responsible for decapsulating the packet.

An easy way to implement this would be to use an MPLS header. Today's switches already have the ability to hash on and decapsulate MPLS labels. Sources would simply encapsulate the original packet with an MPLS label that resulted in the correct ECMP decisions on all switches on the upward path.

More generally, encapsulation allows us to stitch a theoretically arbitrary number of path segments together. See Section 4.2.2 for a brief discussion of the limits of deep MPLS encapsulation.

4.6.2 Arbitrary Topologies

Unstructured topologies are not amenable to any clean division of hash groups. Unfortunately, it is these topologies that need Volur the most—routing table scalability is one of the primary challenges in these types of networks. For these topologies, we introduce two tools:

- *Encapsulation:* The mechanism described in the previous section works as long as no paths require more than one ECMP decision in any given hash group. We henceforth refer to paths that satisfy this "one decision per group" property as path segments. Encapsulation allows us to stitch path segments together.
- *Offline Checking:* For random or otherwise unstructured topologies, division of switches into hash groups is exceedingly difficult. For these random networks, we propose to do random assignment of hash groups with a offline check to verify the quality of the random assignment.

We therefore adapt Volur to random topologies by first randomly assign bit ranges and hash seeds to switches. Assuming sufficient randomness, a good hash function and a large enough set of usable header bits (i.e., address space), many useful between a given source and destination should be addressable.

To verify that the hash settings indeed provide good connectivity without encapsulation, we add a verification step. When designing the topology offline, we manually check the shortest N edge-disjoint paths between every source and every destination ToR. These paths will provide a few good load-balancing options. Verification of a path involves solving a system of n linear equations, where n is the number of switches on the path (which should be small given that we are looking for shortest paths). The complexity of the entire operation is then $O(Nr^2n^3)$, where r is the number of ToR switches and we use simple Gaussian elimination to solve the system of equations. Note again, that this is done entirely offline and only on physical expansion of the network.

Failures can cause some of these paths to become unusable and can even cause otherwise correct paths to become unroutable due to changes in the modulo divisor. We handle this by dynamically choosing the next shortest path and/or using encapsulation to route along previously unroutable paths.

4.6.3 Resilient Hashing

In a traditional switch, when a failure occurs, the number of next hop entries in the routing table may change, causing the hashing function (or more specifically, the modulus step at the end) to change as well. This in turn causes widespread rebinding of almost all flows—even those on ports that are unaffected by the failure.

To avoid rebinding, some recent switches have implemented a feature called resilient hashing, which attempts to avoid as much rebinding as possible. It does so by leveraging a separate 'flow table'. When a new flow arrives, the hashing mechanism will create an entry in the flow table that maps the flow to a physical egress port. Whenever a packet arrives, the switch will check the flow table before recomputing the hash value. Thus, when a failure occurs, the switch will continue to use the cached entry for all flows that are unaffected by the failure. Flows on failed links are rebound to the

surviving links. Similarly, when a new link is activated, resilient hashing will attempt to minimize the number of rebound flows. Specifically, it will choose a number of existing flows to rebind, but not touch any other flows.

In the context of Volur, this feature is both a positive and a negative. On one hand, it means that topology changes cause significantly less path uncertainty. On the other, some flows that are not even on the changed path may be moved permanently. This is another instance where further cooperation from switch ASIC manufacturers would be helpful. If we know the algorithm for rebinding, it could be included in the path deduction and control algorithms. The other alternative is to disable this feature.

4.7 Summary

In this chapter, we have dealt with the challenge opaque routing presents for fine-grained failure localization in datacenter networks. Rather than proposing another workaround that is likely fragile in the facing of evolving networks, Volur explores an alternative approach where we make predictability an architectural requirement. One of our observations is that pure source routing is not needed. Instead, switches are allowed to make routing decisions in the middle of the network as long as they are based only on the current packet header and infrequently changing state that is eventually visible to endpoints.

Our results have demonstrated that failure handling becomes much simpler in a predictable network. In addition, most existing sources of routing complexity can fit into our model—we have verified this by building a test deployment on a large, unmodified production datacenter network. For in-network load balancing, which is not allowed in our model, we have shown that we can achieve similar results using an end-host-based approach.

Chapter 5
Conclusion

Datacenter applications are increasingly the key underpinning of many Internet and computing services we use in our everyday life. They are an important class of applications whose performance and reliability we must constantly seek to improve. In this dissertation, I identified three challenges that make failure diagnosis for datacenter applications difficult, namely complex component dependency, gray failures, and unpredictable component behaviors. I demonstrated that these challenges can be addressed by (1) a global view, (2) a penalized-regression-based algorithm, and (3) a predictable component architecture. To evaluate the thesis that *fast and accurate failure diagnosis for datacenter applications is possible*, I presented two systems: Deepview which can localize virtual hard disk failures in IaaS in a fast and accurate manner, and Volur which shows that with a predictable datacenter network routing controllable by end-hosts, fine-grained failure localization and fast failure recovery are possible.

There are a few limitations with this work and multiple future directions:

Robustness of the Statistical Techniques to Multiple Failures. While the penalized-regression-based algorithm is shown to achieve high precision and recall for both VHD failures and network failures when the failure count is low, it suffers low recall when the underlying failures are no longer sparse. It remains an open question how to generalize the penalized-regression-based localization algorithm when many concurrent failures are present. Bayesian networks is a powerful technique for failure localization. While effective, they are often too slow for production use and thus warrant more studies to apply in real practical use cases.

Extensions to Failure Prediction. While we have focused on diagnosing failures after they occur, it may be possible to extend our techniques to predict failures before they happen. For example, gray

failures in storage clusters often show signs that herald their onset: the number of failed VMs that use the failed storage cluster slowly creeps up. In some of those cases, it may be possible to apply machine learning techniques to predict the component failures before they cascade to wider-spread outages.

Generalization to Other Applications. We studied failure diagnosis just for VHD failures. It remains to be seen whether the approach proposed can work for other applications, and whether it can help achieve greater levels of nines than five-nines.

Generalization to More Diverse Failures Scenarios. Gray failures in general remain poorly understood. They include not just reliability issues but also performance ones. For example, the datacenter networks may fail in ways more than dropping packets but also delaying packets. One interesting future direction is to use the proposed techniques to diagnose performance issues in addition to reliability ones.

Handling Future In-Network Features. The predictable network architecture I proposed precludes a class of in-network features that are inherently unpredictable at end-hosts. Notable examples include load balancing based on local queue length and local mechanisms enabled by programmable switches [90]. While I showed that Volur can approach CONGA for load balancing performance, it remains to be seen if we can emulate future in-network features that operator may desire. We need to either explore whether practical endpoint solutions exist that deliver the same features, or improve these in-network technologies to make them diagnosable too. It may be the case that neither direction works out for some of those features, then network operators need to decide whether to use these in-network features.

Bibliography

[1] Amazon EC2 Root Device Volume. http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/RootDeviceStorage.html#RootDeviceStorageConcepts.

[2] Azure Kusto (Preview). https://docs.microsoft.com/en-us/connectors/kusto/.

[3] Introducing Application Insights Analytics. https://blogs.msdn.microsoft.com/bharry/2016/03/28/introducing-application-analytics/.

[4] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. B. Zdonik. Aurora: a New Model and Architecture for Data Stream Management. *The VLDB Journal*, 12:120–139, 2003.

[5] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. A. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI*, 2016.

[6] A. Adams, P. Lapukhov, and J. H. Zeng. NetNORAD: Troubleshooting Networks via End-to-end Probing. https://code.facebook.com/posts/1534350660228025/netnorad-troubleshooting-networks-via-end-to-end-probing/, Febrary 2016.

[7] A. Adya, G. Cooper, D. Myers, and M. Piatek. Thialfi: a Client Notification Service for Internet-scale Applications. In *SOSP*, 2011.

[8] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. In *SIGCOMM*, 2008.

[9] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *NSDI*, 2010.

[10] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav, and G. Varghese. CONGA: Distributed Congestion-aware Load Balancing for Datacenters. In *SIGCOMM*, 2014.

[11] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data Center TCP (DCTCP). In *SIGCOMM*, 2010.

[12] A. Andreyev. Introducing Data Center Fabric, the Next-generation Facebook Data Center Network. https://code.facebook.com, Nov. 2014.

[13] B. Arzani, S. Ciraci, B. T. Loo, A. Schuster, and G. Outhred. Taking the Blame Game Out of Data Centers Operations with NetPoirot. In *SIGCOMM*, 2016.

[14] P. Bahl, R. Chandra, A. Greenberg, S. Kandula, D. A. Maltz, and M. Zhang. Towards Highly Reliable Enterprise Network Services via Inference of Multi-level Dependencies. In *SIGCOMM*, 2007.

[15] P. Bailis and K. Kingsbury. The Network is Reliable. *Commun. ACM*, 2014.

[16] M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker. Fault-tolerance in the Borealis Distributed Stream Processing System. In *SIGMOD Conference*, 2005.

[17] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. L. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *SOSP*, 2003.

[18] L. A. Barroso, J. Clidaras, and U. Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition*. 2013.

[19] Y. Benjamini and Y. Hochberg. Controlling the False Discovery Rate: a Practical and Powerful Approach to Multiple Testing. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 289–300, 1995.

[20] B. Beyer, C. Jones, J. Petoff, and N. R. Murphy. *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, 2016.

[21] P. J. Bickel, Y. Ritov, and A. B. Tsybakov. Simultaneous Analysis of Lasso and Dantzig Selector. *The Annals of Statistics*, pages 1705–1732, 2009.

[22] A. Birrell and B. J. Nelson. Implementing Remote Procedure Calls. *ACM Trans. Comput. Syst.*, 1984.

[23] T. Bishop. Microsoft Says Google's Cloud Reliability Claim vs. Azure and Amazon Web Services Does Not Compute, 2017. `https://www.geekwire.com/2017/microsoft-says-googles-cloud-reliability-claim-vs-azure-amazon-web-services-not-compute`.

[24] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming Protocol-independent Packet Processors. *SIGCOMM CCR*, 44(3):87–95, July 2014.

[25] P. Bratach and P. Lumbis. Equal Cost Multipath Load Sharing - Hardware ECMP, 2017. `https://docs.cumulusnetworks.com/display/DOCS/Equal+Cost+Multipath+Load+Sharing+-+Hardware+ECMP`.

[26] M. Burrows. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *OSDI*, 2006.

[27] B. Calder et al. Windows Azure Storage: a Highly Available Cloud Storage Service with Strong Consistency. In *SOSP*, 2011.

[28] J. Case, R. Mundy, D. Partain, and B. Stewart. Introduction and Applicability Statements for Internet-Standard Management Framework. RFC 3410, December 2002.

[29] G. Casella and R. L. Berger. *Statistical Inference*, volume 2. Duxbury Pacific Grove, CA, 2002.

[30] R. Castro, M. Coates, G. Liang, R. Nowak, and B. Yu. Network Tomography: Recent Developments. *Statistical Science*, 19, Auguest 2004.

[31] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. A. Brewer. Pinpoint: Problem Determination in Large, Dynamic Internet Services. In *DSN*, 2002.

[32] M. Y. Chen, A. X. Zheng, J. Lloyd, M. I. Jordan, and E. A. Brewer. Failure Diagnosis Using Decision Trees. pages 36–43, 2004.

[33] B. Claise. Cisco Systems NetFlow Services Export Version 9. RFC 3954, October 2004.

[34] C. Clos. A Study of Non-blocking Switching Networks. *The Bell System Technical Journal*, 32(2):406–424, March 1953.

[35] J. Corbet. Extending extended BPF, 2014. `https://lwn.net/Articles/603983/`.

[36] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Google's Globally Distributed Database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[37] M. Davies. Traffic Distribution Techniques Utilizing Initial and Scrambled Hash Values, Oct. 26 2010. US Patent 7,821,925.

[38] J. Dean. Designs, Lessons and Advice From Building Large Distributed Systems. *Keynote from LADIS*, 1, 2009.

[39] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. W. Senior, P. A. Tucker, K. Yang, and A. Y. Ng. Large Scale Distributed Deep Networks. In *NIPS*, 2012.

[40] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *SOSP*, 2007.

[41] A. Dhamdhere, R. Teixeira, C. Dovrolis, and C. Diot. NetDiagnoser: Troubleshooting Network Unreachabilities Using End-to-end Probes and Routing Data. In *CoNEXT*, 2007.

[42] N. G. Duffield. Network Tomography of Binary Network Performance Characteristics. *IEEE Transactions on Information Theory*, 52:5373–5388, 2006.

[43] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu, B. Cheyney, W. Shang, and J. D. Hosein. Maglev: A Fast and Reliable Software Network Load Balancer. In *NSDI*, 2016.

[44] C. Estan, K. Keys, D. Moore, and G. Varghese. Building a Better NetFlow. In *SIGCOMM*, 2004.

[45] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in Globally Distributed Storage Systems. In *OSDI*, 2010.

[46] J. Friedman, T. Hastie, and R. Tibshirani. Regularization Paths for Generalized Linear Models via Coordinate Descent. *Journal of statistical software*, 33(1):1, 2010.

[47] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *SOSP*, 2003.

[48] S. Ghorbani, B. Godfrey, Y. Ganjali, and A. Firoozshahian. Micro Load Balancing in Data Centers with DRILL. In *HotNets*, 2015.

[49] R. Govindan, I. Minei, M. Kallahalla, B. Koley, and A. Vahdat. Evolve or Die: High-Availability Design Principles Drawn From Google's Network Infrastructure. In *SIGCOMM*, 2016.

[50] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: A Scalable and Flexible Data Center Network. In *SIGCOMM*, 2009.

[51] B. Gregg. BCC: Dynamic Tracing Tools for Linux, 2017. `https://iovisor.github.io/bcc/`.

[52] C. Guo et al. Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis. In *SIGCOMM*, 2015.

[53] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown. I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks. In *NSDI*, 2014.

[54] T. J. Hastie, R. Tibshirani, and J. H. Friedman. The Elements of Statistical Learning: Data Mining, Inference, and Prediction, 2nd Edition. In *Springer series in statistics*, 2009.

[55] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *NSDI*, 2011.

[56] C. Hopps. Analysis of an Equal-Cost Multi-Path Algorithm. RFC 2992, November 2000.

[57] S. Hu, K. Chen, H. Wu, W. Bai, C. Lan, H. Wang, H. Zhao, and C. Guo. Explicit path control in commodity data centers: Design and applications. In *NSDI*, 2015.

[58] P. Huang, C. Guo, L. Zhou, J. R. Lorch, Y. Dang, M. Chintalapati, and R. Yao. Gray Failure: The Achilles' Heel of Cloud-Scale Systems. In *HotOS*, 2017.

[59] B. Hubert. GRE and Other Tunnels, 2017. `http://lartc.org/howto/lartc.tunnel.gre.html`.

[60] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *USENIX Annual Technical Conference*, 2010.

[61] A. Kabbani, B. Vamanan, J. Hasan, and F. Duchene. FlowBender: Flow-level adaptive routing for improved latency and throughput in datacenter networks. In *CoNEXT*, 2014.

[62] M. Kalkunte. High Speed Trunking in a Network Device, Mar. 16 2010. US Patent 7,680,107.

[63] S. Kandula, D. Katabi, and J.-P. Vasseur. Shrink: a Tool for Failure Diagnosis in IP Networks. In *MineNet*, 2005.

[64] S. Kandula, R. Mahajan, P. Verkaik, S. Agarwal, J. Padhye, and P. Bahl. Detailed Diagnosis in Enterprise Networks. In *SIGCOMM*, 2009.

[65] D. Katz and D. Ward. Bidirectional Forwarding Detection (BFD). RFC 5880, June 2010.

[66] D. Koller and N. Friedman. Probabilistic Graphical Models - Principles and Techniques. 2009.

[67] R. R. Kompella, J. Yates, A. Greenberg, and A. C. Snoeren. IP Fault Localization via Risk Modeling. In *NSDI*, 2005.

[68] A. Lê-Quôc. Learning from AWS' gray failures. `https://www.datadoghq.com/blog/gray-aws-failures/`, October 2013.

[69] G. Leopold. AWS Rates Highest on Cloud Reliability, 2015. `https://www.enterprisetech.com/2015/01/06/aws-rates-highest-cloud-reliability`.

[70] Y. Li, R. Miao, C. Kim, and M. Yu. FlowRadar: A Better NetFlow for Data Centers. In *NSDI*, 2011.

[71] V. Liu, D. Halperin, A. Krishnamurthy, and T. E. Anderson. F10: A Fault-Tolerant Engineered Network. In *NSDI*, 2013.

[72] V. Liu, D. Zhuo, S. Peter, A. Krishnamurthy, and T. Anderson. Subways: A Case for Redundant, Inexpensive Data Center Edge Links. In *CoNEXT*, 2015.

[73] B. Matthews, B. Kwan, and P. Agarwal. Dynamic load balancing, Jan. 15 2013. US Patent 8,355,328.

[74] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling Innovation in Campus Networks. In *SIGCOMM*, 2008.

[75] Mike Y. Chen and Anthony Accardi and Emre Kiciman and Jim Lloyd and Dave Patterson and Armando Fox and Eric Brewer. Path-Based Failure and Evolution Management. In *NSDI*, 2004.

[76] J. C. Mogul, R. Isaacs, and B. Welch. Thinking About Availability in Large Service Infrastructures. In *HotOS*, 2017.

[77] R. N. Mysore, R. Mahajan, A. Vahdat, and G. Varghese. Gestalt: Fast, Unified Fault Localization for Networked Systems. In *USENIX ATC*, 2014.

[78] J. Networks, 2016. `https://www.juniper.net/documentation/en_US/junos/topics/concept/load-balance-technique-overview.html`.

[79] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. PortLand: a Scalable Fault-tolerant Layer 2 Data Center Network Fabric. In *SIGCOMM*, 2009.

[80] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling Memcache at Facebook. In *NSDI*, 2013.

[81] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. 1988.

[82] P. Phaal, S. Panchen, and N. McKee. InMon Corporation's sFlow: A Method for Monitoring Traffic in Switched and Routed Networks. RFC 3176, September 2001.

[83] J. C. Platt, E. Kiciman, and D. A. Maltz. Fast Variational Inference for Large-scale Internet Diagnosis. In *NIPS*, 2007.

[84] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley. Improving datacenter performance and robustness with multipath TCP. In *SIGCOMM*, 2011.

[85] R. Ricci, E. Eide, and The CloudLab Team. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *USENIX ;login:*, 39(6), Dec. 2014.

[86] E. Rosen, A. Viswanathan, and R. Callon. Multiprotocol Label Switching Architecture. RFC 3031, January 2001.

[87] A. Roy, H. Zeng, J. Bagga, and A. C. Snoeren. Passive Realtime Datacenter Fault Detection and Localization. In *NSDI*, 2017.

[88] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: Flexible, Scalable Schedulers for Large Compute Clusters. In *EuroSys*, 2013.

[89] S. Sen, D. Shue, S. Ihm, and M. J. Freedman. Scalable, Optimal Flow Routing in Datacenters via Local Link Balancing. In *CoNEXT*, 2013.

[90] N. K. Sharma, M. Liu, K. Atreya, and A. Krishnamurthy. Approximating Fair Queueing on Reconfigurable Switches. In *NSDI*, 2018.

[91] A. Singh and et al. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *SIGCOMM*, 2015.

[92] M. Steinder and A. S. Sethi. End-to-end Service Failure Diagnosis using Belief Networks. In *NOMS*, 2002.

[93] M. Subramanian, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, S. Viswanathan, L. Tang, and S. Kumar. f4: Facebook's Warm BLOB Storage System. In *OSDI*, 2014.

[94] P. Tammana, R. Agarwal, and M. Lee. Simplifying Datacenter Network Debugging with PathDump. In *OSDI*, 2016.

[95] D. Thaler and C. Hopps. Multipath Issues in Unicast and Multicast Next-Hop Selection. RFC 2991, November 2000.

[96] R. Tibshirani. Regression Shrinkage and Selection via the Lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 267–288, 1996.

[97] B. Treynor, M. Dahlin, V. Rau, and B. Beyer. The Calculus of Service Availability. *ACM Queue*, 15, 2017.

[98] R. van Renesse and D. Altinbüken. Paxos Made Moderately Complex. *ACM Comput. Surv.*, 47:42:1–42:36, 2015.

[99] E. Vanini, R. Pan, M. Alizadeh, T. Edsall, and P. Taheri. Let it Flow: Resilient Asymmetric Load Balancing with Flowlet Switching. In *NSDI*, 2017.

[100] P. Viswav. Microsoft Dismisses Google's Cloud Reliability Claim, 2017. `https://mspoweruser.com/microsoft-dismisses-googles-cloud-reliability-claim`.

[101] C. A. Waldspurger. Memory Resource Management in VMware ESX Server. In *OSDI*, 2002.

[102] C. Widanapathirana, J. C. Li, Y. A. Sekercioglu, M. V. Ivanovich, and P. G. Fitzpatrick. Intelligent Automated Diagnosis of Client Device Bottlenecks in Private Clouds. *2011 Fourth IEEE International Conference on Utility and Cloud Computing*, pages 261–266, 2011.

[103] X. Wu, D. Turner, G. Chen, D. Maltz, X. Yang, L. Yuan, and M. Zhang. NetPilot: Automating Datacenter Network Failure Mitigation. In *SIGCOMM*, 2012.

[104] M. Yu, A. G. Greenberg, D. A. Maltz, J. Rexford, L. Yuan, S. Kandula, and C. Kim. Profiling Network Performance for Multi-tier Data Center Applications. In *NSDI*, 2011.

[105] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized Streams: Fault-tolerant Streaming Computation at Scale. In *SOSP*, 2013.

[106] H. Zhang, J. Zhang, W. Bai, K. Chen, and M. Chowdhury. Resilient Datacenter Load Balancing in the Wild. In *SIGCOMM*, 2017.

[107] Q. Zhang, G. Yu, C. Guo, Y. Dang, N. Swanson, X. Yang, R. Yao, M. Chintalapati, A. Krishnamurthy, and T. E. Anderson. Deepview: Virtual Disk Failure Diagnosis and Pattern Detection for Azure. In *NSDI*, 2018.

[108] Q. Zhang, D. Zhuo, V. Liu, P. Lapukhov, S. Peter, A. Krishnamurthy, and T. Anderson. Volur: Concurrent Edge/Core Route Control in Data Center Networks. *arXiv preprint arXiv:1804.06945*, 2018.

[109] S. Zhang, I. L. Cohen, M. Goldszmidt, J. Symons, and A. Fox. Ensembles of Models for Automated Diagnosis of System Performance Problems. *2005 International Conference on Dependable Systems and Networks (DSN'05)*, pages 644–653, 2005.

[110] S. Zhao, A. Shojaie, and D. Witten. In Defense of the Indefensible: A Very Naive Approach to High-Dimensional Inference. *arXiv preprint arXiv:1705.05543*, 2017.

[111] J. Zhou, M. Tewari, M. Zhu, A. Kabbani, L. Poutievski, A. Singh, and A. Vahdat. WCMP: Weighted Cost Multipathing for Improved Fairness in Data Centers. In *EuroSys*, 2014.

[112] Y. Zhu and et al. Packet-Level Telemetry in Large Datacenter Networks. In *SIGCOMM*, 2015.

Appendix 1

P-value Correction for Multiple Testing

A.1 Interpretation of p-values

We construct t-test statistics as in (3.7) for each j for $j = 1, ..., N$, where N is the total number of components of interest. Given the form of the alternative hypothesis, intuitively the smaller (greater in absolute value but with negative sign) the test statistics is, the more likely the null hypothesis would be rejected for some significance levels.

To make decisions without setting an ad-hoc threshold, we compute p-values for each test. The p-value is the probability, under the null hypothesis, of the sampling test statistic having a value at least as extreme as that which was observed, i.e., as computed in (3.7). If the null hypothesis is true, we should expect the probability of observing test statistic (3.7) to be moderate. The smaller the p-value is, the more confident we are to say that the null hypothesis under consideration is not adequate to explain the observations. In other words, if the p-value associated with a certain test statistic (3.7) is extremely small, we should reject the null hypothesis $H_0(j)$, under which it is almost impossible to observe what we have observed.

If the p-value is not smaller than the pre-specified threshold, however, the test has no result. Instead of confidently saying that the component of interest remains healthy (as claimed in the null hypothesis), we should interpret the testing procedure as that the evidence is not adequate to support any conclusion. In particular, extra consideration should be given to those components whose p-values are close to (but not smaller than) the significance level. For example, we produce warnings with lower priority for those components whose p-values are within a certain range of values that are greater than the significance level. Such an adjustment should decrease the false negative rate.

The question of choosing the appropriate significance level remains. In other words: how do

we characterize that a certain p-value is "extremely small". For testing a single hypothesis, common choices of significance level include 1%, 5%, and 10%.

However, when testing multiple hypothesis, things become complicated. Consider the following textbook example, where we are testing 100 null hypothesis, all of which are true. If we use 5% as significance level, then there is roughly 5% probability that we incorrectly reject the null hypothesis. So the expected number of false positives is 5. Moreover, if all these 100 tests are independent, then we have that

$$\begin{aligned}\mathbb{P}\left(\text{at least one false positive}\right) &= 1 - \mathbb{P}\left(\text{no false positive}\right) \\ &= 1 - 0.95^{100} = 0.994. \end{aligned} \tag{A.1}$$

In other words, we are almost certain that at least one false positive will be made. Intuitively, the more hypothesis are tested simultaneously, the more likely that erroneous inferences are to happen. So for testing multiple hypothesis, we need to provide a stricter significance level than a single test, in order to compensate for the tendency of making error due to the number of simultaneous testings. This process is often called multiple testing correction, and has gained particular notice in research in the past several years.

A.2 Family-wise Error Rate Correction

Essentially there are two types of approaches to multiple testing correction, namely family-wise error rate (FWER) control correction and false discovery rate (FDR) control correction.

FWER is defined as the probability of making at least one false discovery (false positive), i.e., $\mathbb{P}(\text{at least one false positive}) = 1 - \mathbb{P}(\text{no false positive})$. The calculation we did in (A.1) is for FWER. So if we ensure $FWER \leq \alpha$, then we've control the probability of making one or more false positive below α.

A naive but widely used FWER control correction is Bonferroni correction. Consider a setting where we are performing N tests, and denote the p-value for the i-th test as P_i, then reject $H_0(i)$ if $P_i < \frac{\alpha}{N}$. To see that this rule will control the FWER below α, denote M_0 as the index set of the true

null hypothesis, note that

$$\begin{aligned} FWER =& \mathbb{P}\left(\bigcup_{i \in M_0} \{\text{reject } H_0(i)\} \right) \\ & \leq \sum_{i \in M_0} \mathbb{P}\left(p_i < \frac{\alpha}{m} \mid H_0(i) \text{ is true} \right) \leq |M_0| \frac{\alpha}{N} \leq \alpha, \end{aligned}$$

where the first inequality is a union bound, and the second inequality uses the fact that the p-values, when the underlying null hypothesis is true, follows a uniform distribution on $[0, 1]$.

The main criticism about Bonferroni correction, or in more general sense, FWER correction, is that the correction is too stringent. In other words, it usually produces a much smaller threshold, which means that more false negative could be made and the statistical power is decreased.

A.3 False Discovery Rate Correction

FDR control has been particularly influential in recent years, as it gives a more powerful alternative (of course, with higher Type-I error) to the FWER control methods.

Denote V to be the number of false positive (i.e., Type-I error, or in plain words, the healthy components that we falsely blame), and R to be the total number of rejected hypothesis (i.e., the total number of blamed components). Then the false discovery rate (FDR) is defined as

$$FDR := \mathrm{E}[Q] := \mathrm{E}\left[V/R\right] \tag{A.2}$$

The seminal contribution of Benjamini-Hochberg procedure [19] is the most popular FDR control procedure due to its simplicity and effectiveness. The procedure as follows:

1. Do N individual testings and get p-values P_1, P_2, …, P_N corresponding to null hypothesis $H_0(1), H_0(2), \ldots, H_0(N)$.
2. Sort p-values in ascending order and denote them by $P_{(1)}, P_{(2)}, \cdots, P_{(N)}$.
3. For a given threshold on FDR α, find the largest K such that $P_{(K)} \leq \frac{K}{N}\alpha$.
4. Reject all null hypothesis of which their p-values are smaller than or equal to $P_{(K)}$.