UNIVERSITY OF CALIFORNIA, SAN DIEGO

Automated Scalable Management of Data Center Networks

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Computer Science

by

Radhika Niranjan Mysore

Committee in charge:

Professor Amin Vahdat, Chair
Professor Yeshaiahu Fainman
Professor George Papen
Professor Alex Snoeren
Professor Geoff Voelker

2013

The dissertation of Radhika Niranjan Mysore is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

_____

_____

_____

_____

_____

Chair

University of California, San Diego

2013

DEDICATION

To my family, my strength.

EPIGRAPH

When you have reached your goal with your own efforts,
What is there to fear the future?
—Rekha Niranjan (mom)

Your goals are closer than they appear to your mind.
—Niranjan Kumar (dad)

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

ACKNOWLEDGEMENTS

I remain unapologetic for the length of this section. If anything, it should be longer and is limited only by my memory.

I can safely confess now that I might not have had a PhD career but for Amin Vahdat, my advisor. I find it quite impossible to thank him in a short paragraph. Amin has shaped my taste for practical problems and taught me critical reasoning. He inspires his students to think deeply about problems in systems and networking and present ideas in a coherent manner, because of the kind of researcher he is. He can easily navigate a ten thousand foot view while not losing sight of the smallest details. He has taught me to be fearless in research. But beyond this, he has been an enormous positive influence. His core values of consistency, pursuit of truth and always doing the right thing have become my values. For this and other positive influences I remain deeply indebted.

I have been incredibly lucky to have Ratul Mahajan as a mentor and as an unofficial co-advisor for the greater part of my PhD. He has taught me to be open to new ideas and listen with an unbiased mind. He has inculcated scientific curiosity in me, taught me the importance of being systematic in the research process and the importance of simplicity in writing. I have learned to become comfortable with uncertainty inherent to research (in Ratul's words:'embrace it'), and approach problems with fewer preconceived expectations (I'm still working on it). It has been a joy working with him and I am indebted to him for being such a wonderful role-model.

I've had the privilege of working with and learning from George Varghese in my PhD career. The childlike curiosity and sheer happiness he exudes are infectious. He has taught me by example to be uninhibited in learning, that simple questions might give the deepest answers. Among other things, I've had the opportunity to learn and be touched by his qualities of humility, thoughtfulness, of frequently going above and beyond the call of duty and the funny ability to convincingly give others credit for his ideas.

Keith Marzullo has nurtured me with kid gloves during the early part of my graduate career. I am thankful to him for giving me the confidence to collaborate on theoretical aspects of systems research.

John Dunagan has constantly pushed me to communicate precisely and encouraged me. I feel lucky to have a mentor and friend in him.

I am thankful to have found a wonderful collaborator, friend and mentor in Meg Walraed-Sullivan. I cherish our long conversations and look forward to many more.

I would also like to thank Elaine Weyuker for being a patient, loving mentor through my graduate career and for inspiring me in so many ways.

I have been fortunate to work with the most amazing collaborators, and will always cherish the long but invigorating work days with Andreas Pamboris, Sivasankar Radhakrishnan and Nelson Huang. I'd also like to thank friends and collaborators: Mohammad Al-Fares, John McCullough, Patrick Verkaik, Nathan Farrington, Malveeka Tewari, Alex Rasmussen, Mike Conley, Alex Pucher, Harsha Madhyastha, Vikram Subramanya for being an important part of my time in graduate school.

I'd like to thank the sysnet faculty: Geoff Voelker, Stefan Savage, Alex Snoeren and George Porter for their timely advice and for being wonderful role-models. I feel very fortunate to have had numerous opportunities to learn by observing them. I am also thankful to George Papen and Yeshaiahu Fainman for serving on my committee. Their feedback has contributed to this dissertation.

I have had a fantastic time interning at Microsoft Research and would like to thank Srikanth Kandula, Ming Zhang, Sharad Agarwal and Victor Bahl for their role in making it a wonderful experience.

I must thank Karsten Schwan, Ada Gavrilovska and Brian Cooper for supporting me and encouraging me to pursue a PhD in Computer Science.

I feel constantly blessed to have the infinite love and unwavering support of my parents Rekha Niranjan and Niranjan Kumar and my husband Gopal Pai. Their approach to life and work, sincerity, ability to constantly grow inspire and amaze me. My husband has patiently nurtured me through this journey and has inspired my focus and creativity. My father has frequently surprised me by learning technical details behind the buzz words 'cloud computing' and has often read my papers end-to-end to give me valuable feedback right before submission deadlines. My mother has inspired me by example, by completing her PhD during difficult times. She has been a constant source of moral support and is my best friend. I feel incredibly lucky to have a funny and inspiring, brilliant brother, Adithya Niranjan. I cherish his knack of always making me laugh in any situation. Together, my family takes much of the credit for this dissertation and

successful completion of my graduate career. Just like them, I plan to continue learning and being amazed by this wonderful life.

Finally, I would like to thank all my friends for bringing cheer and wisdom in my life. I would also like to thank the ever helpful staff at UC San Diego particularly Julie Conner, Jennifer Folkestad, Kathy Krane, Brian Kantor, Cindy Moore and Paul Terry; and UC San Diego itself for giving me the most wonderful experiences and freedom to pursue my interests.

Chapters 1 and 2, in part, contain material as it appears in the Proceedings of the ACM SIGCOMM 2009 conference on Data communication. Niranjan Mysore, Radhika; Pamboris, Andreas; Farrington, Nathan; Huang, Nelson; Miri, Pardis; Radhakrishnan, Sivasankar; Subramanya, Vikream; Vahdat, Amin. The dissertation author was the primary investigator and author of this paper.

Chapters 1 and 3, in part, contain material submitted for publication as "Fastrak: Enabling express express lanes in multi-tenant data centers" Niranjan Mysore, Radhika; Porter, George; Vahdat, Amin. The dissertation author was the primary investigator and author of this paper.

Chapters 1 and 4, in part, contain material submitted for publication as "Gestalt: Unifying fault localization for networked systems" Niranjan Mysore, Radhika; Mahajan, Ratul; Vahdat, Amin; Varghese, George. The dissertation author was the primary investigator and author of this paper.

VITA

2003	B. S. in Computer Science Sri Jayachamarajendra College of Engineering
2007	M. S. in Electrical and Computer Engineering Georgia Institute of Technology
2013	Ph. D. in Computer Science University of California San Diego

ABSTRACT OF THE DISSERTATION

Automated Scalable Management of Data Center Networks

by

Radhika Niranjan Mysore

Doctor of Philosophy in Computer Science

University of California, San Diego, 2013

Professor Amin Vahdat, Chair

Data centers today are growing in size and becoming harder to manage. It is more important than ever to concentrate on management of such large networks, and arrive at simple yet efficient designs that involve minimum manual intervention. Reducing network management costs can lead to better service availability, response times and increase return on investment. In this dissertation, we focus on three aspects of data center network management, the network fabric, policy enforcement and fault localization.

There are inherent challenges due to scale in each of these areas. Firstly, simple, plug-and-play networks are known not to scale, leading network operators to often stitch complex interior and exterior gateway protocols to connect large data centers. Second, network isolation policies can become too huge for network hardware to handle as the number of applications multiplexed on a single data center increase. Thirdly diagnosis

can become extremely hard because of the sheer number of components interacting for a service to be successful. Localizing the fault is often left to knowledgeable operators who work together in war rooms to track down and fight problems. Such an approach can be time consuming, tedious and reduce availability.

To address these challenges, we propose to compose the data center management system with these three contributions:

(i) PortLand: A scalable layer 2 network fabric that completely eliminates loops and broadcast storms and combines the best elements of traditional layer 2 and layer 3 network fabrics: plug-and-play, support for scale, mobility and path diversity.

(ii) FasTrak: A policy enforcement system that moves network isolation rules between server software and network hardware so that performance sensitive traffic is not subject to unnecessary overheads and latency. FasTrak enables performance sensitive applications to move into multi-tenant clouds and supports their requirements.

(iii) Gestalt: A fault localization algorithm, developed from first principles, that can operate in large scale networks and beats existing localization algorithms on localization accuracy or time or both.

We have prototyped and evaluated each of these systems and believe that these can be easily implemented with minor modifications to data center switches and end hosts.

Chapter 1

Introduction

There is an increasing trend toward migrating applications, computation and storage into data centers spread across the Internet. Benefits from commodities of scale are leading to the emergence of "mega data centers" hosting applications running on tens of thousands of servers [meg]. For instance, a web search request may access an inverted index spread across 1,000+ servers, and data storage and analysis applications may interactively process petabytes of information stored on thousands of machines. Data centers networks usually host multiple such applications having unique network demands.

These data center networks represent a new class of networks differentiated by scale and the strict, often conflicting requirements of quality of service, isolation and agility that applications place on them. Management and control of such networks is hard. Like the Internet, these networks are large, and it is desirable that they be easy to configure. Routing should be efficient and additionally provide visibility for ease of diagnosis and repair. Mobility of applications is common, so data center network protocols must be designed to allow mobility across the network without service disruption (something that is achieved only partially in the Internet). Finally it is important that the network provide quality of service guarantees to applications and isolate them from each other. As such data center networks must deal with a lot of application specific network state. Unlike the Internet, all these properties must hold together, and best-effort network service will not do.

At the same time, data center networks come with certain advantages; For e.g.,

they are largely managed by a single entity and as such do not have stringent inter-operability demands, they usually are highly engineered with many common recurring design elements and have a more controlled stucture than traditional networks. Because the constraints these networks operate in are so different, directly applying different solutions for each of the above requirements from what exists for the general Internet usually results in a very kludgy, error prone management setup. Our interactions with Facebook and Microsoft inform us that traditional solutions have frequently failed or require excessive investment in money or time [fac09, lyn12].

The key question that this dissertation aims to answer is how to build an management system tailored to meet data center network requirements.

1.1 Data Center Network Management Challenges

Many challenges around data center network management can be trivially traced back to scale. How can one reduce manual data center configuration required with scale? Where should the network state that increases with increasing number of data center applications be placed so that it can be effectively be applied on a per-packet basis? How does one design a fault localization algorithm that can explore all possible sets of failures in a data center in a short amount of time? These questions are hard due to the following challenges:

1.1.1 Address assignment and forwarding: Plug-and-play fabrics do not scale

What kind of addresses should the network fabric forward on? Traditionally layer 2 fabrics forward based on flat MAC addresses that come pre-configured with the device. As such layer 2 networks have a plug-and-play property not requiring manual configuration. Unfortunately layer 2 fabrics do not scale very well because every network device is required to remember where each flat address is located with respect to itself. In a data center, this means that network devices have to remember locations of hundreds of thousands of servers. Network devices do not have sufficient memory to

handle all of this state. Layer 2 fabrics have other shortcomings that prevent them from scaling and being used directly in data centers.

On the other hand, layer 3 fabrics forward based on hierarchical IP addresses that encode location. Sufficient manual configuration is required to assign IP addresses to servers based on their location in the network, which increases with increase in network size. However once configured, these fabrics scale very well because the network device can tell where each server is based on its IP prefix.

Thus, scalable network fabrics are those that forward on location based addresses. But assignment of location based addresses mandates manual configuration, which in turn does not scale, and is error prone. In this work we ask the question: Is it possible to build a network fabric that forwards on location based addresses and yet does not mandate manual configuration?

1.1.2 Policy enforcement: Multi-tenant data centers cannot both scale to large number of applications and expect bare-metal network performance

As the number of applications multiplexed on a single data center increase, the network state required to ensure adequate quality of service and isolation for application traffic increases. This problem is increasingly apparent in multi-tenant data centers where many customers with conflicting needs run applications on a shared network. Since network devices have limited memory, they cannot hold all the application specific network state. The common solution employed today is to place this network state in software in the servers. This network state has to be consulted on a per-packet basis and puts stress on server CPU. The resulting software network processing pipeline also adds unpredictable latency and reduces throughput of application traffic.

Thus there is a trade-off between number of applications that a data center can multiplex, and the performance it can provide to the applications. In this work we seek to find techniques to provide adequate network performance to critical applications while multiplexing as many applications as possible within data centers.

1.1.3 Fault localization: With scale, ensuring both accurate and timely diagnosis becomes hard

Data center networks and applications that run on them depend on the health of a very large number of entities and exhibit complex interactions with these entities, which makes diagnosis hard. For e.g., in Lync [lyn12], a messaging system that runs in Microsoft's data centers, there are over 8000 components that interact in complex ways. When diagnosing what state of the system caused a particular set of failures, the localization algorithm might have to theoretically explore 2^{8000} possible component states to localize a fault in Lync, assuming each component has two valid states i.e., it is either healthy or has failed. However if a localization algorithm takes a long time, there might be limited use of its result because it is possible to run manual diagnosis within that amount of time to get to the diagnosis. As such the usefulness of a localization algorithm is dependent on not only its accuracy, but also the time it takes to deliver a diagnosis.

As scale of networks grow, automated fault localization becomes increasingly difficult and needs to compromise on either accuracy or time. Things are further exacerbated by monitoring noise and other factors. In this work we try to derive a fault localization algorithm from first principles that is more accurate and/or less time consuming compared to existing fault localization algorithms built for large networked systems. The main goal is to reduce mean-time-to-recovery and reduce service disruption in large networks.

1.2 Management in Today's Data Centers

Management in most data centers today derives heavily from techniques used in wide area networks. In this section we explore most commonly used techniques for each of the three challenges presented.

1.2.1 Address assignment and Forwarding

There are a number of available data forwarding techniques in data center networks. The high-level dichotomy is between creating a layer 2 network or a layer 3 network, each with associated trade-offs. A layer 3 approach assigns IP addresses to hosts hierarchically based on their directly connected switch.

Standard intra-domain routing protocols such as OSPF [Moy98] may be employed among switches to find shortest paths among hosts. Failures in large-scale network topologies will be commonplace. OSPF can detect such failures and then broadcast the information to all switches to avoid failed links or switches. Transient loops with layer 3 forwarding is less of an issue because the IP-layer TTL limits per-packet resource consumption while forwarding tables are being asynchronously updated. Unfortunately layer 3 forwarding imposes high administrative burden requiring manual assignment of IP addresses to subnetworks, set subnet identifiers on a per switch basis, synchronize DHCP server state with subnet identifiers, etc.

For these reasons, certain data centers deploy a layer 2 network where forwarding is performed based on flat MAC addresses. A layer 2 fabric imposes less administrative overhead. Layer 2 fabrics have their own challenges of course. Standard Ethernet bridging [otICS01] does not scale to networks with tens of thousands of hosts because of the need to support broadcast across the entire fabric. Worse, the presence of a single forwarding spanning tree (even if optimally designed) would severely limit performance in topologies that consist of multiple available equal cost paths.

A middle ground between a layer 2 and layer 3 fabric consists of employing virtual lans, i.e., VLANs to allow a single logical layer 2 fabric to cross multiple switch boundaries. While feasible for smaller-scale topologies, VLANs also suffer from a number of drawbacks. For instance, they require bandwidth resources to be explicitly assigned to each VLAN at each participating switch, limiting flexibility for dynamically changing communication patterns. Next, each switch must maintain state for all hosts in each VLAN that they participate in, limiting scalability. Finally, VLANs also use a single forwarding spanning tree, limiting performance.

1.2.2 Policy enforcement

Virtualization is a commonly used technique in data centers to isolate multiple applications and customers. A virtual switch in the hypervisor, called the vswitch, manages traffic transiting between virtual machines (VMs) on a single host, and between those VMs and the network at large. In a multi-tenant setting the vswitch is typically configured to isolate traffic of VMs belonging to different tenants. Such a vswitch is also used to enforce security, QoS rules and interface rate limits as per application requirements. The network fabric on the other hand is only responsible for routing packets to the appropriate servers.

Since these rules are being applied on a per-packet basis, network processing in the virtual switch results in software queuing. Unlike hardware queues that have deterministic processing time, the vswitch queue is processed opportunistically between other tasks the hypervisor has to perform. Therefore packets in the queue are susceptible to unpredictable delays. These delays can cause significant drop in service throughput. As more and more VMs are multiplexed on the same server and network traffic increases, the vswitch queue can become a bottleneck.

1.2.3 Fault localization

Practical fault localization in large scale systems is a black art. Most localization algorithms are used in conjunction with manual diagnosis, because manual diagnosis is usually considered to yield the best results in terms of accuracy. But this comes at the expense of large time-to-diagnosis and operator effort. We have consistently heard from operators (e.g., at both Google and Microsoft) that the effectiveness of existing fault localization algorithms depends on the network, and this dependence is mysterious. As networks grow it is hard to keep the localization algorithm up to date. There are no studies that connect network characteristics to the choice of algorithm; thus, determining an appropriate fault localization approach for a given network is difficult.

With Lync we find that manual localization is employed for most trouble-tickets and diagnosis takes hours to multiple days.

1.3 Emerging trends for redesigning for manageability

Recently there have been a number of proposals to overcome the shortcomings of existing protocols applied to data centers. In this section we explore alternative network fabrics, policy enforcement techniques, and proposed fault localization algorithms.

1.3.1 Address assignment and Forwarding

RBridges and TRILL [PED+09] and their IETF standardization effort, address some of the routing challenges in layer 2 Ethernet. RBridges run a layer 2 routing protocol among switches. Essentially switches broadcast information about their local connectivity along with the identity of all directly connected end hosts. Thus, all switches learn the switch topology and the location of all hosts. To limit forwarding table size, ingress switches map destination MAC addresses to the appropriate egress switch (based on global knowledge) and encapsulate the packet in an outer MAC header with the egress switch identifier. In addition, RBridges add a secondary header with a TTL field to protect against loops. To reduce the state and communication overhead associated with routing in large-scale networks, recent work [CCN+06, CCK+06, CKR08] explores using DHTs to perform forwarding on flat labels. These proposals scale better than layer 2 and also support mobility. However each of these has some shortcomings; for e.g., RBridges still requires manual configuration, and SEATTLE [CKR08] is prone to transient loops and service disruption for long time during switch failures.

1.3.2 Policy enforcement

[MYM+11, PYK+10, SKG+11, CKRS10, ovs, hyp, mid] suggest that the vswitch is the best place to achieve scalable tenant network communication, security, and resource isolation. Most of these proposals do not consider the resulting performance impact at scale. vCRIB [MYSG13], proposes a unified rule management system that splits network rules between hypervisors and network hardware. While vCRIB recognizes that the vswitch can become a bottleneck if used excessively for network processing, vCRIB only aims to reduce the number of rules placed in vswitch, without reducing the number of packets that have to transit the vswitch. It is well known that effect of processor con-

text switches, copies, and interrupt overhead in I/O-intensive virtualized environments significantly reduces network performance [AA06, RS07, Liu10]. As such vCRIB does not achieve significant performance gains. In the same vein, NIC tunnel offloads are being proposed and standardized [emu, stt] to reduce some of the network processing burden from software. Tunnel termination network switches [ari] are being introduced at the boundary of virtualized environments and non-virtualized environments, typically configured with static tunnels. However, security rule checking and rate limiting are still largely retained in the vswitch and L4 software or hardware middleboxes.

1.3.3 Fault localization

Fault localization has also been studied widely [KYGCS05, CKFF02, BCG+07, KMV+09, DTDD07, ALMP10, KKV05, KYGCS07, KYY+95, Ris05, SS04a]. However Codebook is the only technique known to be applied in real systems in conjunction with manual diagnosis. Some diagnostic tools like [MGS+09, NKN12, OA11] leave fault localization to a knowledgeable network operator and aim to provide the operator with a reduced dependency graph for a particular failure. In this work we explore inference techniques for large networks that narrow the space of possible culprits as much as possible for the network operator with minimal manual intervention.

1.4 Automated scalable management system for data center networks

In this work we propose a automated scalable management system which consists of three parts: A scalable layer 2 fabric for data centers called PortLand, a scalable policy enforcement system that optimizes rule placement to maximize application network performance called FasTrak, and a scalable fault localization algorithm for large networked systems called Gestalt. Both PortLand and FasTrak use logically centralized controllers to control network state while Gestalt can run within such a controller to localize and pinpoint faults. The PortLand controller controls naming and forwarding paths set up in the network and the FasTrak controller determines where policy rules are

placed. We have built prototypes for all three parts and evaluated them to get a sense of their benefits and costs.

1.4.1 PortLand [NMPF+09]: A Scalable Network Fabric for Data Centers

PortLand is built with the goal of achieving the following five goals together: 1) Data center fabrics must be plug-and-play 2) Workload mobility across the entire data must be supported 3) The fabric must completely utilize path diversity in physical infrastructure 4) The fabric must ensure there are no loops and 5) Failure convergence must be rapid. These five goals were those that were highlighted from our conversations with network operators. Our key insight in building PortLand was that data centers are usually built as multi-rooted trees. This insight helps us build a distributed location discovery protocol that enables the fabric to be plug-and-play, and yet similar to layer 3 fabrics.

1.4.2 FasTrak [NMPV]: A Scalable Policy Enforcement Controller for Multi-Tenant Data Centers

FasTrak is built with a goal to enable performance sensitive network bound applications to be migrated to the shared environment of multi-tenant clouds. Multi-tenant data centers help share the same physical infrastructure among tens of thousands of tenants, eliminating infrastructure costs for them. But because of the enormous amount of network state that needs to be maintained on behalf of the tenants, these data centers place all policy rules in hypervisors on servers. Enforcing these rules in software imposes significant computational overhead, as well as increased latency. FasTrak uses network hardware as a cache and moves rules back and forth between hypervisors and switch hardware, focusing on minimizing the overall average latency of the network.

1.4.3 Gestalt [NMMVV]: A Scalable Fault Localization Algorithm for Large Networked Systems

To develop a localization algorithm that can work for a variety of networks from first principles, we develop a framework that captures the design space of existing practical fault localization algorithms. It is based on the observation that the essence of these algorithms can be anatomized into a common three-part pattern, and different algorithms can be seen as making different choices for each part. Using this framework, we analyze the effectiveness of each algorithm and its choices at addressing challenges that large, complex networks present. Our analysis helps develop a fault localization tool Gestalt that can localize faults in any large network. Experiments with simulated and real failures over three diverse networks show that Gestalt has higher accuracy or lower overhead than existing algorithms.

1.5 Acknowledgment

Chapter 1, in part, contains material as it appears in the Proceedings of the ACM SIGCOMM 2009 conference on Data communication. Niranjan Mysore, Radhika; Pamboris, Andreas; Farrington, Nathan; Huang, Nelson; Miri, Pardis; Radhakrishnan, Sivasankar; Subramanya, Vikream; Vahdat, Amin. The dissertation author was the primary investigator and author of this paper.

Chapter 1, in part, contains material submitted for publication as ”Fastrak: Enabling express express lanes in multi-tenant data centers” Niranjan Mysore, Radhika; Porter, George; Vahdat, Amin. The dissertation author was the primary investigator and author of this paper.

Chapter 1, in part, contains material submitted for publication as ”Gestalt: Unifying fault localization for networked systems” Niranjan Mysore, Radhika; Mahajan, Ratul; Vahdat, Amin; Varghese, George. The dissertation author was the primary investigator and author of this paper.

Chapter 2

PortLand: A Scalable Fault Tolerant Layer 2 Data Center Network Fabric

This chapter considers the requirements for a scalable, easily manageable, fault-tolerant, and efficient data center network fabric. Trends in multi-core processors, end-host virtualization, and commodities of scale are pointing to future single-site data centers with millions of virtual end points. Existing layer 2 and layer 3 network protocols face some combination of limitations in such a setting: lack of scalability, difficult management, inflexible communication, or limited support for virtual machine migration. To some extent, these limitations may be inherent for Ethernet/IP style protocols when trying to support arbitrary topologies. We observe that data center networks are often managed as a single logical network fabric with a known baseline topology and growth model. We leverage this observation in the design and implementation of PortLand, a scalable, fault tolerant layer 2 routing and forwarding protocol for data center environments. Through our implementation and evaluation, we show that PortLand holds promise for supporting a "plug-and-play" large-scale, data center network.

2.1 Introduction

There is an increasing trend toward migrating applications, computation and storage into data centers spread across the Internet. Benefits from commodities of scale are leading to the emergence of "mega data centers" hosting applications running on

tens of thousands of servers [meg]. For instance, a web search request may access an inverted index spread across 1,000+ servers, and data storage and analysis applications may interactively process petabytes of information stored on thousands of machines. There are significant application networking requirements across all these cases.

In the future, a substantial portion of Internet communication will take place within data center networks. These networks tend to be highly engineered, with a number of common design elements. And yet, the routing, forwarding, and management protocols that we run in data centers were designed for the general LAN setting and are proving inadequate along a number of dimensions. Consider a data center with 100,000 servers, each hosting 32 virtual machines (VMs). This translates to a total of three million IP and MAC addresses in the data center. Assuming one switch is required for every 25 physical hosts and accounting for interior nodes, the topology would consist of 8,000 switches.

Current network protocols impose significant management overhead at this scale. For example, an end host's IP address may be determined by its directly-connected physical switch and appropriately synchronized with replicated DHCP servers. VLANs may provide some naming flexibility across switch boundaries but introduce their own configuration and resource allocation overheads. Ideally, data center network architects and administrators would have "plug-and-play" deployment for switches. Consider some of the requirements for such a future scenario:

- **R1.** Any VM may migrate to any physical machine. Migrating VMs should not have to change their IP addresses as doing so will break pre-existing TCP connections and application-level state.
- **R2.** An administrator should not need to configure any switch before deployment.
- **R3.** Any end host should be able to efficiently communicate with any other end host in the data center along any of the available physical communication paths.
- **R4.** There should be no forwarding loops.
- **R5.** Failures will be common at scale, so failure detection should be rapid and efficient. Existing unicast and multicast sessions should proceed unaffected to the extent allowed by underlying physical connectivity.

Let us now map these requirements to implications for the underlying network protocols. R1 and R2 essentially require supporting a single layer 2 fabric for the entire data center. A layer 3 fabric would require configuring each switch with its subnet information and synchronizing DHCP servers to distribute IP addresses based on the host's subnet. Worse, transparent VM migration is not possible at layer 3 (save through techniques designed for IP mobility) because VMs must switch their IP addresses if they migrate to a host on a different subnet. Unfortunately, layer 2 fabrics face scalability and efficiency challenges because of the need to support broadcast. Further, R3 at layer 2 requires MAC forwarding tables with potentially hundreds of thousands or even millions of entries, impractical with today's switch hardware. R4 is difficult for either layer 2 or layer 3 because forwarding loops are possible during routing convergence. A layer 2 protocol may avoid such loops by employing a single spanning tree (inefficient) or tolerate them by introducing an additional header with a TTL (incompatible). R5 requires efficient routing protocols that can disseminate topology changes quickly to all points of interest. Unfortunately, existing layer 2 and layer 3 routing protocols, e.g., ISIS and OSPF, are broadcast based, with every switch update sent to all switches. On the efficiency side, the broadcast overhead of such protocols would likely require configuring the equivalent of routing areas [cisd], contrary to R2.

Hence, the current assumption is that the vision of a unified plug-and-play large-scale network fabric is unachievable, leaving data center network architects to adopt ad hoc partitioning and configuration to support large-scale deployments. Recent work in SEATTLE [CKR08] makes dramatic advances toward a plug-and-play Ethernet- compatible protocol. However, in SEATTLE, switch state grows with the number of hosts in the data center, forwarding loops remain possible, and routing requires all-to-all broadcast, violating R3, R4, and R5. Section 2.3.7 presents a detailed discussion of both SEATTLE and TRILL [TP09].

In this paper, we present PortLand, a set of Ethernet- compatible routing, forwarding, and address resolution protocols with the goal of meeting R1-R5 above. The principal observation behind our work is that data center networks are often physically inter-connected as a multi-rooted tree [cisa]. Using this observation, PortLand employs a lightweight protocol to enable switches to discover their position in the topology. Port-

Land further assigns internal Pseudo MAC (PMAC) addresses to all end hosts to encode their position in the topology. PMAC addresses enable efficient, provably loop-free forwarding with small switch state.

We have a complete implementation of PortLand. We provide native fault-tolerant support for ARP, network-layer multicast, and broadcast. PortLand imposes little requirements on the underlying switch software and hardware. We hope that PortLand enables a move towards more flexible, efficient and fault-tolerant data centers where applications may flexibly be mapped to different hosts, i.e. where the data center network may be treated as one unified fabric.

2.2 Background

2.2.1 Data Center Networks

Topology Current data centers consist of thousands to tens of thousands of computers with emerging mega data centers hosting 100,000+ compute nodes. As one example, consider our interpretation of current best practices [cisa] for the layout of a 11,520-port data center network. Machines are organized into racks and rows, with a logical hierarchical network tree overlaid on top of the machines. In this example, the data center consists of 24 rows, each with 12 racks. Each rack contains 40 machines interconnected by a top of rack (ToR) switch that delivers non-blocking bandwidth among directly connected hosts. Today, a standard ToR switch contains 48 GigE ports and up to 4 available 10 GigE uplinks.

ToR switches connect to end of row (EoR) switches via 1-4 of the available 10 GigE uplinks. To tolerate individual switch failures, ToR switches may be connected to EoR switches in different rows. An EoR switch is typically a modular 10 GigE switch with a number of ports corresponding to the desired aggregate bandwidth. For maximum bandwidth, each of the 12 ToR switches would connect all 4 available 10 GigE uplinks to a modular 10 GigE switch with up to 96 ports. 48 of these ports would face downward towards the ToR switches and the remainder of the ports would face upward to a *core* switch layer. Achieving maximum bandwidth for inter-row communication in this example requires connecting 48 upward facing ports from each of 24 EoR switches

to a core switching layer consisting of 12 96-port 10 GigE switches.

Forwarding There are a number of available data forwarding techniques in data center networks. The high-level dichotomy is between creating a Layer 2 network or a Layer 3 network, each with associated tradeoffs. A Layer 3 approach assigns IP addresses to hosts hierarchically based on their directly connected switch. In the example topology above, hosts connected to the same ToR could be assigned the same /26 prefix and hosts in the same row may have a /22 prefix. Such careful assignment will enable relatively small forwarding tables across all data center switches.

Standard intra-domain routing protocols such as OSPF [Moy98] may be employed among switches to find shortest paths among hosts. Failures in large-scale network topologies will be commonplace. OSPF can detect such failures and then broadcast the information to all switches to avoid failed links or switches. Transient loops with layer 3 forwarding is less of an issue because the IP-layer TTL limits per-packet resource consumption while forwarding tables are being asynchronously updated.

Unfortunately, Layer 3 forwarding does impose administrative burden as discussed above. In general, the process of adding a new switch requires manual administrator configuration and oversight, an error prone process. Worse, improperly synchronized state between system components, such as a DHCP server and a configured switch subnet identifier can lead to unreachable hosts and difficult to diagnose errors. Finally, the growing importance of end host virtualization makes Layer 3 solutions less desirable as described below.

For these reasons, certain data centers deploy a layer 2 network where forwarding is performed based on flat MAC addresses. A layer 2 fabric imposes less administrative overhead. Layer 2 fabrics have their own challenges of course. Standard Ethernet bridging [otICS01] does not scale to networks with tens of thousands of hosts because of the need to support broadcast across the entire fabric. Worse, the presence of a single forwarding spanning tree (even if optimally designed) would severely limit performance in topologies that consist of multiple available equal cost paths.

A middle ground between a Layer 2 and Layer 3 fabric consists of employing VLANs to allow a single logical Layer 2 fabric to cross multiple switch boundaries.

While feasible for smaller-scale topologies, VLANs also suffer from a number of drawbacks. For instance, they require bandwidth resources to be explicitly assigned to each VLAN at each participating switch, limiting flexibility for dynamically changing communication patterns. Next, each switch must maintain state for all hosts in each VLAN that they participate in, limiting scalability. Finally, VLANs also use a single forwarding spanning tree, limiting performance.

End Host Virtualization The increasing popularity of end host virtualization in the data center imposes a number of requirements on the underlying network. Commercially available virtual machine monitors allow tens of VMs to run on each physical machine in the data center[1], each with their own fixed IP and MAC addresses. In data centers with hundreds of thousands of hosts, this translates to the need for scalable addressing and forwarding for millions of unique end points. While individual applications may not (yet) run at this scale, application designers and data center administrators alike would still benefit from the ability to arbitrarily map individual applications to an arbitrary subset of available physical resources.

Virtualization also allows the entire VM state to be transmitted across the network to migrate a VM from one physical machine to another [CFH+05]. Such migration might take place for a variety of reasons. A cloud computing hosting service may migrate VMs for statistical multiplexing, packing VMs on the smallest physical footprint possible while still maintaining performance guarantees. Further, variable bandwidth to remote nodes in the data center could warrant migration based on dynamically changing communication patterns to achieve high bandwidth for tightly-coupled hosts. Finally, variable heat distribution and power availability in the data center (in steady state or as a result of component cooling or power failure) may necessitate VM migration to avoid hardware failures.

Such an environment currently presents challenges both for Layer 2 and Layer 3 data center networks. In a Layer 3 setting, the IP address of a virtual machine is set by its directly-connected switch subnet number. Migrating the VM to a different

[1] One rule of thumb for the degree of VM-multiplexing allocates one VM per thread in the underlying processor hardware. x86 machines today have 2 sockets, 4 cores/processor, and 2 threads/core. Quad socket, eight core machines will be available shortly.

switch would require assigning a new IP address based on the subnet number of the new first-hop switch, an operation that would break all open TCP connections to the host and invalidate any session state maintained across the data center, etc. A Layer 2 fabric is agnostic to the IP address of a VM. However, scaling ARP and performing routing/forwarding on millions of flat MAC addresses introduces a separate set of challenges.

2.2.2 Fat Tree Networks

Recently proposed work [AFLV08,GLM+08,GWT+08] suggest alternate topologies for scalable data center networks. In this paper, we consider designing a scalable fault tolerant layer 2 domain over one such topology, a fat tree. As will become evident, the fat tree is simply an instance of the traditional data center multi-rooted tree topology (Section 2.2.1). Hence, the techniques described in this paper generalize to existing data center topologies. We present the fat tree because our available hardware/software evaluation platform (Section 2.4) is built as a fat tree.

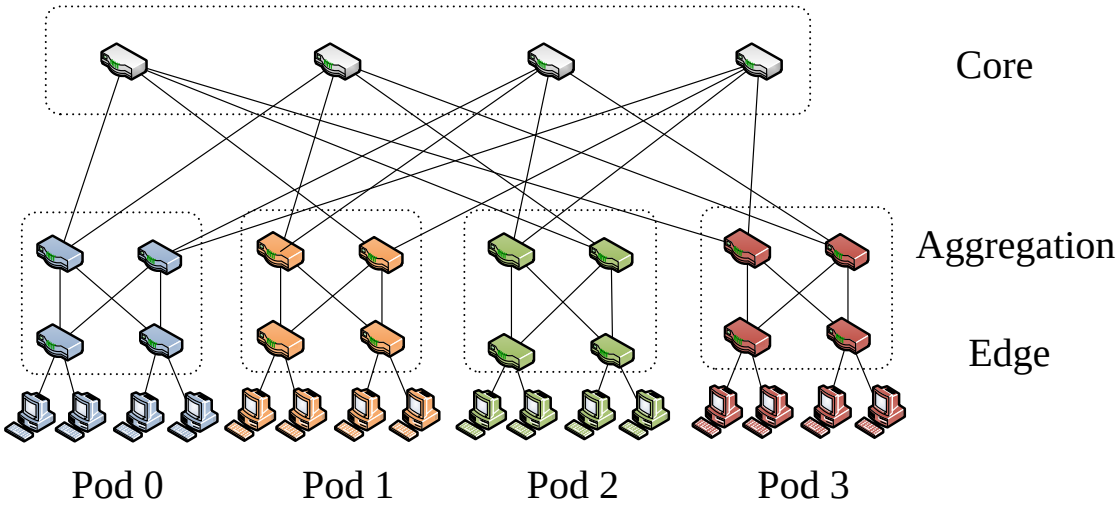


Figure 2.1: Sample fat tree topology.

Figure 2.1 depicts a 16-port switch built as a multi-stage topology from constituent 4-port switches. In general, a three-stage fat tree built from k-port switches can

support non-blocking communication among $k^3/4$ end hosts using $5k^2/4$ individual k-port switches. We split the fat tree into three layers, labeled edge, aggregation and core as in Figure 2.1. The fat tree as a whole is split into k individual *pods*, with each pod supporting non-blocking operation among $k^2/4$ hosts. Non-blocking operation requires careful scheduling of packets among all available paths, a challenging problem. While a number of heuristics are possible, for the purposes of this work we assume ECMP-style hashing of flows [Hop00] among the $k^2/4$ available paths between a given source and destination. While current techniques are less than ideal, we consider the flow scheduling problem to be beyond the scope of this paper.

2.2.3 Related Work

Recently, there have been a number of proposals for network architectures specifically targeting the data center. Two recent proposals [GLM+08, AFLV08] suggest topologies based on fat trees [Lei85]. As discussed earlier, fat trees are a form of multi-rooted trees that already form the basis for many existing data center topologies. As such, they are fully compatible with our work and in fact our implementation runs on top of a small-scale fat tree. DCell [GWT+08] also recently proposed a specialized topology for the data center environment. While not strictly a multi-rooted tree, there is implicit hierarchy in the DCell topology, which should make it compatible with our techniques.

Others have also recently recognized the need for more scalable layer 2 networks. SmartBridge [RTA01] extended the original pioneering work on learning bridges [otICS01] to move beyond single spanning tree networks while maintaining the loop free property of extended LANs. However, SmartBridge still suffers from the scalability challenges characteristic of Ethernet networks. Contemporaneous to our work, MOOSE [SC08] also suggests the use of hierarchical Ethernet addresses and header rewriting to address some of Ethernet's scalability limitations.

RBridges and TRILL [PED+09], its IETF standardization effort, address some of the routing challenges in Ethernet. RBridges run a layer 2 routing protocol among switches. Essentially switches broadcast information about their local connectivity along with the identity of all directly connected end hosts. Thus, all switches learn

the switch topology and the location of all hosts. To limit forwarding table size, ingress switches map destination MAC addresses to the appropriate egress switch (based on global knowledge) and encapsulate the packet in an outer MAC header with the egress switch identifier. In addition, RBridges add a secondary header with a TTL field to protect against loops. We also take inspiration from CMU Ethernet [MNZ04], which also proposed maintaining a distributed directory of all host information. Relative to both approaches, PortLand is able to achieve improved fault tolerance and efficiency by leveraging knowledge about the baseline topology and avoiding broadcast-based routing protocols altogether.

Failure Carrying Packets (FCP) [LCR+07] shows the benefits of assuming some knowledge of baseline topology in routing protocols. Packets are marked with the identity of all failed links encountered between source and destination, enabling routers to calculate new forwarding paths based on the failures encountered thus far. Similar to PortLand, FCP shows the benefits of assuming knowledge of baseline topology to improve scalability and fault tolerance. For example, FCP demonstrates improved routing convergence with fewer network messages and lesser state.

To reduce the state and communication overhead associated with routing in large-scale networks, recent work [CCN+06, CCK+06, CKR08] explores using DHTs to perform forwarding on flat labels. We achieve similar benefits in per-switch state overhead with lower network overhead and the potential for improved fault tolerance and efficiency, both in forwarding and routing, by once again leveraging knowledge of the baseline topology.

2.3 Design

The goal of PortLand is to deliver scalable layer 2 routing, forwarding, and addressing for data center network environments. We leverage the observation that in data center environments, the baseline multi-rooted network topology is known and relatively fixed. Building and maintaining data centers with tens of thousands of compute elements requires modularity, advance planning, and minimal human interaction. Thus, the baseline data center topology is unlikely to evolve quickly. When expansion does

occur to the network, it typically involves adding more "leaves" (e.g., rows of servers) to the multi-rooted tree topology described in Section 2.2.1.

2.3.1 Fabric Manager

PortLand employs a logically centralized *fabric manager* that maintains soft state about network configuration information such as topology. The fabric manager is a user process running on a dedicated machine responsible for assisting with ARP resolution, fault tolerance, and multicast as further described below. The fabric manager may simply be a redundantly-connected host in the larger topology or it may run on a separate control network.

There is an inherent trade off between protocol simplicity and system robustness when considering a distributed versus centralized realization for particular functionality. In PortLand, we restrict the amount of centralized knowledge and limit it to soft state. In this manner, we eliminate the need for any administrator configuration of the fabric manager (e.g., number of switches, their location, their identifier). In deployment, we expect the fabric manager to be replicated with a primary asynchronously updating state on one or more backups. Strict consistency among replicas is not necessary as the fabric manager maintains no hard state.

Our approach takes inspiration from other recent large-scale infrastructure deployments. For example, modern storage [GGL03] and data processing systems [DG04] employ a centralized controller at the scale of tens of thousands of machines. In another setting, the Route Control Platform [CCF+05] considers centralized routing in ISP deployments. All the same, the protocols described in this paper are amenable to distributed realizations if the tradeoffs in a particular deployment environment tip against a central fabric manager.

2.3.2 Positional Pseudo MAC Addresses

The basis for efficient forwarding and routing as well as VM migration in our design is hierarchical Pseudo MAC (PMAC) addresses. PortLand assigns a unique PMAC address to each end host. The PMAC encodes the location of an end host in the topol-

ogy. For example, all end points in the same pod will have the same prefix in their assigned PMAC. The end hosts remain unmodified, believing that they maintain their actual MAC (AMAC) addresses. Hosts performing ARP requests receive the PMAC of the destination host. All packet forwarding proceeds based on PMAC addresses, enabling very small forwarding tables. Egress switches perform PMAC to AMAC header rewriting to maintain the illusion of unmodified MAC addresses at the destination host.

PortLand edge switches learn a unique pod number and a unique position number within each pod. We employ the Location Discovery Protocol (Section 2.3.4) to assign these values. For all directly connected hosts, edge switches assign a 48-bit PMAC of the form $pod.position.port.vmid$ to all directly connected hosts, where pod (16 bits) reflects the pod number of the edge switch, $position$ (8 bits) is its position in the pod, and $port$ (8 bits) is the switch-local view of the port number the host is connected to. We use $vmid$ (16 bits) to multiplex multiple virtual machines on the same physical machine (or physical hosts on the other side of a bridge). Edge switches assign monotonically increasing $vmid$'s to each subsequent new MAC address observed on a given port. PortLand times out $vmid$'s without any traffic and reuses them.

When an ingress switch sees a source MAC address never observed before, the packet is vectored to the switch software. The software creates an entry in a local PMAC table mapping the host's AMAC and IP address to its PMAC. The switch constructs the PMAC as described above and communicates this mapping to the fabric manager as depicted in Figure 2.2. The fabric manager uses this state to respond to ARP requests (Section 2.3.3). The switch also creates the appropriate flow table entry to rewrite the PMAC destination address to the AMAC for any traffic destined to the host.

In essence, we separate host location from host identifier [MN06] in a manner that is transparent to end hosts and compatible with existing commodity switch hardware. Importantly, we do not introduce additional protocol headers. From the underlying hardware, we require flow table entries to perform deterministic PMAC $\leftrightarrow$ AMAC rewriting as directed by the switch software. We also populate switch forwarding entries based on longest prefix match against a destination PMAC address. OpenFlow [ope] supports both operations and native hardware support is also available in commodity switches [cisb].

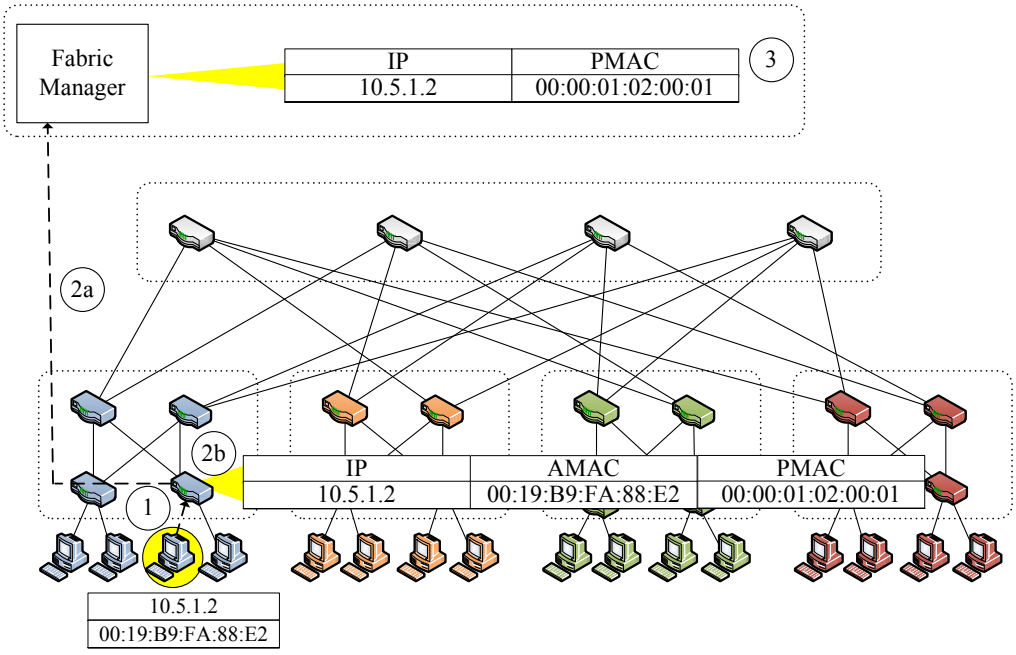


Figure 2.2: Actual MAC to Pseudo MAC mapping.

2.3.3 Proxy-based ARP

Ethernet by default broadcasts ARPs to all hosts in the same layer 2 domain. We leverage the fabric manager to reduce broadcast overhead in the common case, as depicted in Figure 2.3. In step 1, an edge switch intercepts an ARP request for an IP to MAC address mapping and forwards the request to the fabric manager in step 2. The fabric manager consults its PMAC table to see if an entry is available for the target IP address. If so, it returns the PMAC in step 3 to the edge switch. The edge switch creates an ARP reply in step 4 and returns it to the original host.

It is possible that the fabric manager does not have the IP to PMAC mapping available, for example after failure. In this case, the fabric manager will fall back to broadcast to all end hosts to retrieve the mapping. Efficient broadcast is straightforward in the failure-free case (fault-tolerance extensions are described below): the ARP is transmitted to any core switch, which in turn distributes it to all pods and finally all edge switches. The target host will reply with its AMAC, which will be rewritten by the ingress switch to the appropriate PMAC before forwarding to both the querying host

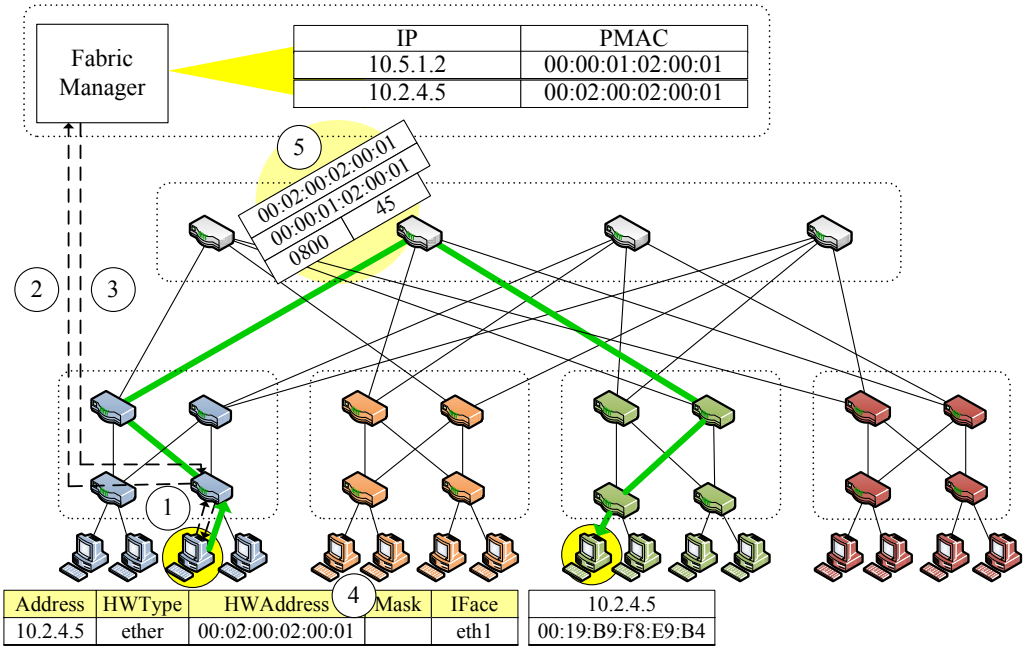


Figure 2.3: Proxy ARP.

and the fabric manager.

Note that end hosts receive PMACs in response to an ARP request and that all packet forwarding proceeds based on the hierarchical PMAC. The egress switch performs PMAC to AMAC rewriting only on the last hop to the destination host. In the baseline, forwarding in each switch requires just $O(k)$ state using hierarchical PMAC addresses. This required state compares favorably to standard layer 2 switches that require an entry for every flat MAC address in the network, i.e., tens or even hundreds of thousands in large deployments. Additional forwarding state may be required to perform per-flow load balancing across multiple paths [AFLV08].

There is one additional detail for supporting VM migration. Upon completing migration from one physical machine to another, the VM sends a gratuitous ARP with its new IP to MAC address mapping. This ARP is forwarded to the fabric manager in the normal manner. Unfortunately, any hosts communicating with the migrated VM will maintain that host's previous PMAC in their ARP cache and will be unable to continue communication until their ARP cache entry times out. To address this limitation, the

fabric manager forwards an invalidation message to the migrated VM's previous switch. This message sets up a flow table entry to trap handling of subsequent packets destined to the invalidated PMAC to the switch software. The switch software transmits a unicast gratuitous ARP back to any transmitting host to set the new PMAC address in that host's ARP cache. The invalidating switch may optionally transmit the packet to the actual destination to prevent packet loss.

2.3.4 Distributed Location Discovery

PortLand switches use their position in the global topology to perform more efficient forwarding and routing using only pairwise communication. Switch position may be set manually with administrator intervention, violating some of our original goals. Since position values should be slow to change, this may still be a viable option. However, to explore the limits to which PortLand switches may be entirely plug-and-play, we also present a location discovery protocol (LDP) that requires no administrator configuration. PortLand switches do not begin packet forwarding until their location is established.

PortLand switches periodically send a Location Discovery Message (LDM) out all of their ports both, to set their positions and to monitor liveness in steady state. LDMs contain the following information:

- *Switch identifier (switch_id)*: a globally unique identifier for each switch, e.g., the lowest MAC address of all local ports.
- *Pod number (pod)*: a number shared by all switches in the same pod (see Figure 2.1). Switches in different pods will have different pod numbers. This value is never set for core switches.
- *Position (pos)*: a number assigned to each edge switch, unique within each pod.
- *Tree level (level)*: 0, 1, or 2 depending on whether the switch is an edge, aggregation, or core switch. Our approach generalizes to deeper hierarchies.
- *Up/down (dir)*: Up/down is a bit which indicates whether a switch port is facing downward or upward in the multi-rooted tree.

Initially, all values other than the switch identifier and port number are unknown and we assume the fat tree topology depicted in Figure 2.1. However, LDP also generalizes to multi-rooted trees as well as partially connected fat trees. We assume all switch ports are in one of three states: disconnected, connected to an end host, or connected to another switch.

The key insight behind LDP is that edge switches receive LDMs only on the ports connected to aggregation switches (end hosts do not generate LDMs). We use this observation to bootstrap level assignment in LDP. Edge switches learn their level by determining that some fraction of their ports are host connected. Level assignment then flows up the tree. Aggregations switches set their level once they learn that some of their ports are connected to edge switches. Finally, core switches learn their levels once they confirm that all ports are connected to aggregation switches.

Algorithm 1 presents the processing performed by each switch in response to LDMs. Lines 2-4 are concerned with position assignment and will be described below. In line 6, the switch updates the set of switch neighbors that it has heard from. In lines 7-8, if a switch is not connected to more than $k/2$ neighbor switches for sufficiently long, it concludes that it is an edge switch. The premise for this conclusion is that edge switches have at least half of their ports connected to end hosts. Once a switch comes to this conclusion, on any subsequent LDM it receives, it infers that the corresponding incoming port is an upward facing one. While not shown for simplicity, a switch can further confirm its notion of position by sending pings on all ports. Hosts will reply to such pings but will not transmit LDMs. Other PortLand switches will both reply to the pings and transmit LDMs.

In lines 10-11, a switch receiving an LDM from an edge switch on an upward facing port concludes that it must be an aggregation switch and that the corresponding incoming port is a downward facing port. Lines 12-13 handle the case where core/aggregation switches transmit LDMs on downward facing ports to aggregation/edge switches that have not yet set the direction of some of their ports.

Determining the level for core switches is somewhat more complex, as addressed by lines 14-20. A switch that has not yet established its level first verifies that all of its active ports are connected to other PortLand switches (line 14). It then verifies in lines

15-18 that all neighbors are aggregation switches that have not yet set the direction of their links (aggregation switch ports connected to edge switches would have already been determined to be downward facing). If these conditions hold, the switch can conclude that it is a core switch and set all its ports to be downward facing (line 20).

Edge switches must acquire a unique position number in each pod in the range of $0..\frac{k}{2}-1$. This process is depicted in Algorithm 2. Intuitively, each edge switch proposes a randomly chosen number in the appropriate range to all aggregation switches in the same pod. If the proposal is verified by a majority of these switches as unused and not tentatively reserved, the proposal is finalized and this value will be included in future LDMs from the edge switch. As shown in lines 2-4 and 29 of Algorithm 1, aggregation switches will hold a proposed position number for some period of time before timing it out in the case of multiple simultaneous proposals for the same position number.

LDP leverages the fabric manager to assign unique pod numbers to all switches in the same pod. In lines 8-9 of Algorithm 2, the edge switch that adopts position 0 requests a pod number from the fabric manager. This pod number spreads to the rest of the pod in lines 21-22 of Algorithm 1.

For space constraints, we leave a description of the entire algorithm accounting for a variety of failure and partial connectivity conditions to separate work. We do note one of the interesting failure conditions, miswiring. Even in a data center environment, it may still be possible that two host facing ports inadvertently become bridged. For example, someone may inadvertently plug an Ethernet cable between two outward facing ports, introducing a loop and breaking some of the important PortLand forwarding properties. LDP protects against this case as follows. If an uninitialized switch begins receiving LDMs from an edge switch on one of its ports, it must be an aggregation switch or there is an error condition. It can conclude there is an error condition if it receives LDMs from aggregation switches on other ports or if most of its active ports are host-connected (and hence receive no LDMs). In an error condition, the switch disables the suspicious port and signals an administrator exception.

2.3.5 Provably Loop Free Forwarding

Once switches establish their local positions using LDP, they employ updates from their neighbors to populate their forwarding tables. For instance, core switches learn the pod number of directly-connected aggregation switches. When forwarding a packet, the core switch simply inspects the bits corresponding to the pod number in the PMAC destination address to determine the appropriate output port.

Similarly, aggregation switches learn the position number of all directly connected edge switches. Aggregation switches must determine whether a packet is destined for a host in the same or different pod by inspecting the PMAC. If in the same pod, the packet must be forwarded to an output port corresponding to the $position$ entry in the PMAC.

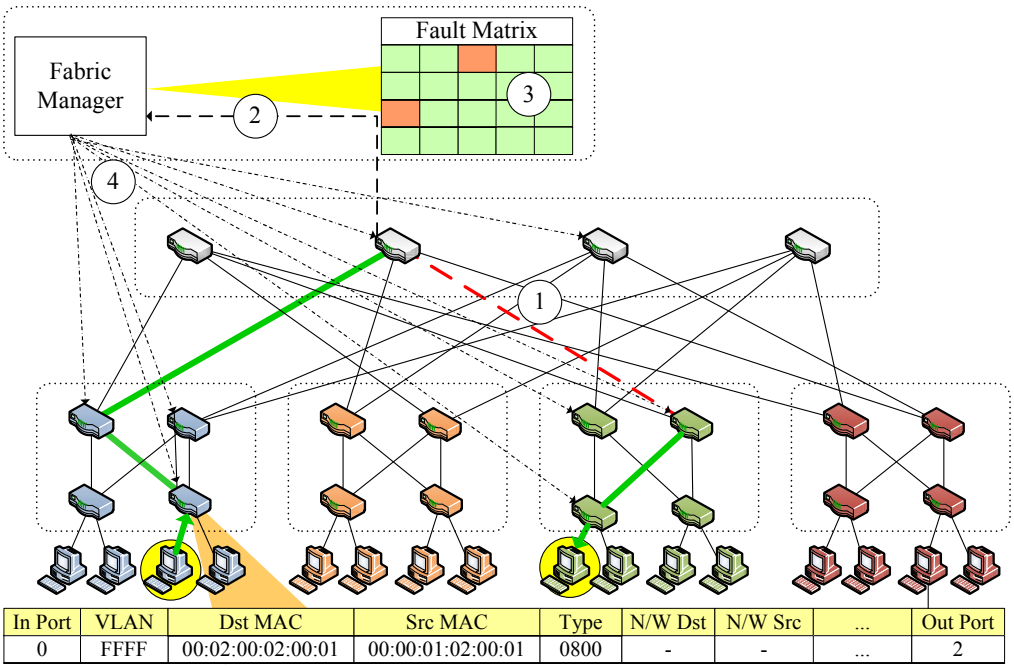
If in a different pod, the packet may be forwarded along any of the aggregation switch's links to the core layer in the fault-free case. For load balancing, switches may employ any number of techniques to choose an appropriate output port. The fabric manager would employ additional flow table entries to override the default forwarding behavior for individual flows. However, this decision is orthogonal to this work, and so we assume a standard technique such as flow hashing in ECMP [Hop00].

PortLand maps multicast groups to a core switch using a deterministic hash function. PortLand switches forward all multicast packets towards this core, e.g., using flow hashing to pick among available paths. With simple hardware support, the hash function may be performed in hardware with no additional state in the fault-free case (exceptions for failures could be encoded in switch SRAM). Without hardware support, there would be one entry per multicast group. Edge switches forward IGMP join requests to the fabric manager using the PMAC address of the joining host. The fabric manager then installs forwarding state in all core and aggregation switches necessary to ensure multicast packet delivery to edge switches with at least one interested host.

Our forwarding protocol is provably loop free by observing up-down semantics [SBB+91] in the forwarding process as explained in Appendix A. Packets will always be forwarded up to either an aggregation or core switch and then down toward their ultimate destination. We protect against transient loops and broadcast storms by ensuring that once a packet begins to travel down, it is not possible for it to travel back

up the topology. There are certain rare simultaneous failure conditions where packets may only be delivered by, essentially, detouring back down to an aggregation switch to get to a core switch capable of reaching a given destination. We err on the side of safety and prefer to lose connectivity in these failure conditions rather than admit the possibility of loops.

2.3.6 Fault Tolerant Routing



In Port	VLAN	Dst MAC	Src MAC	Type	N/W Dst	N/W Src	...	Out Port
0	FFFF	00:02:00:02:00:01	00:00:01:02:00:01	0800	-	-	...	2

Figure 2.4: Unicast: Fault detection and action.

Given a largely fixed baseline topology and the ability to forward based on PMACs, PortLand's routing protocol is largely concerned with detecting switch and link failure/recovery. LDP exchanges (Section 2.3.4) also serve the dual purpose of acting as liveness monitoring sessions. We describe our failure recovery process using an example, as depicted in Figure 2.4. Upon not receiving an LDM (also referred to as a *keepalive* in this context) for some configurable period of time, a switch assumes a link failure in step 1. The detecting switch informs the fabric manager about the failure in

step 2. The fabric manager maintains a logical fault matrix with per-link connectivity information for the entire topology and updates it with the new information in step 3. Finally, in step 4, the fabric manager informs all affected switches of the failure, which then individually recalculate their forwarding tables based on the new version of the topology. Required state for network connectivity is modest, growing with $k^3/2$ for a fully-configured fat tree built from k-port switches.

Traditional routing protocols require all-to-all communication among n switches with $O(n^2)$ network messages and associated processing overhead. PortLand requires $O(n)$ communication and processing, one message from the switch detecting failure to the fabric manager and, in the worst case, n messages from the fabric manager to affected switches.

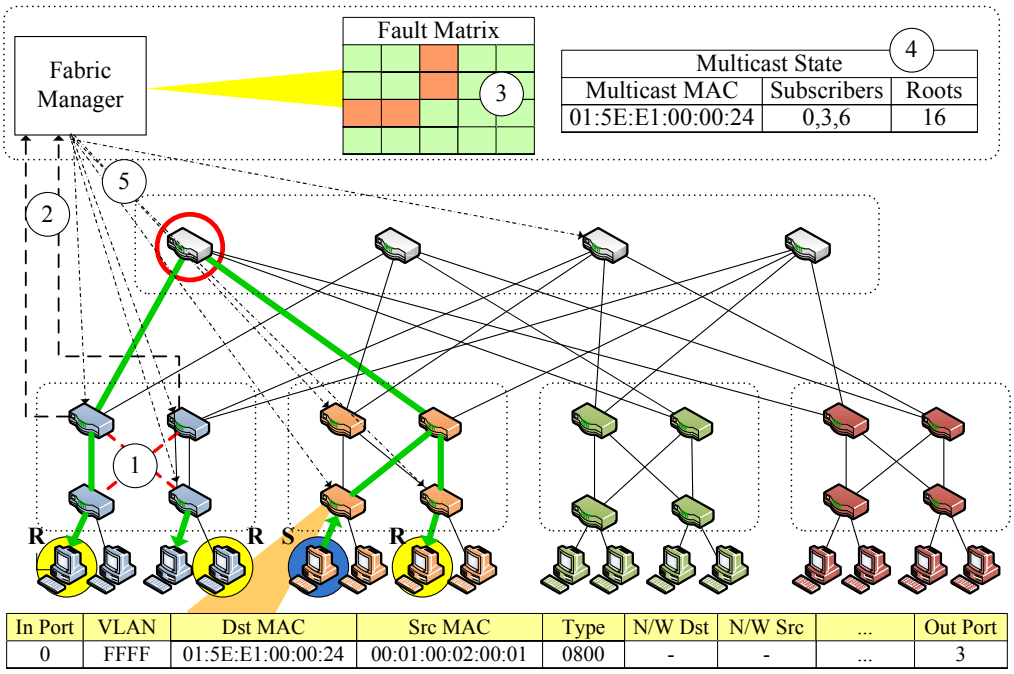


Figure 2.5: Multicast: Fault detection and action.

We now consider fault tolerance for the multicast and broadcast case. Relative to existing protocols, we consider failure scenarios where there is no single spanning tree rooted at a core switch able to cover all receivers for a multicast group or broadcast session. Consider the example in Figure 2.5. Here, we have a multicast group mapped

to the left-most core switch. There are three receivers, spread across pods 0 and 1. A sender forwards packets to the designated core, which in turn distributes the packets to the receivers. In step 1, two highlighted links in pod 0 simultaneously fail. Two aggregation switches detect the failure in step 2 and notify the fabric manager, which in turn updates its fault matrix in step 3. The fabric manager calculates forwarding entries for all affected multicast groups in step 4.

In this example, recovering from the failure requires forwarding through two separate aggregation switches in pod 0. However, there is no single core switch with simultaneous connectivity to both aggregation switches. Hence, a relatively simple failure scenario would result in a case where no single core-rooted tree can cover all interested receivers. The implications are worse for broadcast. We deal with this scenario by calculating a greedy set cover for the set of receivers associated with each multicast group. This may result in more than one designated core switch associated with a multicast or broadcast group. The fabric manager inserts the required forwarding state into the appropriate tables in step 5 of Figure 2.5.

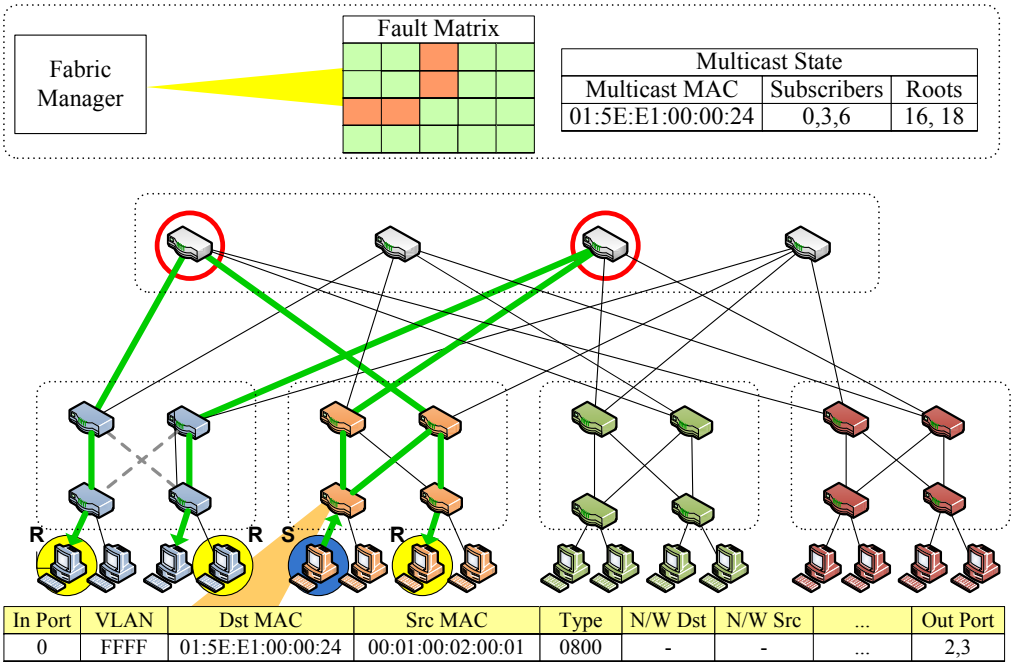


Figure 2.6: Multicast: After fault recovery.

Finally, Figure 2.6 depicts the forwarding state for the sender after the failure recovery actions. The multicast sender's edge switch now forwards two copies of each packet to two separate cores that split the responsibility for transmitting the multicast packet to the receivers.

2.3.7 Discussion

Given an understanding of the PortLand architecture, we now compare our approach to two previous techniques with similar goals, TRILL [PED+09] and SEATTLE [CKR08]. Table 2.1 summarizes the similarities and differences along a number of dimensions. The primary difference between the approaches is that TRILL and SEATTLE are applicable to general topologies. PortLand on the other hand achieves its simplicity and efficiency gains by assuming a multi-rooted tree topology such as those typically found in data center settings.

For forwarding, both TRILL and SEATTLE must in the worst case maintain entries for every host in the data center because they forward on flat MAC addresses. While in some enterprise deployment scenarios the number of popular destination hosts is limited, many data center applications perform all-to-all communication (consider search or MapReduce) where every host talks to virtually all hosts in the data center over relatively small time periods. PortLand forwards using hierarchical PMACs resulting in small forwarding state. TRILL employs MAC-in-MAC encapsulation to limit forwarding table size to the total number of switches, but must still maintain a rewriting table with entries for every global host at ingress switches.

Both TRILL and SEATTLE employ a broadcast-based link state protocol to discover the network topology. PortLand leverages knowledge of a baseline multi-rooted tree to allow each switch to establish its topological position based on local message exchange. We further leverage a logically centralized fabric manager to distribute failure information.

TRILL handles ARP locally since all switches maintain global topology knowledge. In TRILL, the link state protocol further broadcasts information about all hosts connected to each switch. This can add substantial overhead, especially when considering virtual machine multiplexing. SEATTLE distributes ARP state among switches

using a one-hop DHT. All switches register the IP address to MAC mapping for their local hosts to a designated resolver. ARPs for an IP address may then be forwarded to the resolver rather than broadcast throughout the network.

While decentralized and scalable, this approach does admit unavailability of otherwise reachable hosts during the recovery period (i.e., several seconds) after a resolver switch fails. Worse, simultaneous loss of soft state in both the resolving switch and a host's ingress switch may leave certain hosts unreachable for an extended period of time. PortLand protects against these failure conditions by falling back to broadcast ARPs in the case where a mapping is unavailable in the fabric manager and associated state is lost. We are able to do so because the PortLand broadcast protocol is efficient, fault tolerant, and provably loop free.

To protect against forwarding loops, TRILL adds a secondary TRILL header to each packet with a TTL field. Unfortunately, this means that switches must both decrement the TTL and recalculate the CRC for every frame, adding complexity to the common case. SEATTLE admits routing loops for unicast traffic. It proposes a new "group" construct for broadcast/multicast traffic. Groups run over a single spanning tree, eliminating the possibility of loops for such traffic. PortLand's forwarding is provably loop free with no additional headers. It further provides native support for multicast and network-wide broadcast using an efficient fault-tolerance mechanism.

2.4 Implementation

2.4.1 Testbed Description

Our evaluation platform closely matches the layout in Figure 2.1. Our testbed consists of 20 4-port NetFPGA PCI card switches [LMW+07]. Each switch contains 4 GigE ports along with Xilinx FPGA for hardware extensions. We house the NetFPGAs in 1U dual-core 3.2 GHz Intel Xeon machines with 3GB RAM. The network interconnects 16 end hosts, 1U quad-core 2.13GHz Intel Xeon machines with 3GB of RAM. All machines run Linux 2.6.18-92.1.18el5.

The switches run OpenFlow v0.8.9r2 [ope], which provides the means to control switch forwarding tables. One benefit of employing OpenFlow is that it has already

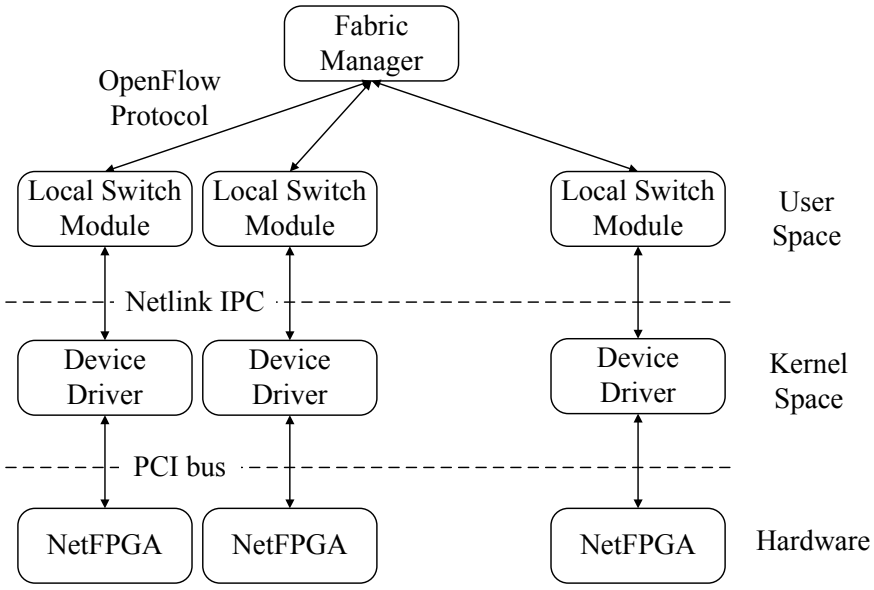


Figure 2.7: System architecture.

been ported to run on a variety of hardware platforms, including switches from Cisco, Hewlett Packard, and Juniper. This gives us some confidence that our techniques may be extended to commercial platforms using existing software interfaces and hardware functionality. Each switch has a 32-entry TCAM and a 32K entry SRAM for flow table entries. Each incoming packet's header is matched against 10 fields in the Ethernet, IP and TCP/UDP headers for a match in the two hardware flow tables. Each TCAM and SRAM entry is associated with an action, e.g., forward the packet along an output port or to the switch software. TCAM entries may contain "don't care" bits while SRAM matches must be exact.

2.4.2 System Architecture

PortLand intercepts all ARP requests and IGMP join requests at the edge switch and forwards them to the local switch software module running separately on the PC hosting each NetFPGA. The local switch module interacts with the OpenFlow fabric manager to resolve ARP requests and to manage forwarding tables for multicast sessions. The first few packets for new flows will miss in hardware flow tables and will

be forwarded to the local switch module as a result. The switch module uses ECMP style hashing to choose among available forwarding paths in the switch and inserts a new flow table entry matching the flow. On receiving failure and recovery notifications from the fabric manager, each switch recalculates global connectivity and modifies the appropriate forwarding entries for the affected flows through the switch.

The OpenFlow fabric manager monitors connectivity with each switch module and reacts to the liveness information by updating its fault matrix. Switches also send keepalives to their immediate neighbors every 10ms. If no keepalive is received after 50ms, they assume link failure and update the fabric manager appropriately.

Figure 2.7 shows the system architecture. OpenFlow switch modules run locally on each switch. The fabric manager transmits control updates using OpenFlow messages to each switch. In our testbed, a separate control network supports communication between the fabric manager and local switch modules. It is of course possible to run the fabric manager simply as a separate host on the data plane and to communicate inband. The cost and wiring for a separate lower-speed control network will actually be modest. Consider a control network for a 2,880-switch data center for the $k = 48$ case. Less than 100 low-cost, low-speed switches should suffice to provide control plane functionality. The real question is whether the benefits of such a dedicated network will justify the additional complexity and management overhead.

Table 2.2 summarizes the state maintained locally at each switch as well as the fabric manager. Here
$k = Number\ of\ ports\ on\ the\ switches$,
$m = Number\ of\ local\ multicast\ groups$,
$p = Number\ of\ multicast\ groups\ active\ in\ the\ system$.

2.5 Evaluation

In this section, we evaluate the efficiency and scalability of our implementation. We describe the experiments carried out on our system prototype and present measurements to characterize convergence and control overhead for both multicast and unicast communication in the presence of link failures. We ran all experiments on our testbed

described in Section 2.4.

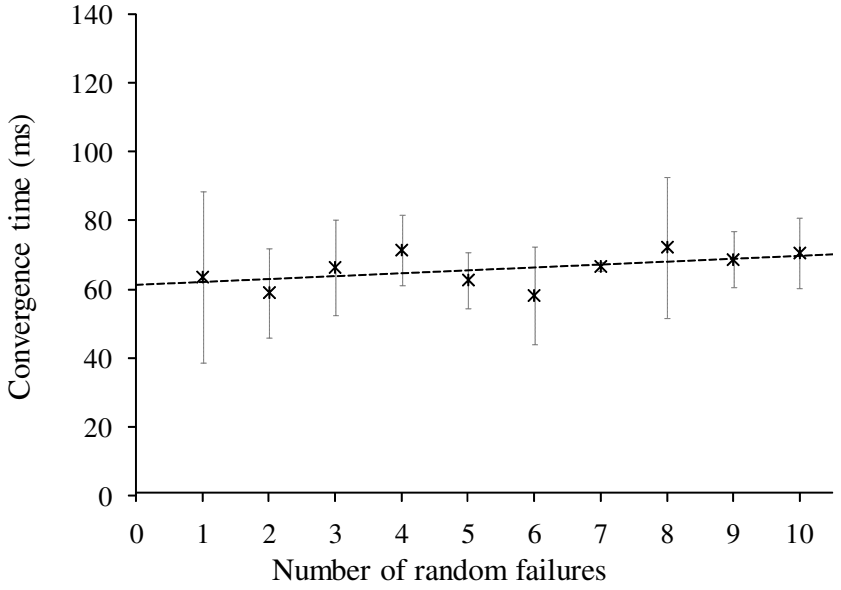


Figure 2.8: Convergence time with increasing faults.

Convergence Time With Increasing Faults We measured convergence time for a UDP flow while introducing a varying number of random link failures. A sender transmits packets at 250Mbps to a receiver in a separate pod. In the case where at least one of the failures falls on the default path between sender and receiver, we measured the total time required to re-establish communication.

Figure 2.8 plots the average convergence time across 20 runs as a function of the number of randomly-induced failures. Total convergence time begins at about 65ms for a single failure and increases slowly with the number of failures as a result of the additional processing time.

TCP convergence We repeated the same experiment for TCP communication. We monitored network activity using tcpdump at the sender while injecting a link failure along the path between sender and receiver. As illustrated in Figure 2.9, convergence

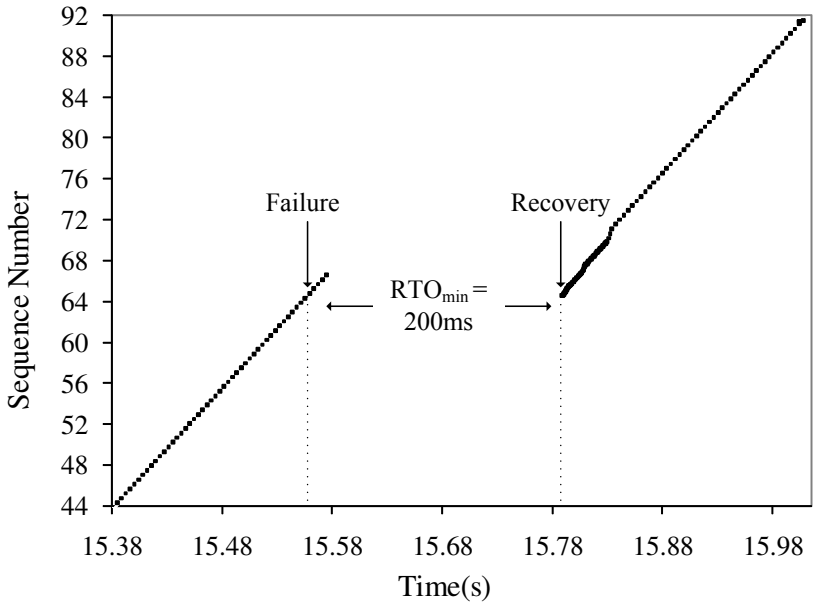


Figure 2.9: TCP convergence.

for TCP flows takes longer than the baseline for UDP despite the fact that the same steps are taken in the underlying network. This discrepancy results because TCP loses an entire window worth of data. Thus, TCP falls back to the retransmission timer, with TCP's RTO$_{min}$ set to 200ms in our system. By the time the first retransmission takes place, connectivity has already been re-established in the underlying network.

Multicast Convergence We further measured the time required to designate a new core when one of the subscribers of a multicast group loses connectivity to the current core. For this experiment, we used the same configuration as in Figure 2.5. In this case, the sender transmits a multicast flow to a group consisting of 3 subscribers, augmenting each packet with a sequence number. As shown in Figure 2.10, 4.5 seconds into the experiment we inject two failures (as depicted in Figure 2.5), causing one of the receivers to lose connectivity. After 110ms, connectivity is restored. In the intervening time, individual switches detect the failures and notify the fabric manager, which in turn reconfigures appropriate switch forwarding tables.

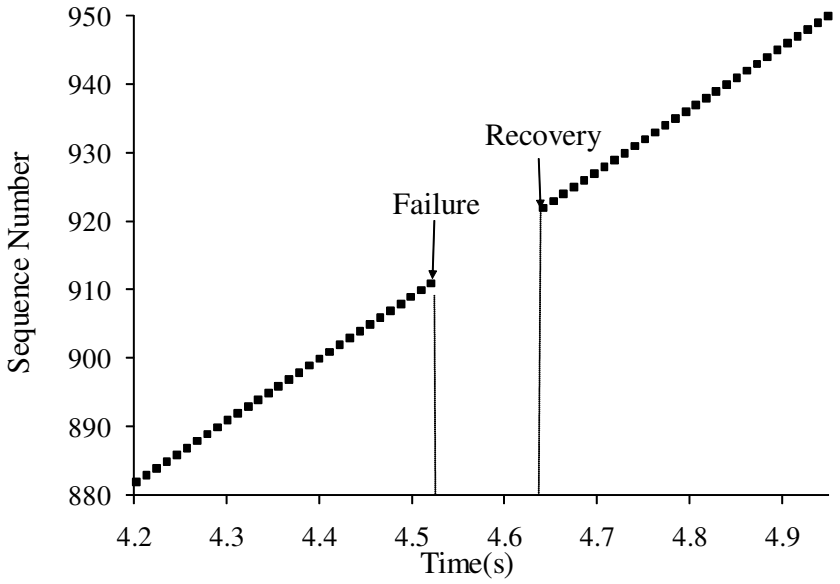


Figure 2.10: Multicast convergence.

Scalability One concern regarding PortLand design is scalability of the fabric manager for larger topologies. Since we do not have a prototype at scale, we use measurements from our existing system to project the requirements of larger systems. Figure 2.11 shows the amount of ARP control traffic the fabric manager would be expected to handle as a function of overall cluster size. One question is the number of ARPs transmitted per host. Since we are interested in scalability under extreme conditions, we considered cases where each host transmitted 25, 50 and 100 ARP requests/sec to the fabric manager. Note that even 25 ARPs/sec is likely to be extreme in today's data center environments, especially considering the presence of a local ARP cache with a typical 60-second timeout. In a data center with each of the 27,648 hosts transmitting 100 ARPs per second, the fabric manager must handle a manageable 376Mbits/s of control traffic. More challenging is the CPU time required to handle each request. Our measurements indicate approximately 25 μs of time per request in our non-optimized implementation. Fortunately, the work is highly parallelizable, making it amenable to deployment on multiple cores and multiple hardware thread contexts per core. Figure 2.12 shows

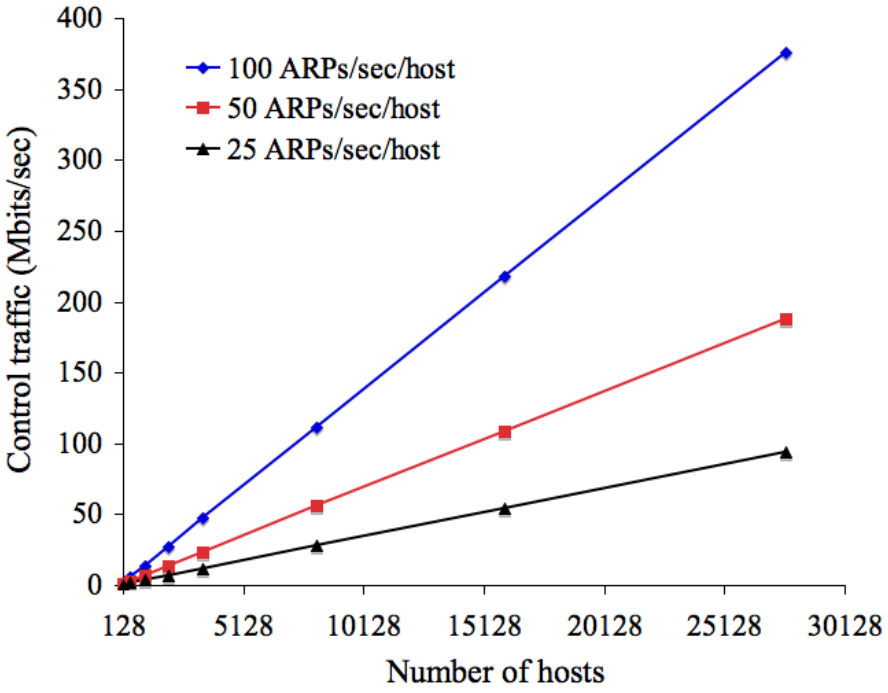


Figure 2.11: Fabric manager control traffic.

the CPU requirements for the fabric manager as a function of the number of hosts in the data center generating different numbers of ARPs/sec. For the highest levels of ARPs/sec and large data centers, the required level of parallelism to keep up with the ARP workload will be approximately 70 independent cores. This is beyond the capacity of a single modern machine, but this also represents a relatively significant number of ARP misses/second. Further, it should be possible to move the fabric manager to a small-scale cluster (e.g., four machines) if absolutely necessary when very high frequency of ARP requests is anticipated.

VM Migration Finally, we evaluate PortLand's ability to support virtual machine migration. In this experiment, a sender transmits data at 150 Mbps to a virtual machine (hosted on Xen) running on a physical machine in one pod. We then migrate the virtual machine to a physical machine in another pod. On migration, the host transmits a gratuitous ARP with its new MAC address, which is in turn forwarded to all hosts communicating with that VM by the previous egress switch. The communication is not

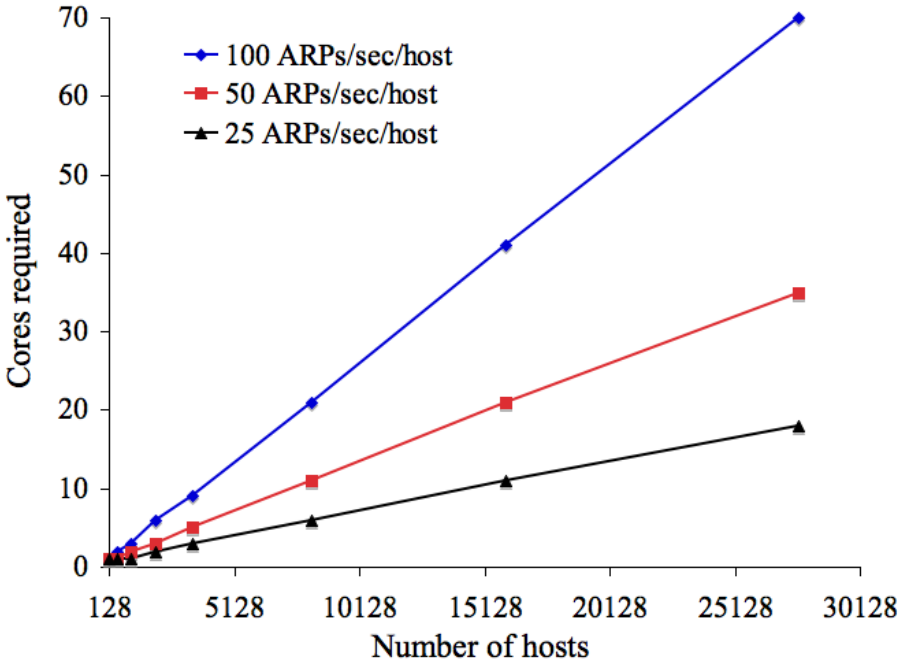


Figure 2.12: CPU requirements for ARP requests.

at line-rate (1 Gbps) since we use software MAC layer rewriting capability provided by OpenFlow to support PMAC and AMAC translation at edge switches. This introduces additional per packet processing latency. Existing commercial switches have MAC layer rewriting support directly in hardware [cisb].

Figure 2.13 plots the results of the experiment with measured TCP rate for both state transfer and flow transfer (measured at the sender) on the y-axis as a function of time progressing on the x-axis. We see that 5+ seconds into the experiment, throughput of the tcp flow drops below the peak rate as the state of the VM begins to migrate to a new physical machine. During migration there are short time periods (200-600ms) during which the throughput of the flow drops to near zero (not visible in the graph due to the scale). Communication resumes with the VM at full speed after approximately 32 seconds (dominated by the time to complete VM state transfer).

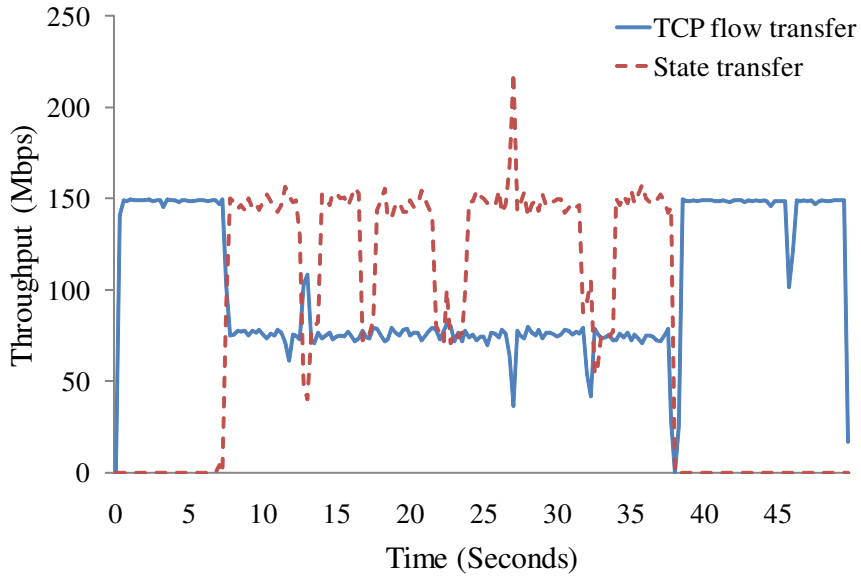


Figure 2.13: State and TCP application transfer during VM migration.

2.6 Summary

The goal of PortLand is to explore the extent to which entire data center networks may be treated as a single plug-and-play fabric. Modern data centers may contain 100,000 hosts and employ virtual machine multiplexing that results in millions of unique addressable end hosts. Efficiency, fault tolerance, flexibility and manageability are all significant concerns with general-purpose Ethernet and IP-based protocols. In this chapter, we present PortLand, a set of Ethernet-compatible routing, forwarding, and address resolution protocols specifically tailored for data center deployments. It is our hope that through protocols like PortLand, data center networks can become more flexible, efficient, and fault tolerant.

2.7 Acknowledgment

Chapter 2, in part, contains material as it appears in the Proceedings of the ACM SIGCOMM 2009 conference on Data communication. Niranjan Mysore, Rad-

hika; Pamboris, Andreas; Farrington, Nathan; Huang, Nelson; Miri, Pardis; Radhakrishnan, Sivasankar; Subramanya, Vikream; Vahdat, Amin. The dissertation author was the primary investigator and author of this paper.

Algorithm 2.1: LDP_listener_thread()

```
1:  While (true)
2:      For each tp in tentative_pos
3:          If (curr_time − tp.time) > timeout
4:              tentative_pos ← tentative_pos − {tp};
5:      ▷ Case 1: On receipt of LDM P
6:      Neighbors ← Neighbors ⋃ {switch that sent P}
7:      If (curr_time − start_time > T and |Neighbors| ≤ k/2)
8:          my_level ← 0; incoming_port ← up;
9:          Acquire_position_thread();
10:     If (P.level = 0 and P.dir = up)
11:         my_level ← 1; incoming_port ← down;
12:     Else If (P.dir = down)
13:         incoming_port ← up;
14:     If (my_level = −1 and |Neighbors| = k)
15:         is_core ← true;
16:         For each switch in Neighbors
17:             If (switch.level ≠ 1 or switch.dir ≠ −1)
18:                 is_core ← false; break;
19:         If (is_core = true)
20:             my_level ← 2; Set dir of all ports to down;
21:     If (P.pos ≠ −1 and P.pos ⊈ Pos_used)
22:         Pos_used ← Pos_used ⋃ {P.pos};
23:     If (P.pod ≠ −1 and my_level ≠ 2)
24:         my_pod ← P.pod;
25:
26:     ▷ Case 2: On receipt of position proposal P
27:     If (P.proposal ⊈ (Pos_used ⋃ tentative_pos))
28:         reply ← {"Yes"};
29:         tentative_pos ← tentative_pos ⋃ {P.proposal};
30:     Else
31:         reply ← {"No", Pos_used, tentative_pos};
```

Algorithm 2.2: Acquire_position_thread()

1: $taken\_pos = \{\}$;
2: **While** $(my\_pos = -1)$
3: $proposal \leftarrow random()\%\frac{k}{2}$, s.t. $proposal \notin taken\_pos$
4: Send proposal on all upward facing ports
5: Sleep(T);
6: **If** (more than $\frac{k}{4} + 1$ switches confirm $proposal$)
7: $my\_pos = proposal$;
8: **If**$(my\_pos = 0)$
9: $my\_pod =$ Request from $Fabric\ Manager$;
10: Update $taken\_pos$ according to replies;

Table 2.1: System comparison

System	Topology	Forwarding		Routing	ARP	Loops	Multicast
		Switch-State	Address-ing				
TRILL	General	*O(number of global hosts)*	Flat; MAC-in-MAC encapsu-lation	Switch broad-cast	All switches map MAC address to remote switch	TRILL header with TTL	ISIS ex-tensions based on MOSPF
SEATTLE	General	*O(number of global hosts)*	Flat	Switch broad-cast	One-hop DHT	Unicast loops possible	New con-struct: groups
PortLand	Multi-rooted tree	*O(number of local ports)*	Hierarch-ical	Location Discov-ery Proto-col; Fabric manager for faults	Fabric manager	Provably loop free; no addi-tional header	Broadcast-free routing; multi-rooted spanning trees

Table 2.2: State requirements.

State	Switch	Fabric Manager
Connectivity Matrix	$O(k^3/2)$	$O(k^3/2)$
Multicast Flows	$O(m)$	$O(p)$
$IP \rightarrow PMAC$ mappings	$O(k/2)$	$O(k^3/4)$

Chapter 3

FasTrak: Enabling Express Express Lanes in Multi-Tenant Data Centers

This chapter focuses on a scalable policy enforcement mechanism that maximises network performance that applications resident on multi-tenant clouds observe. The shared nature of multi-tenant cloud networks requires providing tenant isolation and quality of service, which in turn requires enforcing thousands of network-level rules, policies, and traffic rate limits. Enforcing these rules in virtual machine hypervisors imposes significant computational overhead, as well as increased latency. In FasTrak, we seek to exploit temporal locality in flows and flow sizes to offload a subset of network virtualization functionality from the hypervisor into switch hardware freeing up the hypervisor. FasTrak manages the required hardware and hypervisor rules as a unified set, moving rules back and forth to minimize the overhead of network virtualization, and focusing on flows (or flow aggregates) that are either most latency sensitive or exhibit the highest packets-per-second rates.

3.1 Introduction

'Infrastructure as a Service' offerings such as Amazon EC2 [ec2], Microsoft Azure [azu], and Google Compute Engine [gce] host an increasing fraction of network services. These platforms are attractive to application developers because they transparently scale with user demands. However, the shared nature of cloud networks requires

enforcing tenant isolation and quality of service, which in turn requires a large number of network-level rules, policies, and traffic rate limiting to manage network traffic appropriately. Enforcing these policies comes at some cost today, which will only grow as demand for these offerings increase.

Today, many multi-tenant offerings provide tens of thousands of customers with virtual network slices that can have hundreds of security and QoS rules per VM [VPC]. Such networks must provide isolation for thousands of flows per virtual machine, or tens of thousands of flows per physical server, and up to hundreds of thousands of flows per Top of Rack (ToR) switch. This enormous set of rules must be accessed on a per-packet basis to enable communication and isolation between tenants. The virtual machine hypervisor typically enforces these rules on every packet going in and out of the host.

This results in problems for both the provider and the customer: for the provider, resources that could be used to support more users must instead be diverted to implement these network-level rules; for the customer, the myriad rules implemented within the hypervisor are a source of increased latency and decreased throughput.

In FasTrak, we seek to reduce the cost of rule processing recognizing that the associated functionality is required. We exploit temporal locality in flows and flow sizes to offload a subset of network virtualization functionality from the hypervisor into switch hardware in the network itself to free hypervisor resources. FasTrak manages the required hardware and hypervisor rules as a unified set, moving rules back and forth to minimize the overhead of network virtualization, and focusing on flows (or flow aggregates) that are either most latency sensitive or exhibit the highest packets-per-second rates.

Due to hardware space limitations, only a limited number of rules can be supported in hardware relative to what is required by a server. We argue that this gap is inherent and hence the key challenge we address in this work is identifying the subset of flows that benefit the most from hardware/network offload, and coordinating between applications, VMs, hypervisors, and switches to migrate these rules despite changes in traffic, application behavior, and VM placement.

FasTrak seeks to achieve the following three objectives:

1. Hardware network virtualization: The flows that bypass the hypervisor should still

be subject to all associated isolation rules. This include tunnel mappings, security and QoS rules associated with the flow.

2. Performance isolation: Regardless of whether traffic is subject to rule processing in the hypervisor or in hardware, the aggregate traffic rate of each tenant's VM should not exceed its limits. Likewise, traffic from one tenant VM should not affect another's, even if non-overlapping subset of flows are offloaded to hardware.
3. Performance: We seek to improve both available application bandwidth and to reduce both average and tail communication latency.

Based on these goals, we design, prototype and evaluate FasTrak. We rely on pre-existing technology trends. Hardware with significant levels of network virtualization support, such as tunneling offloads for NICs [emu] and switches [ari] are becoming commodity. The core of our design consists of an SDN controller that decides which subset of active traffic should be offloaded, and a per-VM flow placement module that directs selected flows through either the hypervisor, or through an SR-IOV [sri]-based "FasTrak" path. The flow placement module integrates with an OpenFlow interface, allowing the FasTrak controller to program it. In our design, the network fabric core remains unchanged.

The primary contribution of this work is a demonstration that offloading a subset of flows to an in-network hardware fast path can result in substantial performance gains while minimizing rule processing overhead in servers. We motivate our work with a microbenchmark study that gives us insights into where network latencies and CPU overhead in existing virtualized systems. Further, we examine which types of flows are most subject to this rule processing overhead. We then describe FasTrak design and evaluate it on a testbed. In this evaluation, we find that applications see a $\sim$ 2x improvement in finish times and latencies while server load is decreased by 21%. While the actual benefits of FasTrak will be workload dependent, services that benefit the most are those with substantial communication requirements and some communication locality.

3.2 Background

We first consider some of the characteristics and requirements of multi-tenant virtualized data centers, then briefly summarize host and NIC support for network virtualization.

3.2.1 Requirements of multi-tenant data centers

Multi-tenant data centers employing network virtualization seek an abstraction of a logical private network controlled and isolated in the same way as a dedicated physical infrastructure. The properties typically sought out in such networks are:

- **Control (C1).** A tenant must be able to assign any IP address to their VMs, independent of the IP addresses used by other tenants, or the IP addresses assigned by provider to physical servers. The network must support overlapping tenant IP addresses, e.g., in separate private RFC 1918 [rfc] address spaces.
- **Control (C2).** Security and QoS rules that tenants would apply to traffic in their private deployment should be enforceable in the multi-tenant setting.
- **Isolation (I3).** The provider of a multi-tenant data center must be able to distinguish tenant traffic, enforce tenant-specified rules, and rate limit incoming and outgoing bandwidth allocated to tenant VMs. No single tenant should be able to monopolize network resources.
- **Seamlessness (S4).** VM migration should be transparent to the customer and should not require change in the IP address or connectivity. Security and QoS rules pertaining to the VM should move automatically along with the VM.

These requirements impact the design of the network. To enable **C1**, tenant IP addresses must be decoupled from provider IP addresses. Tenant IP addresses should encode the identities of VMs, while the provider IP addresses should indicate their location and aid in forwarding across the network fabric. Tunneling packets carrying tenant IP addresses across the provider network helps achieve this separation. Further, to distinguish tenant traffic from one another despite overlapping IP addresses, every packet must be tagged with a tenant ID. This tenant ID also helps achieve **I3**, by informing the network of which rules to apply to the packet. For **C1**, the network must maintain

mappings between the tenant VM's IP address and the tenant ID, as well as to the tunnel provider's IP address for every destination VM that a source VM wants to communicate with.

To support **C2**, the network must store and enforce all relevant security and QoS rules associated with a tenant VM. For example, Amazon Virtual Private Cloud(VPC) [amab] allows up to 250 security rules to be configured per VM [VPC]. These rules are typically maintained close to the communicating VMs and applied on a per-packet basis.

The network must support tenant-specific rate limits that can be applied to network interfaces to support **I3**. Finally, to ensure **S4**, the network should migrate all the above rules pertaining to every VM along with the VM. Further, the tunnel mappings should be updated both at source and destination of every traffic flow.

A multi-tenant data center with tens of thousands of tenants must store and orchestrate a considerable amount of network state, and must consult much of this state on a per-packet basis. In terms of manageability, this high volume of state encourages a software approach to network virtualization, where the network state corresponding to every VM is held in the hypervisor of the server on which it is resident.

3.2.2 Virtualized Host Networking

Hypervisor-based

A virtual switch in the hypervisor, called the vswitch, manages traffic transiting between VMs on a single host, and between those VMs and the network at large. In a multi-tenant setting the vswitch is typically configured to isolate traffic of VMs belonging to different tenants. Such a vswitch can tag traffic with tenant IDs, maintain and enforce tunnel mappings, security, QoS rules and interface rate limits.

A widely-deployed vswitch implementation is Open vSwitch(OVS) [ovs]. Apart from invoking the kernel for per-packet processing, OVS provides a user-space component for configuration. In this work, we subject OVS to a series of microbenchmark tests to understand the effect of configuration settings (described in Section 3.3) on software network virtualization overheads. We now describe these configurations.

In its simplest configuration, OVS is a simple L2 software switch (herein referred to as 'Baseline OVS'). Security rules can be configured using the user-space component. When OVS detects traffic that it has not seen before, it forwards these packets to user-space, where the packets are checked against the configured security rules. Then a fast path rule corresponding to this traffic is installed in the kernel component, so that subsequent packets can be handled entirely by the kernel component. This component maintains the rules in an $O(1)$ lookup hash table to speed up per packet processing. We refer to this configuration as 'OVS+Security rules'.

OVS also supports configuration of rate limits using `tc` [tc] on VM interfaces that connect to it. These interfaces are called virtual interfaces (VIF) because they are purely software interfaces. Outgoing traffic from the VM exit these interfaces and are first handled by the vswitch, which imposes the rate limit. The vswitch can then forward this VM traffic on a physical NIC if it is destined to a remote host. Interface limits can be specified for incoming traffic as well. We refer to configuration including this rate limit specification as 'OVS+Rate limiting'.

It is also possible to configure OVS to tunnel packets in and out of the physical servers using VXLAN [vxl] tunneling. VXLAN is a proposed tunneling standard for multi-tenant data centers that also specifies the tunneling encapsulation format. In this configuration, an encapsulation with a destination server IP address is added to VM traffic exiting the server. This address is stripped from incoming packets destined to local VMs. We refer to this configuration as 'OVS+Tunneling'.

Hypervisor Bypass

Due to the overhead of transiting the hypervisor, VMs also have the option of bypassing the vswitch and hypervisor to send traffic in and out of a physical NIC directly. It is necessary that the physical NIC interface is administratively configured to support this configuration for a given VM. Packets are DMAed directly between the NIC and the VM, and thus packets need not be copied to the vswitch. There are two key drawbacks to this configuration, however. First, the NIC port automatically becomes unavailable for use by other VMs; in contrast, the vswitch can help share a physical port across multiple VMs using VIFs. As such this technique is not scalable. Second, the traffic

from the VM is no longer virtualized at the physical server, and thus network policies are no longer properly enforced.

To overcome the first problem, Single Root IO Virtualization (SR-IOV) [sri] allows a single PCIe device to appear as multiple separate physical PCIe devices. With SR-IOV, the NIC partitions hardware resources to support a number of virtual functions (VFs) that can be separately allocated to VMs. These VFs can share a physical port on a NIC up to some limit (e.g., 64). Packets are DMAed to and from the VM directly to the NIC using the VF, avoiding the hypervisor copy and associated context switches. However, the processor and BIOS must support SR-IOV and provide IOMMU [iom] functionality so that the NIC can access VM memory, and deliver received packets directly without hypervisor intervention. VF Interrupts on the other hand are first delivered to the hypervisor. This allows the hypervisor to isolate interrupts securely.

The second issue raised is a key motivation for FasTrak, as supporting flexible offload of network rules to the network is necessary to fully realize the benefits of lower latency and lower CPU involvement in virtualized multi-tenant data centers.

3.3 Potential FasTrak Benefits

Supporting the myriad rules and network policies underpinning multi-tenant data centers has largely become the responsibility of the hypervisor. However, the raw CPU cost on a per-packet basis for supporting network virtualization can be substantial. As virtualized workloads become more communication intensive, and required line rates increase, we expect this overhead to grow. In this section, we examine these overheads, measuring the impact of software processing on application throughput and latency. We then examine the effect of bypassing the hypervisor using SR-IOV. In this way, we can estimate an upper-bound on the potential benefit that FasTrak can deliver, as well as better understand which types of network flows will benefit the most from FasTrak.

3.3.1 Microbenchmark setup

We have setup a small testbed consisting of a pair of HP DL380G6 servers, each with two Intel E5520 CPUs and 24 GB of memory. The servers run Linux 3.5.0-17,

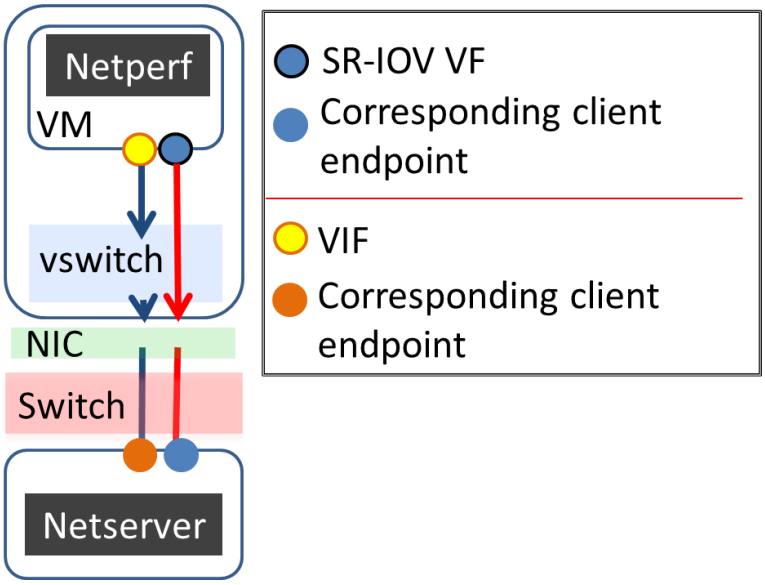


Figure 3.1: Relative network performance measurement setup

the hypervisor is kvm, and guest VMs run Linux 3.5.0. Each VM is equipped with a software virtual interface (VIF) connected to the vswitch (which is OVS), and each NIC also supports an SR-IOV VF. The vswitch can be configured in any of the above mentioned configurations. The network interfaces belonging to VMs and servers used in our tests have TSO and LRO enabled, and the MTU is set to 1500 bytes. The workload we evaluate is netperf [net], measured with four different application data sizes: 64, 600, 1448, 32000 bytes. We first describe how we measure the network overheads of OVS as compared to SR-IOV, and then we describe how we measure CPU overheads.

Measuring network overhead:

Figure 3.1 depicts our experimental setup to quantify the network overhead of virtualization. The three network characteristics we measure and the corresponding experimental setups are:

1. **Throughput:** To measure throughput we attempt to saturate the interface being tested using three netperf threads. These three threads are pinned to three out of four logical CPUs available to the VM, leaving the last CPU for the VM kernel. We use the netperf test TCP_STREAM, and use TCP_NODELAY to ensure that

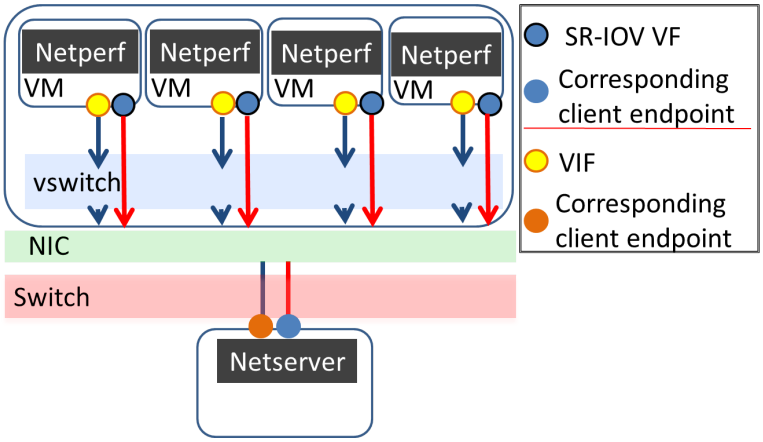


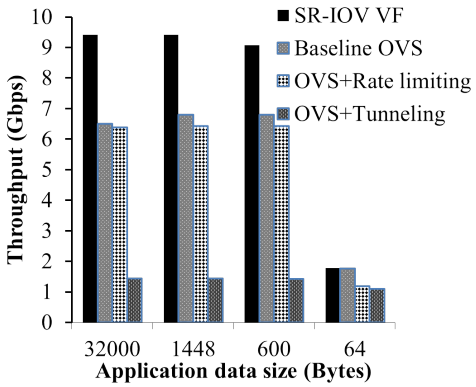
Figure 3.2: Relative CPU overhead measurement setup

netperf sends out data to the interface using the configured application data size. Since we do not limit the amount of CPU available to the VM, we ensure that the throughput measured is the highest achieved with the interface and that the netperf client is not limited by CPU.

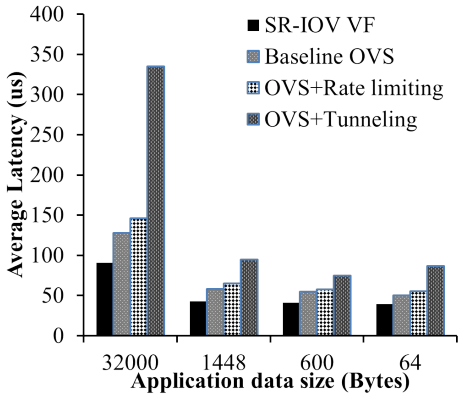
2. **Closed-loop latency:** We use a single netperf thread running TCP_RR to measure latency. In this test, netperf sends out one request at a time and measures the round trip latency to receive the response. Only one request is in transit at a time. We measure both average and 99th percentile latency observed via each interface with this test.
3. **Pipelined latency:** We simulate bursty traffic using three netperf threads running TCP_RR with $burst$ enabled, sending up to 32 requests at a time. Again in this test we try to ensure that the netperf threads utilize maximum possible CPU to achieve high throughput. We measure throughput for this test in terms of transactions per second (TPS) and average latency.

Measuring CPU overhead:

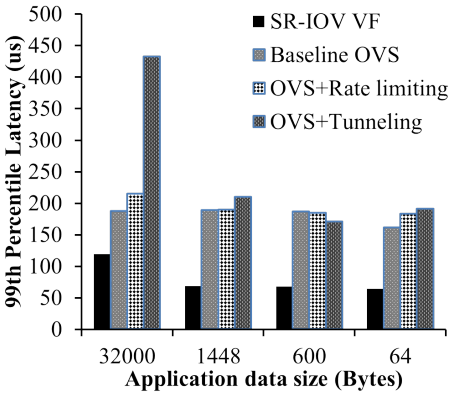
To quantify the CPU required to drive each interface, we rely on the experiment shown in Figure 3.2. Four test VMs on a single physical server each run a single threaded netperf TCP_STREAM test with the TCP_NODELAY option enabled. We measure the total level of CPU utilization during the test.



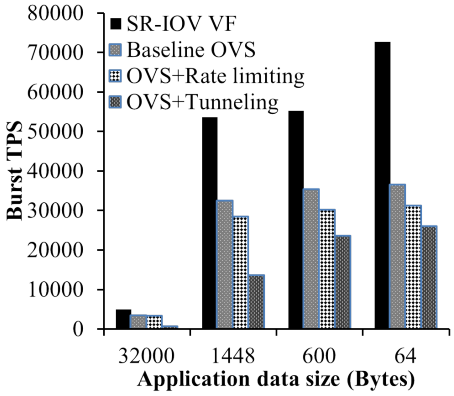
(a) Throughput



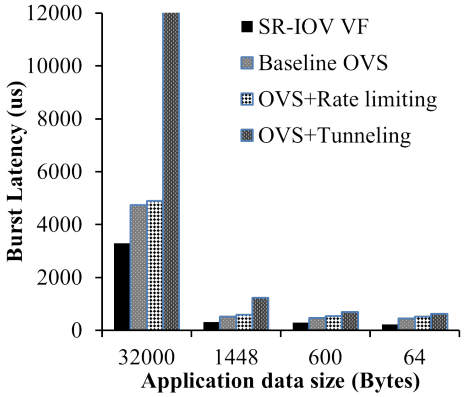
(b) Average Latency



(c) 99th %ile Latency

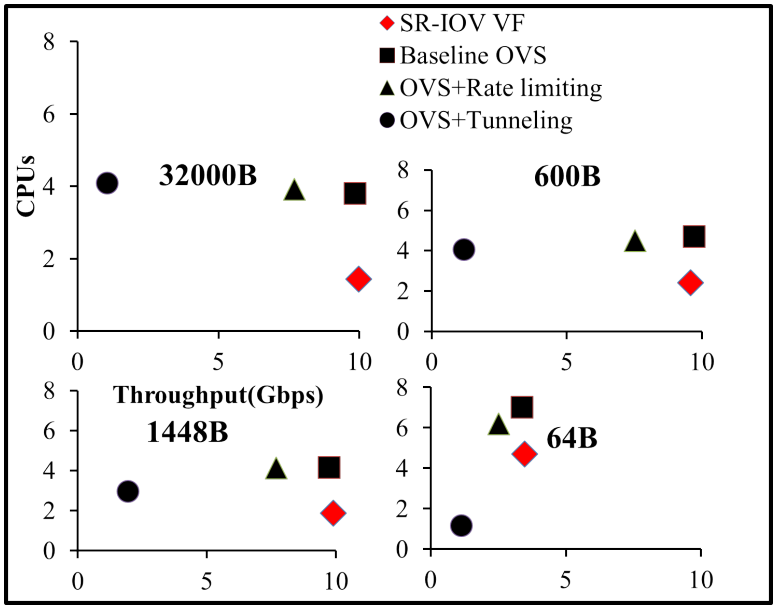


(d) Burst TPS

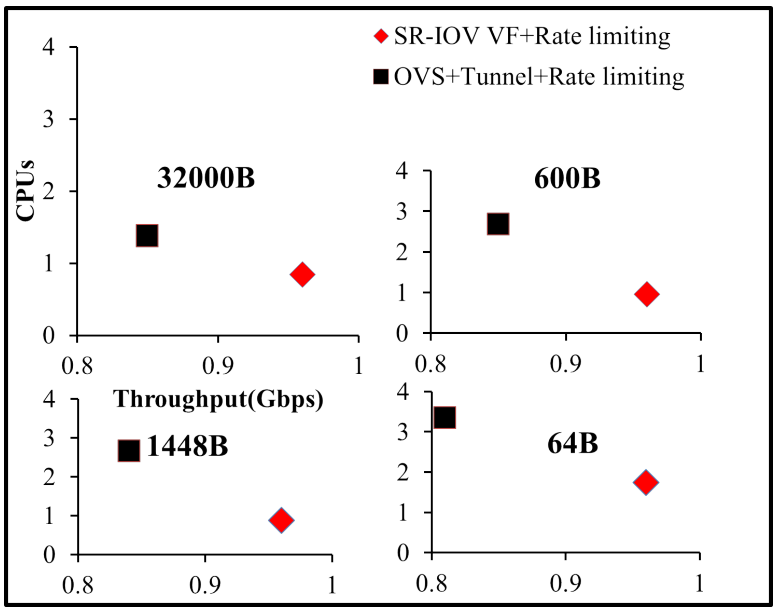


(e) Burst Latency

Figure 3.3: Baseline Network performance



(a) Baseline CPU overhead

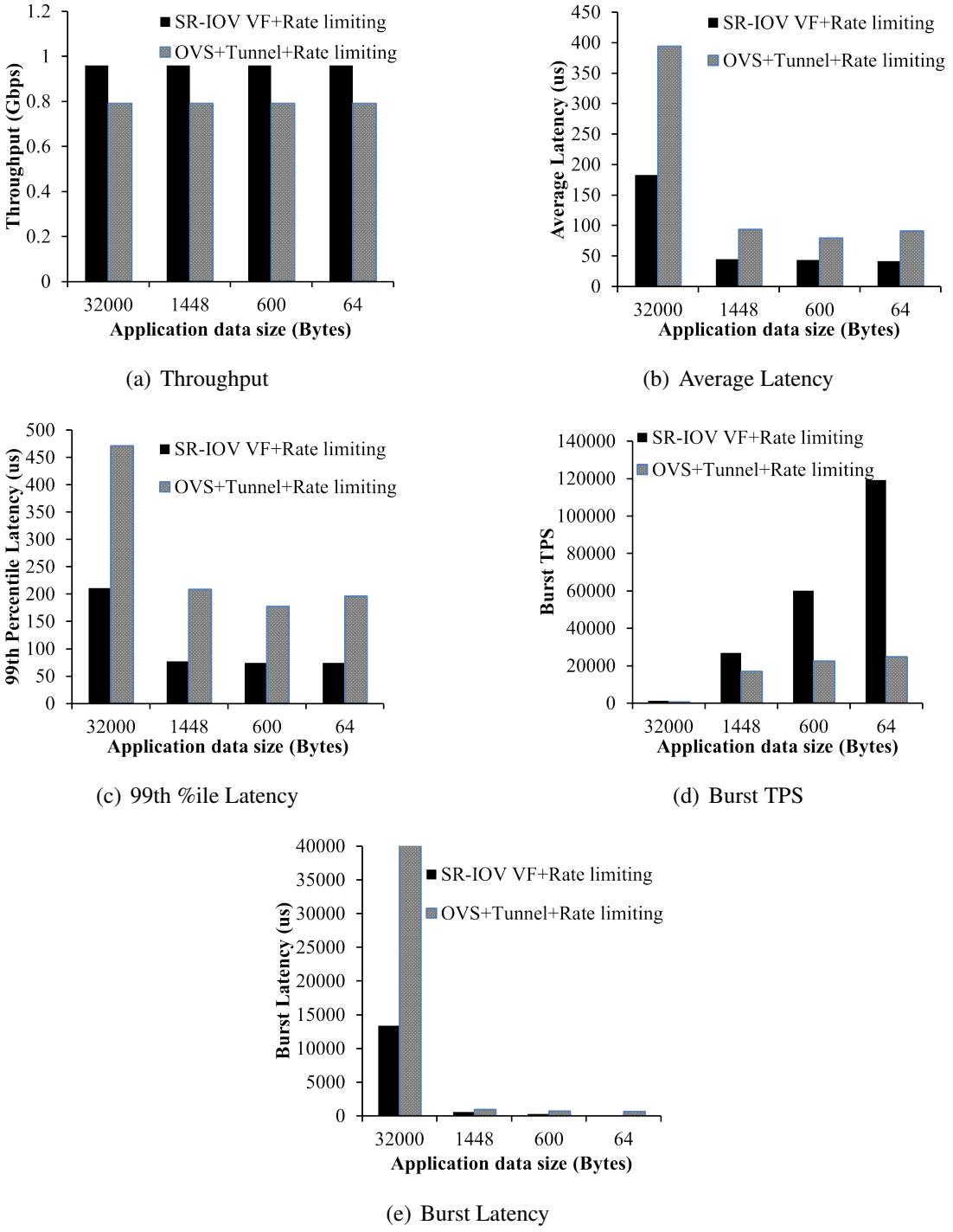


(b) Combined CPU overhead

Figure 3.4: CPU Overheads

3.3.2 Microbenchmark Results

Figure 3.3 shows the results for network performance measurements obtained through the VIF on the hypervisor with three OVS configurations: Baseline, OVS+-



(a) Throughput

(b) Average Latency

(c) 99th %ile Latency

(d) Burst TPS

(e) Burst Latency

Figure 3.5: Combined Network performance

Tunneling and OVS+Rate limiting in comparison with that obtained by bypassing the

hypervisor with SR-IOV. Figure 3.4(a) shows the CPU overhead with the same configurations.

Figures 3.3(a), 3.3(b), 3.3(c) show that SR-IOV delivers significantly higher throughput and lower average and 99th percentile latency. Figure 3.3(d) shows that when pipelining requests, SR-IOV delivers up to twice the transactions per second as compared to baseline OVS. Figure 3.4(a) shows that the CPU required to drive the SR-IOV interface is 0.4-0.7x lower than baseline OVS. With SR-IOV, the hypervisor only isolates interrupts from the VM and leaves the remaining work to the guest VM.

We next traced the system stack (not shown) on those CPU cores dedicated to the host kernel when running the CPU test with baseline OVS. This trace showed that the host CPU spent 96% of time in network I/O (inclusive of servicing interrupts), and up to 55% of time copying data. This is in marked contrast with SR-IOV, where the host CPU was idle 59% of the time and spent 23% of the time servicing interrupts. Thus the OVS overheads are intrinsic to hypervisor packet handling. In fact due to an efficient $O(1)$ implementation for accessing security rules, the same tests measured with an OVS instance populated with $10,000$ security rules showed no measurable difference in overhead compared with baseline OVS.

Overhead of VXLAN tunneling:

The current OVS tunneling implementation was not able to support throughputs beyond 2 Gbps for our target application data sizes. A component of this limitation is that UDP VXLAN packets do not benefit from NIC offload capabilities. Figures 3.3(b) and 3.3(c) show that software tunneling does indeed add to latency. Supporting a link with software tunneling at 1.96 Gbps with TCP requires 2.9 logical CPUs with 1448-byte application data units (shown in Figure 3.4(a)). A stack trace during TCP_STREAM tests showed that tunneling adds 23% overhead relative to baseline OVS. Some of the overhead stems from the fact that the network stack has to additionally decapsulate the packet before it can be processed. We suspect that there are some inefficiencies in current code that causes lookups on VXLAN packets to be much slower than for regular packets. As such the poor performance with tunneling is likely not fundamental, and further engineering should improve the tunneling implementation. Pro-

posals such as STT [stt] also aim to make NIC offloads available to VXLAN packets, and this will further improve throughputs seen with tunneling.

Overhead of rate limiting:

We configured a rate limit of 10 Gbps on the hypervisor VIF while measuring the overhead of queuing and dequeuing packets in `htb`. We found that the latency is somewhat higher (shown in Figures 3.3(b) and 3.3(c)) than baseline OVS, while the throughput (shown in Figure 3.3(a)) is similar. The effect on pipelined latency (shown in Figure 3.3(e)) and transactions per second (shown in Figure 3.3(d)) is more pronounced, with rate limiting reducing the TPS up to 85-88% of baseline.

To measure CPU overhead of rate limiting, we configure a rate limit of 5Gbps for each of three VMs, oversubscribing the 10Gbps physical interface by a factor of 1.5x. As seen in Figure 3.4(a), we cannot achieve line rate with four netperf threads, which we are otherwise able to do with baseline OVS. Furthermore, it requires the same number of logical CPUs as baseline to drive the link, even though we are achieving lower throughput.

Overhead of combined tunneling and rate limiting:

Having observed each software network virtualization functionality in isolation, we now focus on examining the composition of these functions. Because of the inefficiencies in tunneling implementation, we are limited to rates below 1.44 Gbps for these experiments, and so we apply a rate limit of 1 Gbps. We also configure the same rate limit on the SR-IOV interface, and we note that this limit is enforced in the NIC hardware. Figure 3.5 shows the results of the comparison. The average and 99th percentile of latency, the TPS, and pipelined latency of the composed functionality (OVS, tunneling, and rate limiting) are closer to the performance seen with OVS+Tunneling. The pipelined latency is between 1.8-2.1x larger than SR-IOV, as shown in Figure 3.5(e). The CPU overhead of this combination is still 1.6-3x that seen with SR-IOV, as shown in Figure 3.4(b). SR-IOV delivers consistently better throughput overall, as shown in Figure 3.5(a).

Summary:

The above results, and Figure 3.4 show that smaller application data sizes result in higher overall CPU overhead. This is a consequence of virtualization rules being applied on a per-packet basis. As link rates increase, handling each of these per-packet processing steps is increasingly infeasible. Already NICs provide considerable hardware assist to servers to deliver high link rates at lower CPU utilization, including TCP TSO and LRO. By having the hypervisor responsible for enforcing policy on a per-packet basis, we potentially lose the ability to scale with hardware assist in the future.

Baseline OVS without any security rules, without tunneling, and without rate limiting support adds considerable overhead simply due to required kernel crossings. A more subtle effect is that the percentage of improvement in latency increases when a flow is moved from the hypervisor to the SR-IOV interface, based on size. As the application data size decreases, latency improvement increases with hardware offload. For example, when moving flows from baseline OVS to SR-IOV, the pipelined latency improvement increases from 30% for 32000 byte application data sizes to 49% for 64 byte sizes. With OVS and rate limiting, the improvement increases from 32% for 32000 byte application data units to 56% for 64 byte application data units (we omit comparing to tunneling due to the performance issues we observed).

In summary, offloading flows with the highest packet per second rates is going to provide the largest improvements both in terms of CPU overhead and latency overhead. Based on these results, we now consider how FasTrak might benefit both multi-tenant data center customers and providers.

Figure 3.3(d) shows that a bursty application with 64-1448 application data sizes achieves an average TPS of 60K with SR-IOV, and 34K with baseline OVS (or 25K with OVS and tunneling, and 30K with OVS and rate limiting). As an example, achieving a target TPS of 120K requires two VMs with SR-IOV, but four VMs with baseline OVS. Thus, for communication-intensive applications, customers can cut their costs when they use VMs with SR-IOV interfaces.

Looking at this example from the provider's point of view, Figure 3.4(b) shows that, on average, a VIF consumes 1.6-3x the CPU as compared to SR-IOV. If we consider that two logical CPUs are approximately as powerful as an Amazon EC2 medium

instance [ecu12], we can compute an estimate of the potential cost savings. The pricing for this instance type is approximately $821 per year [amaa]. If the provider can save a total of two logical CPUs by avoiding network processing, they can save $821 per server per year.

Given that commodity NICs today support hypervisor bypass through SR-IOV, we now propose a way to leverage that support to lower overall network virtualization costs while improving application performance.

3.4 FasTrak Architecture

Software network virtualization allows for the massive scale required of multi-tenant data centers; but the CPU overheads come at non-zero cost to the provider, and the additional latency can have negative effect on application performance. Commodity network hardware on the other hand can handle a subset of network virtualization functions such as Generic Routing Encapsulation (GRE) [gre] tunnel encap, rate limiting and Access control list (ACL) [acl] or security rule checking, at line rate and lower latency. However, it is fundamentally limited by the number of flows for which it can provide this functionality due to fast path memory limitations. It is also limited in flexibility. For example, it cannot trivially do VXLAN encapsulation until new hardware is built and deployed. FasTrak exploits temporal locality in flows and flow sizes to split network virtualization functionality between hardware and software. It manages the required hardware and vswitch rules as a unified set and moves rules back and forth to minimize the overhead of network virtualization, focusing on flows (or flow aggregates) that are either most latency sensitive or the highest packets-per-second rates.

In this section, we first provide a brief overview of the FasTrak control and data plane functionality. The goal is to provide a comprehensive view of how FasTrak fits in and leverages existing kernel and hardware support to create “express lanes,” or hypervisor bypass paths. We then describe the detailed design of FasTrak that focuses solely on how FasTrak decides which flows to migrate, and how it chooses their path.

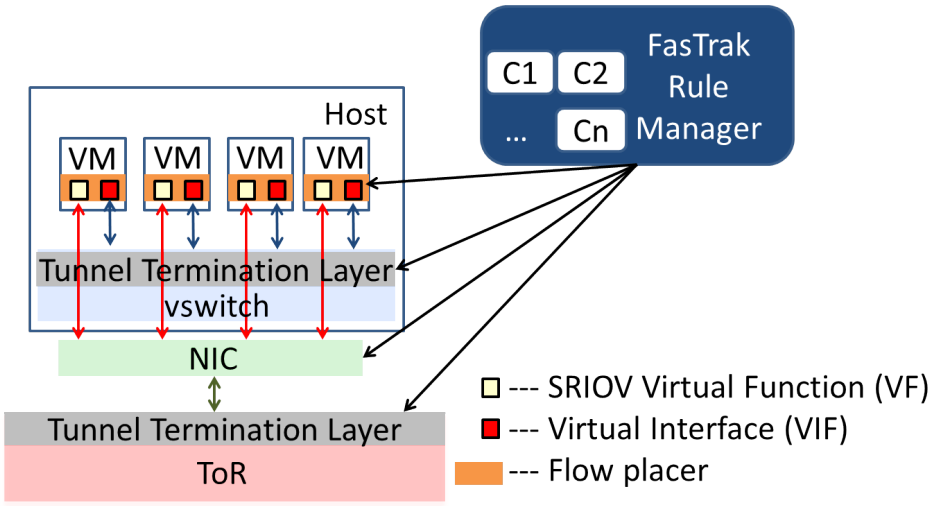


Figure 3.6: FasTrak: Control Plane Overview

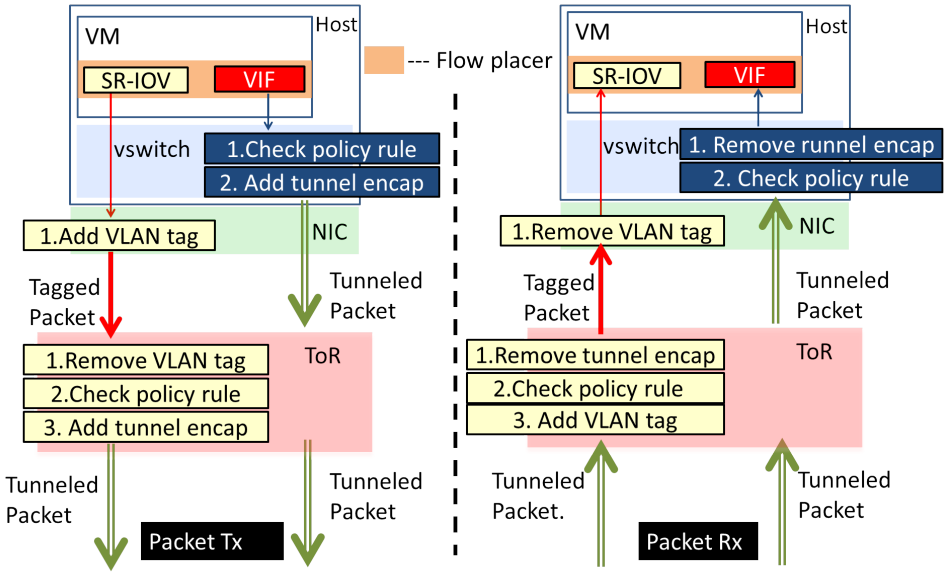


Figure 3.7: FasTrak: Data Plane Overview

3.4.1 Control Plane

Figure 3.6 shows the FasTrak control plane. FasTrak makes two paths available for VM traffic: one through the vswitch, and a second one through the SR-IOV bypass mechanism. The FasTrak rule manager offloads network virtualization rules to and from hardware in response to traffic patterns and VM migration events. These rules not only include tenant security ACLs and QoS rules, but also tunnel mappings required to isolate tenant traffic. We rate limit VM traffic on both the software and hardware paths to a target aggregate, e.g., the total amount of bandwidth available to a tenant/VM.

Enabling hardware 'express lanes'

In FasTrak, each guest VM running on a FasTrak-enabled hypervisor is given two paths to the network: one based on bypassing the hypervisor via SR-IOV, and one based on transiting the default VIF interface through the hypervisor. These are managed transparently to the guest VM by using a bonding driver [bon] for link aggregation. Bonding drivers are able to balance traffic between interfaces. We modify the bonding driver to house a *flow placer* that places traffic on the VIF or SR-IOV VF path. The flow placer module exposes an OpenFlow interface, allowing the FasTrak rule manager to direct a subset of flows via to the SR-IOV interface. It is configured to place flows onto the VIF path by default.

Our design of the flow placer is informed by the design of Open vSwitch. Specifically, the FasTrak rule manager installs wildcard rules in the control plane directing a group of flows matching the wildcard range out of the SR-IOV VF. The flow placer maintains a hash table of exact match rules for active flows in its data plane to allow an $O(1)$ look up when processing packets. When packets from a new flow are not found in this hash table, the packet is passed to the control plane, which then installs a corresponding exact match rule in the hash table. All subsequent packets match this entry in the hash table and are forwarded to the SR-IOV VF. Because the control plane and the data plane of the flow placer exist in the same kernel context, the latency added to the first packet is minimal.

Migrating flows from the network

With express lanes, FasTrak can now choose specific application traffic to offload. Initially, the system uses the policy of migrating flows with high packets-per-second rates to hardware at the switch. FasTrak can also respect customer preferences for performance-sensitive applications as well. A key event in the life-cycle of a VM is its migration. In this case, any offloaded flows must be returned back to the VM's hypervisor before the migration can occur. After VM migration, these flows can again be offloaded at the new destination. By default, FasTrak will migrate flows that no longer have high packet-per-second rates from the network back to the hypervisor.

Hardware-based network virtualization

FasTrak depends on existing hardware functionality in L3 TOR switches to isolate traffic using express lanes. We use Virtual routing and Forwarding (VRF) tables [vrf] available in L3 switches to hold tenant rules. The types of network virtualzation rules offloaded include tunnel mappings, security rules, and QoS rules including rate limits.

Tunneling: We use GRE tunnel capabilities typically available in L3 switches for tunneling offloaded flows. These switches hold GRE tunnel mappings in VRF tables and we leverage this in FasTrak. The GRE tunnel destination IP points to the destination TOR. We reuse the GRE tunnel key to hold the Tenant Id and configure each tunnel belonging to tenant traffic with this key. The GRE key field is 32 bits in size and can accommodate 2^{32} tenants. Without any changes to hardware or router software we can achieve tenant traffic isolation.

Security rules: VRF tables can also be used to hold ACL rules. ACL rules installed in the TOR are explicit 'allow' rules that permit the offloaded traffic to be sent across the network. By default, all other traffic is denied. If a malicious VM sends disallowed traffic via an SR-IOV interface by modifying flow placer rules, the traffic will hit the default rule and be dropped at the TOR.

QoS rules: L3 routers typically provide a set of QoS queues that can be configured and enabled. Rules in the VRF can direct VM traffic to use these specific queues.

Rate limits Enforcement

Usually a rate limit is enforced on each VM interface to only allow as much traffic into and out of the VM as the tenant has paid for. This prevents any single tenant from unfairly depleting available bandwidth on the access link. Since FasTrak exposes two interfaces to each VM by default, it enforces separate rate limits on them such that the aggregate is close to the allowed limit for the VM. These rate limits are calculated using the Flow Proportional Share (FPS) algorithm [RVR+07] applied to periodic traffic demand measurements. As demands change, the rate limits imposed also are changed.

3.4.2 Data Plane

Next we consider the life of packets as they enter and exit VMs. Figure 3.7 shows how the FasTrak data plane functions. All network virtualization functions are taken care of at the vswitch or the first hop TOR. Beyond the TOR, packets are routed through the multi-tenant data center network core as usual.

Packet Transmission

When an application sends traffic using a VM's bonded interface, the packets are looked up in a hash table maintained by the flow placer, and are directed out of the VIF or the SR-IOV VF. Packets that go out via the VIF are virtualized by the vswitch. Such packets tunnel through the NIC and are routed normally at the TOR. All policy checks, tunnel encap and rate limits are enforced by vswitch.

When a packet exits via the SR-IOV VF, the NIC tags it with a VLAN ID (configured by FasTrak) that assists the directly-attached TOR identify the tenant it belongs to. The TOR uses this VLAN tag to find the VRF table corresponding to the tenant (this functionality is typically present in L3 TORs and not unique to FasTrak [cisc]). The TOR then removes the VLAN tag and checks the packet against ACLs installed in that VRF table. The TOR also adds a GRE tunnel encap to the packet based on the tunnel mappings offloaded. The GRE tunnel encap encodes the GRE tunnel destination (TOR), and the GRE tunnel key (tenant ID). At that point, the packet is routed as per the fabric rules based on the GRE tunnel destination.

Packet Reception

When the TOR receives a tunneled packet, it checks the destination IP to see if the packet is destined for it. If the TOR receives a tunneled packet that is not destined for it, it forwards it as per its forwarding tables. Otherwise, it uses the GRE tunnel key (tenant ID) on the packet to identify the VRF table to consult and then decapsulates it. Packets are then checked against the ACL rules in the table, tagged with the tenant VLAN ID and are sent to the destination server. The destination NIC uses the VLAN tag and MAC address on the packet to direct the packet to the right SR-IOV VF after stripping out the VLAN tag, configured by FasTrak.

3.4.3 FasTrak Rule Manager

We now focus on the design of FasTrak rule manager. The rule manager is built as a distributed system of controllers as shown in 3.9. There is a local controller on every physical server, and a TOR controller for every TOR switch. The local controllers on the physical servers coordinate with the TOR controller managing the connected TOR to selectively offload resident VM flows.

Figure 3.8 shows the controller architecture. Both the local and TOR controllers have two components: 1) a measurement engine (ME) and 2) a decision engine (DE). The ME on the local controller periodically measures VM network demand by querying the vswitch for active flows. It conveys this network demand to the TOR controller. The local controller DE installs flow redirection rules in flow placers of co-located VMs. It also decides the rate limits to be configured for each VM interface.

The TOR controller DE receives network demand measurements from attached local controllers. Its ME periodically measures active offloaded flows in the TOR. Based on local and TOR measurements it decides which flows have highest packet-per-second rate and selects them to be offloaded. In the process, offloaded flows that do not have high enough rate can be demoted back to software. The TOR ME also keeps track of amount of fast path memory available in the TOR, so the DE offloads only as many flows as can be accommodated.

We focus next on the ME and DE design.

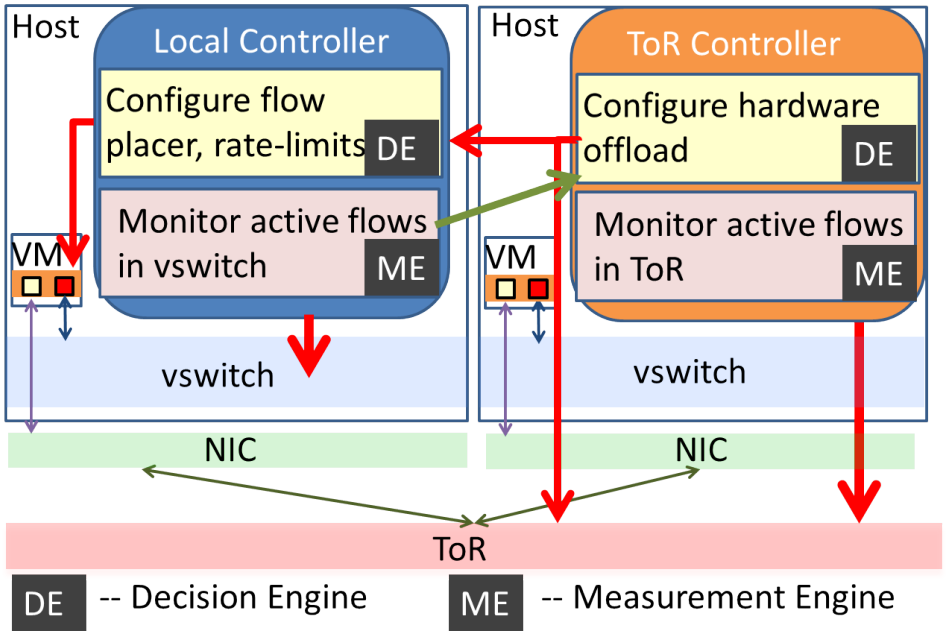


Figure 3.8: FasTrak Controller Architecture

Measurement Engine (ME)

The job of the ME is simple: it collects statistics on packets (p) and bytes (b) observed for every active flow, twice within an interval of t time units. A flow is specified by a 6 tuple: Source and destination IPs, L4 ports, L4 protocol and a Tenant ID. $\frac{\Delta(p)}{t}$ and $\frac{\Delta(b)}{t}$ gives packets per second (pps) and bytes per second (bps) measures for each flow. Pps and bps measurement for active flows is repeated every T time units for N epochs. Every N epochs constitutes a control interval C.

The local controller ME sends a *network demand* report of the form $< flow, pps, bps, epoch\# >$ to the TOR controller every control interval. The report also contains historical information about the median pps and bps seen for flows for the last M control intervals. The TOR controller ME maintains a similar record of active offloaded flows at the TOR for every control interval. At the end of each control interval, these measurements are sent to the TOR DE.

Since the variety of flows seen over time may be large, the ME aggregates flow data periodically. One rule of thumb we use is to aggregate incoming and outgoing flows per VM per application. For example, instead of collecting statistics for every unique 6

tuple, we collect statistics on unique <Source VM IP, Source L4 port, Tenant ID> and <Destination VM IP, Destination L4 port, Tenant ID> flows. Over time, flow statistics are maintained for these aggregates.

The per-VM aggregated flow data collected by the ME forms its *network demand profile*. The network demand profile of a VM contains a history of all flows that either had the VM as a source or as the destination. This network demand profile can be maintained over the lifetime of the VM and is migrated along with the VM. This network demand profile informs FasTrak of the network characteristics of any new VM that is cloned from existing VMs, and allows it to make offload decisions for such VMs on instantiation.

Decision Engine(DE)

The TOR DE receives active flow statistics from local MEs and TOR MEs each control interval. It selects the most frequently used flows having the highest pps rates for offload. Most frequently used flows can be expected to have the highest hit rates. Once the subset of flows has been decided, the TOR DE sends this decision to all connected local controllers. The local DEs in turn configure flow placers on co-resident VMs to redirect the selected flows. They also readjust the rate limits configured on VM interfaces.

Flow placement: The TOR DE uses a simple ranking function to determine the subset of flows to offload. This function ranks flows by giving them a score S. S is calculated using n, the number of epochs the flow was active (i.e., had non-zero pps and bps), and m_pps, the median pps measured over the last N epochs and M control intervals. $S = n \times m\_pps$ ensures that $S \propto n$, the frequency with which the flow is accessed, and $S \propto m\_pps$, the median pps. Flows active both in vswitch and hardware are scored in this fashion.

Certain all-to-all applications may require that all corresponding flows be handled in hardware, or none at all. Such preferences can be given as input to FasTrak. FasTrak encodes these preferences as a number c (indicating tenant priority), and uses $S = n \times m\_pps \times c$ so that higher priority flows get higher scores.

Active flows with highest scores are offloaded, while already offloaded flows

that have lower scores are demoted back to the corresponding vswitch. This decision is conveyed back to the local controllers so they can configure flow placers of co-located VMs.

Rate limiting: By creating alternate express lanes for VM application traffic, we no longer have an aggregation point on which a single rate limit can be enforced. We have to now split up bandwidth in a distributed manner between the VIF and SR-IOV VF.

The local controller DEs decide the ingress and egress rate limit split up for each VM once the TOR DE decides the subset of flows to be offloaded. We use FPS [RVR$^+$07] to calculate the limits L_s and L_h for each VM's VIF and SR-IOV VF. To each of these we add an overflow rate O and install rate limiting rules that allow $R_s = L_s + O$ through the VIF and $R_h = L_h + O$ through SR-IOV VF. R_s and R_h are calculated separately for ingress and egress traffic.

By allowing for O overflow, it is easy to determine when the rate limit on each of these interfaces is overly restrictive. When the capacity required on the interface is higher than the rate limit, the flows will max out the rate limit imposed. FPS uses this information to re-adjust the rates.

3.5 Implementation

3.5.1 Testbed

Our testbed consists of six HP Proliant DL360G6 servers with Intel Xeon E5520 2.26GHz processor and 24 GB RAM. Four of these servers have Intel 520-DA2 10Gbps dual-port NIC that is SR-IOV capable. The other two have Myricom 10 Gbps 10G-PCIE2-8B2-2S+E dual-port NICs. These servers are connected to a Cisco Nexus 5596U-P 96-port 10 Gbps switch running NX-OS 5.0(3)N1(1a). All servers run Linux 3.5.0-17. The hypervisor is kvm, and guest VMs run Linux 3.5.0. On the four servers with the Intel NIC, we configure Open vSwitch(OVS) 1.9.0 with one 10Gbps port. The other 10 Gbps port is configured to support four SR-IOV VFs. We expose a VIF and an SR-IOV VF to each VM.

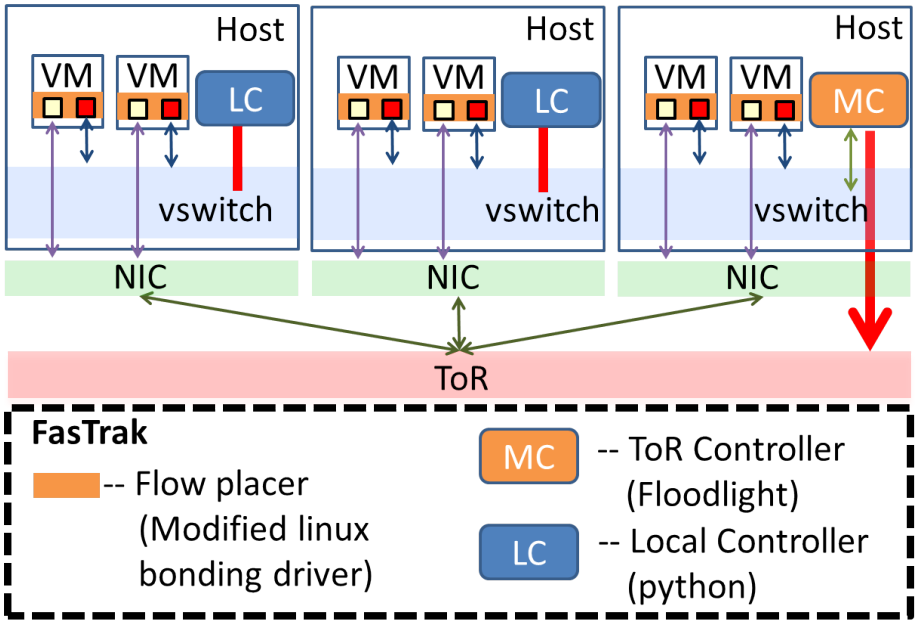


Figure 3.9: Implementation setup

3.5.2 System Architecture

Figure 3.9 shows our implementation set up. The local controller is a python script that queries the OVS datapath for active flow statistics twice within a period of $t = 100ms$ (this period is configurable). This enables measurement of pps and bps for all active flows. This measurement is repeated once every T seconds (we use $T = 5, 0.5$ in different experiments). The flow data is aggregated for N epochs (we use $N = 2$ in our experiments). These measurements are then used to guide the DE.

The TOR Controller is a custom Floodlight controller [flo] that issues OpenFlow table and flow stats requests. Since the Cisco switch is not OpenFlow enabled, we use Open vSwitch to instantiate the TOR switch.

We modify the linux bonding driver [bon] to implement the flow placer. Specifically, we bond the VIF and VF exposed to the VM in bonding-xor mode. We have modified the existing layer 3 and 4 hashing policy to instead direct flows based on the local controller's directive. For this we use a flow hash table which stores an index to the desired outgoing interface (VIF or VF) for every entry.

3.6 Evaluation

We now evaluate the benefits of FasTrak. We choose memcached [mem] as a representative example of a communication intensive application that is network bound. We seek to understand how FasTrak automatically transfers flows onto the hardware path, and to show the resulting benefits as well as costs. While we also evaluated disk-bound applications such as file transfer and Hadoop MapReduce, and found that FasTrak improved their overall throughput and reduced their finishing times, in this evaluation we focus specifically on latency-sensitive applications. In each of the following experiments, we compare to baseline OVS, with no tunneling or rate limiting on the hardware path.

3.6.1 Benefits of hardware offload

We now examine the benefits of offloading flows to the hardware path.

Transaction Throughput

We start by measuring the maximum transaction load in terms of transactions per second(TPS) when accessing the memcached server via the SRIOV interface. The test setup is as described in Figure 3.10. We have three VMs pinned to four CPUs. Each VM is equivalent to an Amazon EC2 large instance [amaa, ecu12] in terms of compute capacity, and has 5GB of memory. The Memcached servers run in two VMs. We use the rest of the five servers as clients to run the memslap benchmark for 90 seconds (only three are shown in the figure for space). The application traffic is routed either via the VIF or via the SR-IOV VF.

Table 3.1: Memcached performance(TPS) w/o background applications

Interface Used	TPS	Mean Latency(us)	# of CPUs for test
VIF	106574	373	3.3
SR-IOV VF	215288	192	3.2

Baseline: With only 2 VMs running memcached servers, Table 3.1 shows the results of the transaction throughput, latency and CPU load seen when using the VIF in

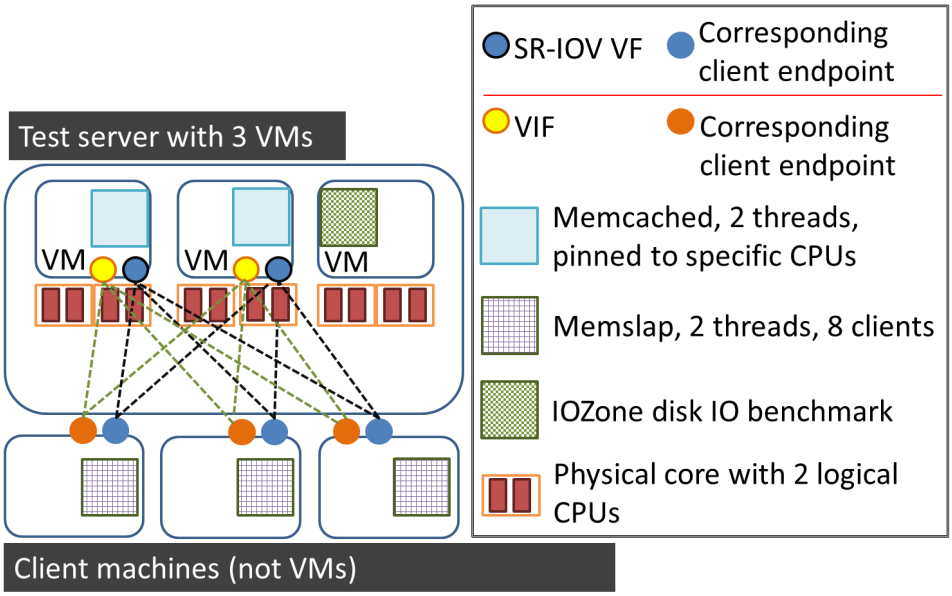


Figure 3.10: Evaluation setup to measure transaction throughput.

comparison to SR-IOV. The same two memcached servers are able to serve twice the number of requests when using the SR-IOV VF with half the latency.

Table 3.2: Memcached performance(TPS) w/ background applications

Interface Used	TPS	Mean Latency(us)	# of CPUs for test
VIF	96093	414	4.1
SR-IOV VF	177559	231	4.1

With background traffic: We repeat the baseline experiment, but this time with the third VM running the IOzone Filesystem Benchmark [ioz]. Table 3.2 shows that the presence of background applications does not alter the overall performance. We also introduced background noise into the VM using the stress tool [str], and in this case also the application achieved higher throughput and lower latency when using the SR-IOV VF, while using the same amount of CPU.

Application finish times

In this set of experiments we evaluate the finish times of the application as seen by the clients, both through the hypervisor and through the SR-IOV interface. The test

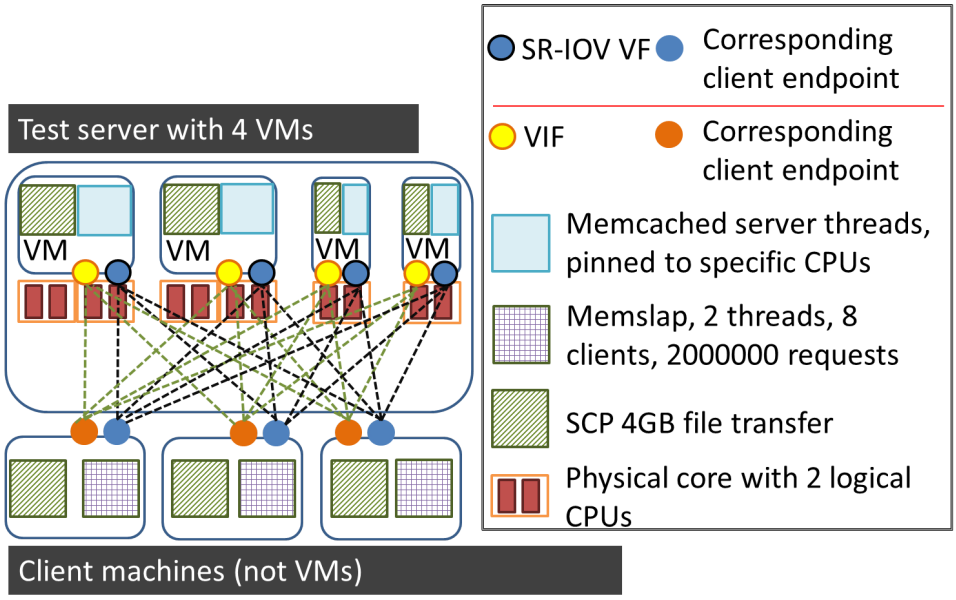


Figure 3.11: Evaluation setup to measure application finish times

setup is described in Figure 3.11. We run four VMs on the test server. Two of these VMs run with 4 CPUs and 5 GB of memory. The other two run with 2 CPUs and 2.5 GB of memory. The latter are equivalent to Amazon EC2 medium instance in terms of compute. All the VMs run a memcached server instance. Again we use the rest of the five servers as clients to run the memslap benchmark, this time each issuing a total of 2M requests to the four memcached servers.

Table 3.3: Memcached performance(Finish times) as the servers are shifted to use the SR-IOV VF

Percentage traffic through VIF	**Mean Finish Times(s)**	**Mean TPS**	**Mean Latency(us)**	**# of CPUs for test**
100%	86.6	23089	331	3.5
75%	82.2	24333	306	3.2
50%	82.3	24335	297	3.2
25%	82.1	23976	275	2.9
0%	54.9	37456	190	2.2

Finish times during flow migration: Table 3.3 shows the the average finishing times at the clients, the average TPS achieved, and the average latency and number of

CPUs used when all the memcached servers are accessed using the software VIF. In this case, the percentage of traffic forwarded through VIF is 100%. We then repeat the experiment, routing traffic from one memcached server via the SR-IOV VF, while using the VIF for the others. In this case, the percentage of traffic through the VIF is 75%. We repeat this exercise, routing traffic from two, three and finally all four memcached servers via SR-IOV so the percentage of traffic forwarded through the hypervisor is zero.

Our results show that the average finish time reduces by 37% when we move all the memcached traffic to SR-IOV VFs. Before that, when some memcached traffic uses the VIF and some use the SR-IOV VFs, the finish times at the client are dominated by the slower memcached traffic. This result confirms the observation that the performance of an application is often dominated by the speed of the slowest flow.

Table 3.4: Memcached performance(Finish times) w/ background applications

Interface Used	**Mean Finish Times(s)**	**Mean TPS**	**Mean Latency(us)**	**# of CPUs for test**
VIF	118.4	16896.2	455.6	7.6
SR-IOV VF	69	29334.6	249	6.3

Finish times with background traffic: We next test the finish times observed at clients when we additionally have a 4GB file transfer which is disk bound ongoing at each of the VMs hosting the memcached servers. This transfer uses the VIF. This experiment is a precursor to our next one where we show how FasTrak shifts flows selectively to SR-IOV VF. Table 3.4 shows that finish times almost double when the memcached traffic uses the VIF, and latency reduces by half.

3.6.2 Flow migration with FasTrak

A key responsibility of FasTrak is managing the migration of flow rules to and from the hypervisor and switch. In this section, we evaluate the impact that migration has on applications.

Application finish times

We now examine the effect of flow migration on the finishing time of an application. We retain the same test set up as the previous experiment, but we now use FasTrak to monitor and measure the throughputs of a file transfer application (scp) and Memcached traffic. The flow placer in the bonding driver directs all traffic via the VIF by default. Within 10 seconds the local controller detects that scp flows are averaging at 135 pps per VM for outgoing traffic, and 115.5 pps per VM for incoming traffic (consisting primarily of acks). In contrast, the memcached flows are averaging at 5618 pps per VM for outgoing traffic (replies) and 5616 pps per VM for incoming traffic (requests). While in this particular example we could offload both applications, we have modified FasTrak to offload only one. As such, FasTrak chooses the memcached flows, and these are shifted to use the SR-IOV VF after 10 seconds.

Table 3.5: Memcached performance(Finish times) w/ background applications using FasTrak

Interface Used	**Mean Finish Times(s)**	**Mean TPS**	**Mean Latency(us)**	**# of CPUs for test**
VIF only	110.9	18044.2	440.2	7.6
VIF(10s) + SR-IOV(rest)	57.34	35339.8	225.6	6.0

Table 3.5 shows the results of this experiments. With FasTrak, Memcached finishes about twice as fast with about half the average latency.

FasTrak costs

FasTrak controllers use negligible CPU once during each measurement and decision period. More importantly, there is a potential impact on existing TCP flows during periods of reconfiguration. To examine this, we offload a single iperf TCP flow one second after it begins, and capture a packet trace at the receiver. We also used netstat at the sender and receiver to collect transport-level metrics. We found that as a result of this offload operation, one delayed ack was sent, TCP recovered twice from packet loss, and there were 30 fast retransmits. When the shift happens, some packets that return

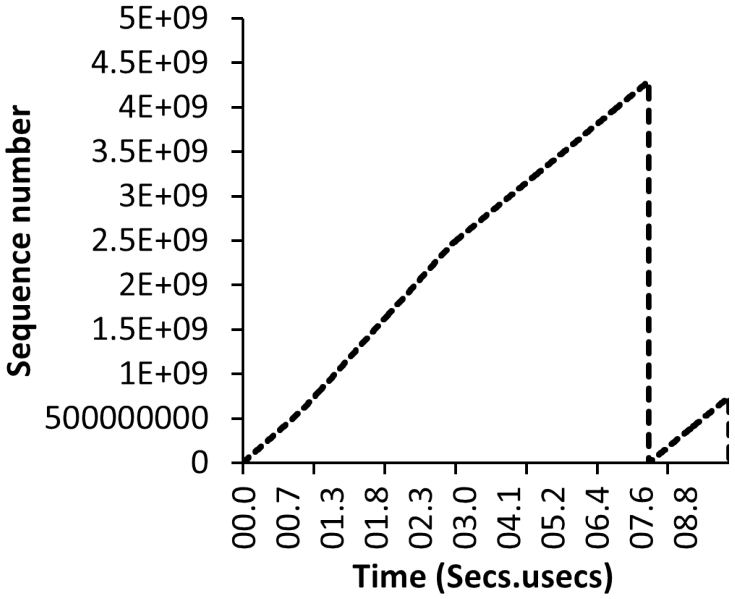


Figure 3.12: TCP progression

via the VIF were lost. A packet capture at the receiver, shown in Figure 3.12, shows the progression of the connection.

3.7 Related Work

Recently there have been several proposals for achieving network virtualization in multi-tenant settings [MYM+11, PYK+10, SKG+11, CKRS10, ovs, hyp, mid]. These suggest that the vswitch is the best place to achieve scalable tenant network communication, security, and resource isolation. NIC tunnel offloads are being proposed and standardized [emu, stt] to reduce some of the network processing burden from software. Tunnel termination network switches [ari] are being introduced at the boundary of virtualized environments and non-virtualized environments, typically configured with static tunnels. However, security rule checking and rate limiting are still largely retained in the vswitch and L4 software or hardware middleboxes.

More broadly, there is a large body of work related to network rule management that we take inspiration from [YRFW10, CMT+11, PKC+12, KCG+10, CFP+07]. Our work concentrates on the specific problem of rule management in multi-tenant networks,

to improve application performance and reduce server load.

The work that closest shares our goals is vCRIB [MYSG13], which proposes a unified rule management system that splits network rules between hypervisors and network hardware. The primary difference between these works is their criteria for migrating flows. FasTrak aims to offload flows with the highest overheads at the hypervisor to maximize the benefits of reducing CPU overhead and latency. FasTrak also differs from vCRIB in that FasTrak does not rely on the vswitch to handle all of the flows (which vCRIB does, if only to redirect a subset of them to other points in the network for security and QoS processing). Finally, unlike vCRIB, FasTrak completely avoids traffic redirection, allowing flows to take the paths prescribed by the network. We note that FasTrak is complementary to vCRIB, and is able to work in tandem with it to manage flows based on a large range of criteria.

The effect of processor context switches, copies, and interrupt overhead on I/O-intensive virtualized environments is well known [AA06,RS07,Liu10]. Several [DYR08, DYL+10, GAH+12] have explored the use of SR-IOV in virtualized environments to overcome these overheads. We build upon these efforts to forward flows over both SR-IOV and hypervisor-based paths.

3.8 Summary

Multi-tenant data centers hosting tens of thousands of customers rely on virtual machine hypervisors to provide network isolation. The latency and CPU costs of processing packets in the hypervisor in this way is significant, increasing costs for both providers and tenants alike.

FasTrak seeks to reduce the cost of rule processing, while maintaining the associated functionality. It exploits temporal locality in flows and flow sizes to offload a subset of network virtualization functionality from the hypervisor into switch hardware in the network itself to free hypervisor resources. FasTrak manages the required hardware and hypervisor rules as a unified set, moving rules back and forth to minimize the overhead of network virtualization, and focusing on flows (or flow aggregates) that are either most latency sensitive or exhibit the highest packets-per-second rates. We find

that applications see a $\sim$ 2x improvement in finish times and latencies while server load is decreased by 21%. While the actual benefits of FasTrak are workload dependent, services that should benefit the most are those with substantial communication requirements and some communication locality.

3.9 Acknowledgment

Chapter 3, in part, contains material submitted for publication as "Fastrak: Enabling express express lanes in multi-tenant data centers" Niranjan Mysore, Radhika; Porter, George; Vahdat, Amin. The dissertation author was the primary investigator and author of this paper.

Chapter 4

Gestalt: Unifying Fault Localization for Networked Systems

This chapter focuses on developing a fault localization algorithm for large networked systems, including data center networks from first principles. Researchers have proposed many algorithms for localizing faults in networked systems, but it is unclear which algorithm is best suited for a given network; the performance of these algorithms differs markedly for different networks. We develop a framework that can explain these differences by anatomizing the algorithms into their basic choices and analyzing these choices with respect to six defining characteristics of real networks. Our analysis also reveals that no existing algorithm simultaneously provides good localization accuracy and low computational overhead. Based on our insights, we develop a new algorithm called Gestalt. To perform well across a range of networks, it combines the good choices of existing algorithms and includes a new method to explore the space of possible faults in a way that is both low overhead and robust to noise. We apply it to three real, diverse networks, an email network, a peer-to-peer messaging system hosted in a data center, and an ISP network. In each case, Gestalt has either significantly higher localization accuracy or an order of magnitude faster running time.

4.1 Introduction

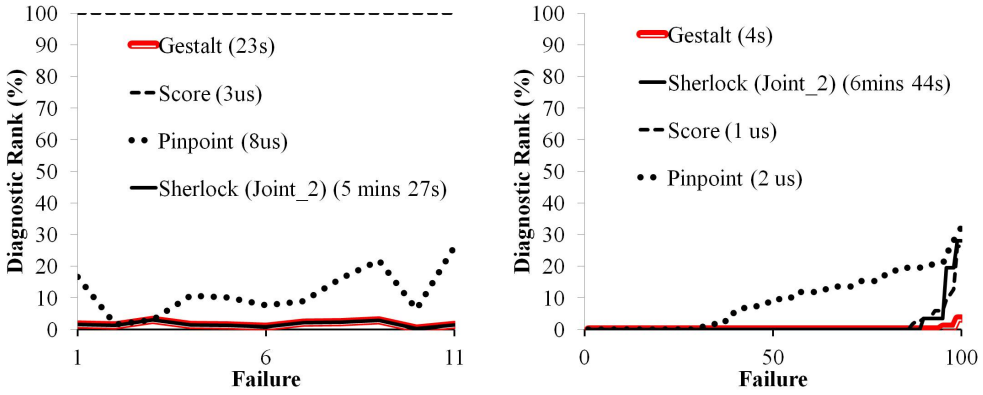
> Gestalt is a general description for concepts that make unity and variety in design. — *Jim Saw*

Consider a large system of routers and servers interconnected by network paths. Such a system could be for integrated audio, video, and text messaging (e.g., Microsoft Lync [lyn12]), for email (e.g., Microsoft Exchange), or even for simple packet delivery (e.g., Abilene). When transactions such as connection requests fail, network operators find it helpful to have a *fault-localization* tool that identifies components likely to have failed. An effective tool allows operators to quickly replace faulty components or implement work-arounds, this increasing the availability of mission-critical networks.

As an example, we conducted a survey of call failures in the Lync messaging system deployed inside data centers belonging to a large corporation. We found that the median time for diagnosis, which was largely manual, was around 8 hours because the operators had to carefully identify the failed components from a large number of possibilities. This time-consuming process is frustrating for operators and leads to significant productivity loss for other employees. A good fault localization tool that can identify a short list of potential suspects in a short amount of time would greatly reduce diagnosis time. Later in this paper, we will show how our fault localization tool, Gestalt, accomplishes this task. With a median running time of under 30 seconds, Gestalt reduces by 60x the number of components that an operator must consider for diagnosis.

Of course, we are not the first to realize the importance of fault localization; many researchers have developed a range of algorithms (e.g., [KYY+95, BCG+07] [KYGCS05, CKFF02, KMV+09, KKV05, KYGCS07, DTDD07, ALMP10]). We contend, however, that existing work has two significant drawbacks: *lack of understanding* and *inadequate performance*.

First, we have consistently heard from operators (e.g., at both Google and Microsoft) that the efectiveness of existing fault localization algorithms depends on the network, and that this dependence is mysterious. There are no studies that connect network characteristics to the choice of algorithm; thus, determining an appropriate fault localization approach for a given network is difficult.



(a) Lync (real failures) (b) Exchange (simulated failures)

Figure 4.1: Applying different algorithms to two systems. Legend shows median time to completion.

Figure 4.1 illustrates this difficulty by running three prior algorithms on two different networks. We picked these algorithms because they use disparate techniques. In the graphs, the y-axis is the diagnostic rank, which is the percentage of network components deemed more likely culprits than the components that actually failed; thus, lower values are better. The failures are sorted by diagnostic rank. We will provide more experimental details in §4.9,

The left graph shows the results for the Lync deployment mentioned above. We see that the algorithms perform differently. Sherlock [BCG+07] does best, and SCORE [KYGCS05] does worst. The right graph shows the results for simulated failures in an Exchange deployment [CZMB08]. We see that the algorithms exhibit different relative performance. SCORE matches Sherlock, and Pinpoint does worst. Further, the appropriate approach for the two networks differs—Sherlock for Lync, and SCORE for Exchange as it combines high localization accuracy and fast running time.

Second, existing algorithms either have poor localization accuracy in the presence of impairments such as noise or have large computational costs for large networks. This tradeoff can be seen in Figure 4.1. While SCORE runs in a few microseconds, it localizes faults poorly for Lync. On the other hand, while Sherlock [BCG+07] has good performance for both networks, it can take a long while to run. In large networks, this time can be days. Running time matters because recovery cannot begin till the algorithm

completes. Our results consistently reveal a tradeoff between localization accuracy and run time in prior work.

Rather than develop yet another localization algorithm with its own poorly understood tradeoffs, we first develop a framework to understand the design space and answer the basic question: When is a given fault localization approach better and why? We observe that existing fault localization algorithms can be anatomized into three parts that correspond to how they model the system, how they compute the likelihood of a component failure, and how they explore the state space of potential failures. Delineating the choices made by an algorithm for each part paves the way for systematically analyzing the algorithm's behavior.

Our anatomization also explains pheonemena found empirically (but not fully explained) in existing work. For example, [KYGCS07] discovers that noise leads the SCORE [KYGCS05] inference algorithm to produce many false positives; the authors suggest mitigation through an additional step of candidate selection using adhoc thresholds. By contrast, we show that the design choices that SCORE makes are inherently sensitive to noise, and changing these would lead to more robust fault localization than the suggested heuristics. As a second example, the Pinpoint algorithm is shown by the authors [CKFF02] to have poor accuracy for even two simultaneous failures. We later show that this problem is fundamentally caused by how Pinpoint explores the state space of failures.

We use our understanding to devise a new fault localization algorithm, called Gestalt. Gestalt combines the best features of existing algorithms to work well in all networks and conditions. While Gestalt benefits from reusing existing components, it also introduces a new method for exploring the space of potential failures that may be of independent interest. This methods represents a continuum between the extremes of greedy failure hypothesis exploration (e.g., SCORE) and combinatorial exploration (e.g., Sherlock). The result is a fault localization algorithm that has both good localization accuracy and low computational cost.

The contributions and a rough outline of this paper are:

1. Anatomization: We show how existing fault localization algorithms can be broken down into a common framework with three parts in Sections 3 and 4. Table 4.3

shows how nine different algorithms map to this framework.

2. Characterization: Section 5 defines six salient network characteristics that pose a challenge to fault localization: noise, uncertainty, covering relationships, simultaneous failures, collective failures, and scale. Section 6 describes our analysis methodology and three disparate real networks (Lync, Exchange, and Abilene) that we use. This is used in Section 7 to discover the relationship between network characteristics and fault localization choices. Table 4.4 summarizes our findings.

3. Design: Our findings lead us to our new algorithm, Gestalt described in Section 8. In Section 9, we show that Gestalt has better diagnostic accuracy or lower overhead than each existing algorithm on all three networks we study. For real Lync failure data, Gestalt improves localization time by an order of magnitude.

4.2 Related Work

Network diagnosis can be thought of as having two phases. The first consumes available information (e.g., log files, passive or active measurements) to estimate system operation and is often used to *detect* faults. Several system-specific techniques exist for this phase [CZMB08, AMW+03, RWM+06, LCD04, MSWA03, DTDD07, KBMJ+08, MGS+09, OA11, NKN12, CTFD09, MGS+09]. Its output is often fed to a second phase that *localizes* faults. Localization identifies which system components are likely to blame for failing transactions.

Fault localization techniques are extremely valuable because information on component health may not be easily available in large networks and manual localization can lead to several hours of downtime. Even where component health information is available, it may be incorrect (as in the case of "gray failures" in which a failed component appears functional to liveness probes) or insufficient towards identifying culprits for failing transactions [BCG+07]. Fault localization has also been studied widely [KYGCS05, CKFF02, BCG+07, KMV+09, DTDD07, ALMP10, KKV05, KYGCS07, KYY+95, Ris05, SS04a]. We focus on this second phase and ask: *given information from the first phase, which fault localization algorithm gives the best accuracy with the lowest overhead, and why?*.

Some diagnostic tools like [MGS+09, NKN12, OA11] leave fault localization to a knowledgeable network operator and aim to provide the operator with a reduced dependency graph for a particular failure. While this is different from what we call fault localization in this paper, the automated fault-localization techniques we discuss can be used in those tools as well to narrow down the list of suspects.

The only survey of fault localization we know is by Steinder and Sethi [SS04b], which considers each approach separately. To the best of our knowledge ours is the first work to analyze the design space for fault localization, and to use this insight to propose a better fault localization tool Gestalt.

4.3 Fault Localization Anatomy

We consider the following common fault localization scenario. The network is composed of many components such as routers and servers. The success of a transaction in the network depends on the health of the components it exercises. The goal of fault localization is to identify components that are likely responsible for failing transactions. While we use the term transaction for simplicity in this paper, it can be any indicator of network health (e.g., link load) for which we want to find the culprit component.

More formally, the state of the network is represented by a vector I with one element $I[j]$ per network component that represents the health of component j. Let O be a vector of observation data such that $O[k]$ represents whether transaction k succeeded. For example, O could represent the results of pings between different sources and destinations. The broad goal is to infer likely values of I that explain the observations O. Specifically, the fault localization algorithm outputs a sequence of possible state vectors $I_1, I_2, ..$ ordered in terms of likelihood.

We measure the goodness of an algorithm by its *diagnostic rank*: given ground truth about the actual components that failed denoted by I_{true}, the diagnostic rank is j if $I_{true} = I_j$ for some j in the output sequence, and n otherwise. For example, a network with two routers R and S and one link E between them will have a 3 element state vector denoting the states of R, S, and E respectively. Let us say that only router R has failed so $I_{true} = (F, U, U)$ where F denotes failed and U denotes up. If the output of the fault

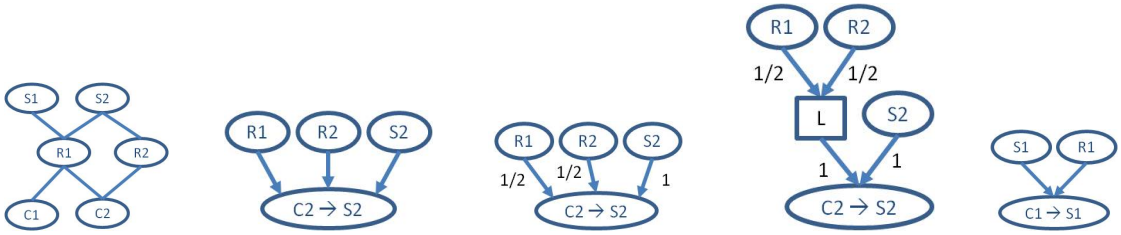
localization algorithm is $(U, F, U), (F, U, U), (U, U, F)$ then the diagnostic rank on this instance of running fault localization is 2 because one other component failure (router S) has been considered more likely. Lower diagnostic rank implies fewer "false leads" that an operator must investigate. A second metric for an algorithm is the computation time required to produce the ranked list given the observation vector O.

We find that practical fault localization algorithms can be anatomized into three parts: a system model, a scoring function, and a state-space explorer. First, any fault localization algorithm needs information such as which components are exercised by each transaction, and possible failure correlations between component failures (e.g., a group of links in a load-balancing relationship). Thus, localization algorithms start with a **system model** S that predicts the observations produced when the system is in state I. System models in past work are often cast in the form of a *dependency graph* between transactions and components but there is considerable variety in the dependency graphs used.

Second, in theory fault localization can be cast as a Bayesian inference problem. Given observation O, rank system states I based on $P_S(I|O)$, the probability that I led to O when passed through the system model S. However, even approximate Bayesian inference [MWJ99,Hec89] can seldom handle the complexity of large networks [KKV05]. So practical algorithms use a heuristic scoring function $Score$ that maps each component to a metric that represents the likelihood of that component failing. The underlying assumption is that for two system states I_i and I_j and respective observations O_i and O_j predicted by S: $P_S(I_i|O) \geq P_S(I_j|O)$ when $Score(O_i, O) \geq Score(O_j, O)$, where O is the actual observation vector. This **scoring function** is the second part of the pattern.

Finally, given the system model and scoring function the final job of a fault localization algorithm is to list and evaluate states that more likely to produce the given observation vector. But system states can be exponential in the number of components since any combination of components can fail. Thus, localization algorithms have a third part that we call **state space exploration** in which heuristic algorithms are used to explore system states, balancing computation time with accuracy.

We do not claim that this pattern fits all possible fault localization algorithms. It does not fit algorithms based on belief propagation [Ris05, SS04a]; such algorithms



(a) Network (b) DTL: $C_2 \rightarrow S_2$ (c) PTL: $C_2 \rightarrow S_2$ (d) PML: $C_2 \rightarrow S_2$ (e) $C_1 \rightarrow S_1$

Figure 4.2: An example network and models for two transactions.

are computationally expensive and have not been shown to work with real systems. As shown in Table 3, this pattern does capture algorithms that have been evaluated for real networks, despite considerable diversity in this set.

4.4 Design Space for Localization

We map existing algorithms into the three-part pattern by describing the choices they make for each part. §4.4.1-4.4.3 describes the choices, and §4.4.4 provides the mapping.

Prior algorithms also use different representations such as binary [KYGCS05, CKFF02, KYGCS07] or probabilities [KMV+09]) for transaction and component states. We use the 3-value representation from Sherlock [BCG+07] as it can model all prior representations. Specifically, the state of a component or transaction is a 3-tuple, $(p^{up}, p^{troubled}, p^{down})$, where p^{up} is the probability of being healthy, p^{down} that of having failed, and $p^{troubled}$ that of experiencing partial failure; p^{up}+$p^{troubled}$+p^{down}=1. The state of a completely successful or failed transaction or component is (1,0,0) or (0,0,1); other tuples represent intermediate degrees of health. A monitoring engine determines the state of an transaction in a system specific way; for example, a transaction that completes but takes a long time may be assigned $p^{troubled} > 0$.

Table 4.1: Transaction state (p^{up}) predicted by different models for transaction $C_2 \rightarrow S_2$ in Figure 4.2

	Failed component (s)			
	R1	R2	S2	R1&R2
DTL	0	0	0	0
PTL	1/2	1/2	0	3/4
PML	1/2	1/2	0	0

4.4.1 System Model

A system model encodes how network components impact transactions. It can be viewed as a directed graph where an edge from A to B says that A impacts B, or B depends on A. We find three system models used by localization algorithms in th literature:

1. Deterministic Two Level (DTL) is a two-level model in which the top-level corresponds to system components and the bottom level to transactions. Components connect to dependent transactions whose success or failure they impact. The model assumes components independently impact dependent transactions. A transaction fails if any of its parent components fails.

2. Probabilistic Two Level (PTL) is similar to DTL except that the impact is modeled as probabilistic. Component failure leads to transaction failure with some probability.

3. Probabilistic Multi Level (PML) can have more than two levels; intermediate levels help encode more complex relationships between components and transactions such as load balancing and failover.

We use the example network in Figure 4.2(a) to illustrate the three models. The network has two clients (C_1, C_2), two servers (S_1, S_2), two routers (R_1, R_2), and several links. Transactions are requests from a client to a server ($C_i \rightarrow S_j$). Each request uses the shortest path, based on hop count, between the client and server. Where multiple shortest paths are present, as for $C_2 \rightarrow S_2$, requests are load balanced across those paths.

Assume that the components of interest for diagnosis are the two routers and the two servers. Then, Figures 4.2(b)-(d) show the models for the transaction $C_2 \rightarrow S_2$.

Different models predict different relationships between the failures of components and that of the transaction. These predictions are shown in Table 4.1. For ease of exposition, the table shows the value of p^{up}; $p^{down} = 1 - p^{up}$ and $p^{troubled} = 0$ in this example. DTL predicts that the transaction fails when any of the components upon which it relies fails. Thus, the transaction is (incorrectly) predicted as always failing even when only one of the routers fails. PTL provides a better approximation in that the transaction is not deemed to completely fail when only one of the router fails. However, it still does not correctly model the impact of both routers failing simultaneously. PML is able to correctly encode complex relationships. While this example shows how PML correctly captures load balancing, it can also model other relationships such as failover [BCG+07]. However, this higher modeling fidelity does not come for free; as we discuss later, PML models have higher computational overhead.

In this network, the three models for the other three types of transactions ($C_1 \rightarrow S_{\{1,2\}}, C_2 \rightarrow S_1$ are equivalent. The model for $C_1 \rightarrow S_1$ is shown in Figure 4.2(e)

4.4.2 Scoring function

Scoring functions evaluate how well the observation vector predicted by the system model for a system state matches the actual observation vector. Let $(p^{up}, p^{troubled}, p^{down})$ be the state of a transaction in the predicted observation vector, and let $(q^{up}, q^{troubled}, q^{down})$ be the actual state determined by the monitoring engine. Then, the computation of various scoring functions can be compactly explained using the following quantities:

Explained failure	$eF = p^{down} q^{down}$
Unexplained failure	$nF = (1 - p^{down}) q^{down}$
Explained success	$eS = p^{up} q^{up} + p^{troubled} q^{troubled}$
Unexplained success	$nS = (1 - p^{up}) q^{up} +$ $(1 - p^{troubled}) q^{troubled}$

eF is the extent to which the prediction explains the actual failure of the transaction, and nF measures the extent to which it does not. eS and nS have similar interpretations for successful transactions. We also define another quantity $TF = \Sigma(eF + nF)$, where the

Table 4.2: Score computed by different scoring functions for three possible failures.

	Failed component		
	R1	R2	S1
ΣeF	3	1	2
ΣnF	0	2	1
ΣeS	0	0	1
ΣnS	1	1	0
FailureOnly (ΣeF/TF)	1	1/3	2/3
InBetween (ΣeF/(TF+ ΣnS))	3/4	1/4	2/3
FailureSuccess (ΣeF+ΣeS)	3	1	3

summation is over all elements of observation vectors. Because $eF + nF = q^{down}$, TF is the total number of failures in the actual observation vector.

Different scoring functions aggregate these basic quantities across observation elements in different ways. We find three classes of scoring functions:

1. FailureOnly (eF, TF): Such scoring functions only measure the extent to which a hypothesis explains actual failures. It thus uses only eF and TF to construct the measure.

2. InBetween (eF, nS, TF): Such scoring functions only measure the extent to which a hypothesis explains failures and unexplained successes.

3. FailureSuccess (eF, eS): Such scoring functions measure *both* the extent to which a hypothesis explains failures and how well it explains successes.

Concrete instances of these classes are shown in Table 4.3. As expected, the score increases as eF and eS increase, and decreases when nF and nS increase. Given the large number of elements, each aggregates across elements in a way that is practical for high-dimensional spaces [Agg01, BGRS99].

Instead of analyzing every instance, in this paper we use a representative for each of the three classes. We have verified that the performance of different functions in a class is qualitatively similar. Our experiments use as representatives the functions used by SCORE (FailureOnly), Pinpoint (InBetween), and Sherlock (FailureSuccess).

To understand how different scoring functions can lead to different diagnoses, consider again the example in Figure 4.2. Assume that R_1 has failed and the actual state of four transactions is available to us. Two of these are $C_1 \rightarrow S1$, both of which have failed (since they depend on $R1$); and the other two are $C_2 \rightarrow S_2$, one of which has failed (because it used $R1$, while the other used $R2$). Table 4.2 shows how the three scoring functions evaluate three system states in which exactly one of R1, R2, and S1 has failed. The computation uses DTL for the system model. The top four rows show the values of the basic quantities. As an example, ΣeF is 3 in Column 1 because R1's failure correctly explains the three failed transactions; it is 1 in Column 2 because R2's failure explains the failure of only one transaction ($C_2 \rightarrow S_2$) and not of the two $C_1 \rightarrow S_1$ transactions.

The bottom three rows of the table show the scores of the three scoring functions for each failure. Even in this simple example, different scoring functions deem different failures as more or less likely. FailureOnly and InBetween deem $R1$ as the most likely failure that explains the observed data, FailureSuccess deems (incorrectly) that the data can be just as well be explained by the failure of $S1$. While it may appear that FailureSuccess is a poor choice, we show later that FailureSuccess actually works reasonably well in a variety of real networks.

4.4.3 State space exploration

State space exploration determines how the potentially large space of possible system states (combinations of failed components) is explored. We find four types of explorers used in prior localization algorithms.

1. Independent only explores system states with exactly one component failure.

2. Joint$_k$ explores system states with at most k failures. It is a generalization of Independent (which is Joint$_1$.

3. Greedy set cover (Gsc) is an iterative method. In each iteration, a single component failure that explains the most failed transactions is chosen, and all explained observations are removed. Iterations repeat until all failed transactions are explained. Thus, it greedily computes the set of component failures that cover all failed transactions.

4. Hierarchical is also an iterative method. As in Gsc, in each iteration the component

C that best explains the actual observations is chosen. However, a major difference is that if there additional observations that C impacts, then these are added to the list of unexplained failures even if they were originally not marked as having failed in the input. Thus unlike Gsc, the set of unexplained failures need not decrease monotonically.

4.4.4 Mapping fault localization algorithms

Table 4.3 maps the fault localization portion of nine prior fault localization algorithms to our framework. Readers familiar with a tool may not immediately see how its computation maps to the choices shown because the original description uses different terminology. But in each case we have analytically and empirically verified the correctness of the mapping: composing the choices shown for the three parts leads to a matching computation (except for aspects mentioned below). Due to space constraints, we omit the results that verify these mappings.

The last column lists fault localization aspects not captured in our framework. Many of these relate to pre- or post-processing data. For example, candidate preselection removes irrelevant components at the start. The table does not list other suggestions by tool authors such as using priors that capture baseline component failure probabilities.

While the mechanisms we do not model are useful enhancements, they are complementary to the core localization algorithm. Our goal is to understand the behavior of fundamental choices made in the core algorithm. By employing these choices, tools inherit their implications (§4.7) even when they use additional enhancements. Our paper abuses notation for simplicity; when we refer to a particular tool by name, we are referring to the computation that results from combining its three-part choices.

4.5 Network Characteristics

Fault localization would be simple if modern networks were simple — in which, for instance, the knowledge of dependencies between components and transactions were perfect, the logged status of transactions were always accurate, and multiple failures were rare. But modern networks are anything but simple, and localization algorithms must handle network characteristics that confound inference. Selecting a localization

approach requires understanding which characteristics are dominant for a given network.

The six characteristics we study are:

1. Uncertainty Most networks have significant non-determinism that makes the impact of a component failure on a transaction uncertain. For example, if a DNS translation is cached, a Ping need not consult the DNS server: thus if the entry is cached, the DNS server failure does not impact the Ping transaction, but otherwise it does. Note that the localization algorithm is not privy to the state of the DNS caches. Load balancing is another common source of non-determinism as is the case for $C_2 \rightarrow S_2$ transaction in Figure 4.2.

More precisely, if a component potentially (but not always) impacts a transaction failure, we say that the dependency is *uncertain*. A network whose system model contains uncertain edges is said to exhibit uncertainty. The degree of uncertainty is measured by the number of uncertain dependencies and the uncertainty of each dependency. Probabilistic models like PTL and PML can naturally encode uncertainty while deterministic models cannot.

2. Observation noise So far, we assumed that observations are measured correctly. However, in practice, pings could be received correctly but lost during transmission to the stored log: thus an "up" transaction can be incorrectly marked as "down". Errors can also occur in reverse. In Lync, for example, the monitoring system measures properties of received voice call data to determine that a voice call is working; however, the voice call may still have been unacceptable to the humans involved. Both problems have been encountered in real networks [KYGCS07, KYGCS05, ALMP10, DTDD07]. They can be viewed as introducing noise in the observation data that can lead sensitive localization algorithms astray. A network with 10% noise can be thought of as flipping 10% of the transaction states before presentation to the localization algorithm.

3. Covering relationships In some systems, when a particular component is used by an transaction, other components are used as well. For example, when a link participates in an end-to-end path, so do the two routers on either end. More precisely, component C covers component D if the set of transactions that C impacts is a superset of the transactions that D impacts.

Covering relationships confuse fault localization because any failed transaction

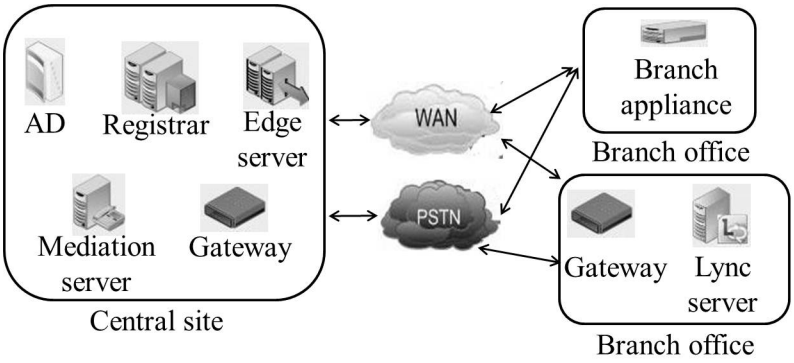


Figure 4.3: Lync architecture.

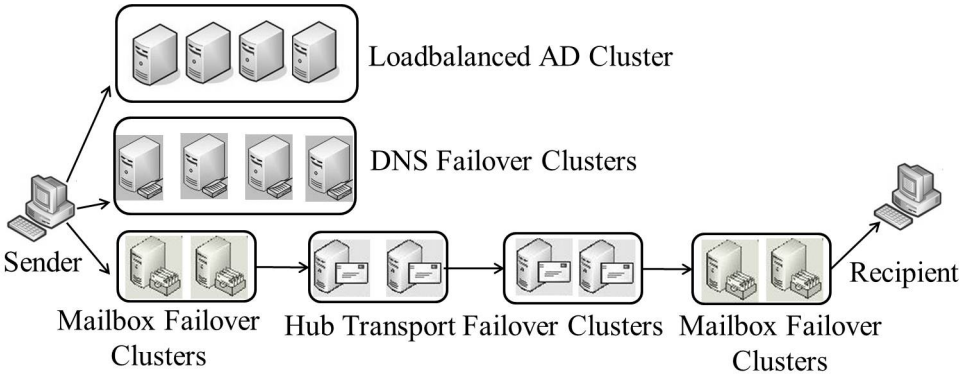


Figure 4.4: Exchange architecture.

explained by the covered component (link) can also be explained by the covering component (router). Other observations can be used to differentiate such failures; when a router fails, there may be path failures that do not involve the covered link. But some fault localization methods are better than others at making this distinction.

4. Simultaneous failures Diagnosing multiple, simultaneous failures is a well-known hurdle. Investigating k simultaneous failures among n components potentially requires examining $O(n^k)$ combinations of components. For example, in Lync, even if we limit localization to components that are actively involved in current transactions, the number of components can be around 600; naively considering 3 simultaneous failures as in $Joint_3$ can take days to run. The key characteristic is the maximum number s of simultaneous failures; the operator must feel that more than s simultaneous failures are extremely unlikely in practice.

5. Collective impact So far, we assumed that a *single* component failure affects an transaction in possibly uncertain fashion. However, many networks exhibit a more complex dependency between an transaction and a *set* of components; the transaction's success depends on the collective health of the components in the set. For instance, when two servers are in a failover arrangement, the transaction succeeds as long as any server is functional, and fails only when they both fail. Collective impact is not limited to failover and load-balancing servers. Routers or links on the primary and backup paths in an IP network also have collective impact on message delivery. Multi-level models such as PML use additional logical nodes to model collective impact, but other models such as DTL and PTL may work badly if the network has a number of components that exhibit collective impact.[1]

6. Scale The scale of the network impacts the speed of fault localization. Faster localization means faster recovery and increased availability. Scale can be captured using the total number of components in the network and/or the typical number of observations fed to the localization algorithm. For Lync, the two numbers are 8000 and 2500.

4.6 Analysis methodology

In this section, we study the relative merits of the choices made by various localization algorithms in the face of the six network characteristics listed above. We do this by combining first principles reasoning and simulations of three diverse, real networks. We first describe our simulation method and the networks we study; the next subsection presents our findings.

4.6.1 Simulation harness

In each simulation, we first select which system components fail. We then generate enough transactions (some of which fail due to the simulated failures) such that diagnosis is not limited by a lack of observations, as is true of large, busy net-

[1] Our notion of collective impact differs from so called "correlated failures" in the literature which refers to components likely to fail together such as two servers are connected to the same power source.

works [KYGCS07, MGS+09]. Finally, we feed these observations to the fault localization algorithm under consideration and obtain its output as a ranked list of likely failures. The set of failures is constant during each run.

Unless otherwise specified, the components to fail and the transaction endpoints are selected randomly. In practice, failures may not be random; we have verified that results are qualitatively similar for skewed failure distributions. In §4.8, we show that our findings agree with diagnosing real failures in Lync.

As is common, we quantify localization performance using *diagnostic rank* and *computation time*. Diagnostic rank is the rank of components that have actually failed.[2] This measure reflects the overhead of identifying and resolving real failures, assuming that operators investigate component failures in the order listed by the localization algorithm.

Our simulation harness takes as input any network, any failure model, and any combination of localization methods and produces results. We will make this harness public to aid the development of future localization algorithms.

4.6.2 Networks considered

To ensure that our findings are general, we study three real networks that are highly diverse in terms of their size, services offered, and network characteristics. The first network (Lync) supports interactive, peer-to-peer communication between users, the second (Exchange) uses a client-server communication model, and the third (Abilene) is an IP-based backbone. Each network has one or more challenging characteristics. For instance, Lync has significant noise and simultaneous failures while Exchange has significant uncertainty. While networks similar to Exchange and Abilene have been studied before, to our knowledge we are the first to study diagnosis in a network similar to Lync.

1. Lync Lync is an enterprise communication system. that supports several communication modes, including instant messages, voice, video and conferencing. We focus on the peer-to-peer communication aspects of Lync. The main components of a Lync

[2] In information retrieval terms, diagnostic rank includes the impact of both precision and recall. It will be high if components deemed more likely are not actual failures (poor precision) or if actual failures are deemed unlikely (poor recall).

network are shown in Figure 4.3. Internal users are registered with registrars and authenticated with AD (active directory). Audio calls connect via mediation servers, and out of the enterprise into a PSTN (public switched telephone network) using gateways. Edge servers handle external calls. Branch offices are connected by a WAN and the PSTN to the main sites.

The deployment of Lync that we study spans many offices worldwide of a large enterprise. It has over 8K components and serves 22K users. We have information on the network topology and locations of users. For a two-month period, we also have information on failures from the network's trouble ticket database and on transactions (observations) from its monitoring engine.

2. Exchange Exchange is a popular email system. Transactions in this network include sending and receiving email, and are based on client-server communication. Important components of an Exchange network deployment are shown in Figure 4.4 and include mail servers, DNS, and AD servers.

We study the Exchange deployment used in [CZMB08]. It has 530 users distributed across 5 regions. The network has 118 components. The number of hubs, mailboxes, DNS and AD servers in a region are proportional to the number of users. AD servers are in a load balancing cluster; hubs, DNS and mailbox servers are in a failover configuration.

3. Abilene Abilene is an IP-based backbone that connects many academic institutions in the USA. The topology [abi05] that we use has 12 routers and 15 links, for a total of 27 network components. The workload used for Abilene consists of paths between randomly selected ingress and egress routers selected.

4.7 Analysis results

Table 4.4: Effectiveness of diagnostic methods with respect to factors of interest. * depends on the network.

	Uncertainty	**Observation Noise**	**Covering relationship**	**Simultaneous failures**	**Collective Impact**	**Scale**
DTL	Good w/ Failure-Only. Poor w/ other scoring funcs.				Poor	Good
PTL	Good				Poor	OK
PML	Good				Good w/ Joint_k. Poor otherwise.	OK
FailureOnly (FO)	Good	Poor	Poor			Good
InBetween	Good w/ PTL, PML Poor with DTL	Ok	Good			OK
FailureSuccess (FS)	Good w/ PTL, PML. Poor with DTL	Good	Good			OK
Independent (Ind)		Good		Poor	Poor	Good

Continued on next page

Table 4.4 – *Continued from previous page*

	Uncertainty	Observation Noise	Covering relationship	Simultaneous failures	Collective Impact	Scale
Joint$_k$(Jt_k)		Good		Good (s$\leq k$). Poor (s$>k$)	Good (c$\leq k$). Poor (c$>k$)	Poor
Gsc		Poor		Good*	Poor	Good
Hierarchical		Poor		Poor	Poor	OK

Table 4.4 summarizes our analysis of the design space by qualitatively rating models, scoring functions and explorers based on how well they handle the six network characteristics from §4.5. For each network characteristic (columns), the Table rates each method as being good, OK, or poor. An empty (shaded) subcolumn for a characteristic implies that each row is qualitatively equivalent with respect to that characteristic. For instance, the choice of state space explorer has little impact on the ability to handle uncertainty. We have empirically verified such equivalence, but we omit these experiments from this paper and focus on parts of the table where different options behave differently. Each such finding highlights the relative merits of choices given a network characteristic[3], and we use it later to guide the design of Gestalt. We have verified each finding on all three networks but present experimental results from only one network since others are in agreement.

4.7.1 Uncertainty

Uncertainty arises when the impact of a component on an transaction is not certain — such as when a DNS server *may* impact a ping, depending on whether the name translation is cached. Probabilistic models (PTL, PML) naturally handle uncertainty;

[3]We a given network may have more than one of these characteristics, we can study each characteristic in isolation because we control the conditions in simulations.

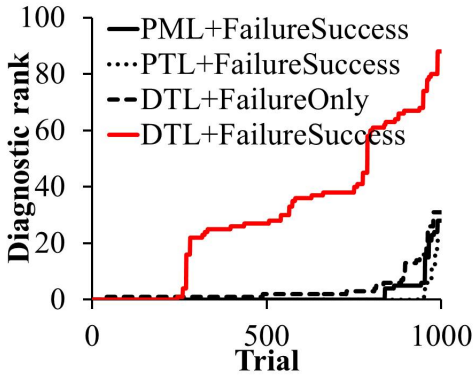


Figure 4.5: DTL can handle uncertainty when used with FailureOnly.

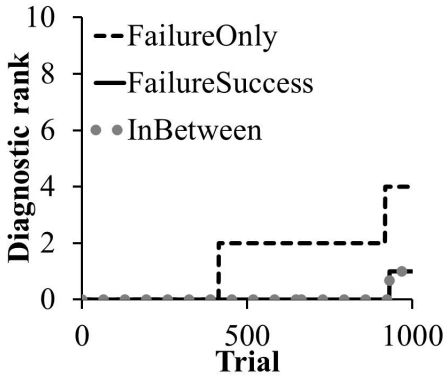


Figure 4.6: FailureOnly performs poorly for covering relationships.

thus researchers advise [KKV05, BCG+07] against using simpler deterministic models such as DTL. But we find, perhaps surprisingly, that despite being deterministic, DTL can handle uncertainty if it uses the right scoring function.

Finding 1 *In the presence of uncertainty, DTL suffices if the scoring function is FailureOnly.* Consider a component such as a DNS server whose impact on a specific transaction such as Ping 1 is uncertain. In DTL, this uncertainty must be resolved (since the model is binary) in favor of assuming impact; for instance, we must assume that Ping 1 depends on the DNS server even if Ping 1 used a locally cached DNS translation. (If we err in the opposite direction and assume that Pings do not depend on the DNS server, we would never be able to implicate the DNS server if the cache is empty and the DNS server fails.)

If this assumption happens to be true, no harm is done. But if false (i.e., the

transaction does not depend on the component), there are two concerns. First, consider the case when the the real failure was a different component; for example, Ping 1 failed because some router R in the path failed and not because the DNS server D failed. In that case, D may be considered a more likely cause of the failure of Ping 1 than R; but this can increase the diagnostic rank of R by at most 1, which is insignificant.

The second, more important, concern is that the ability to diagnose the failure of the falsely connected component itself may be significantly diminished. For example, when the DNS failure D fails, other Pings (say Ping 2 and Ping 3) may succeed because they use cached entries. This can confuse the fault localization algorithm because it increases the number of unexplained successes nS attributed to D, and decreases eS, potentially increasing significantly the diagnostic rank of D.

But since FailureOnly functions use only eF and nF in computing their score, they are not hindered by the false connection. On the other hand, FailureSuccess and InBetween are negatively impacted because they do use eS and nS.

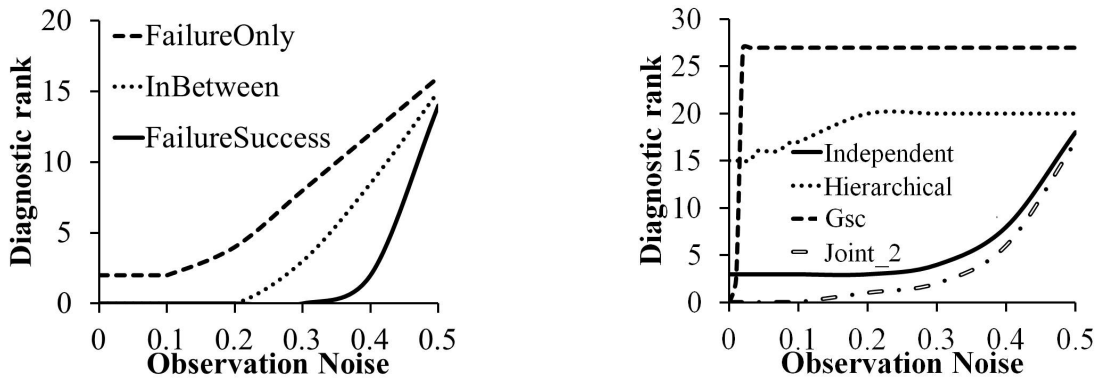
Figure 4.5 provides empirical confirmation for this finding using Exchange which has significant uncertainty because of the use of DNS servers whose results can be cached. It plots the diagnostic rank for 1000 trials; in each trial, a single random failure is injected. Observe that DTL with FailureOnly handles uncertainty just as well as PML and PTL. By contrast, DTL with FailureSuccess has much worse diagnostic rank (50 versus 5 in some trials). An implication of Finding 1 is that if the network has only uncertainty, it can be best handled (with small computation time and comparable diagnostic rank) using DTL and FailureOnly without using probabilistic models such as PTL.

4.7.2 Observation noise

Different scoring functions and state space explorers have different sensitivities to observation noise as we detail below.

Scoring functions

Finding 2 *FailureSuccess is most robust to observation noise, followed by InBetween, and then by FailureOnly.* To understand this finding, note that noise turns



(a) Scoring functions

(b) State space explorers

Figure 4.7: Sensitivity to observation noise.

successful transactions into apparent failures or vice versa. This perturbs the scoring function elements such that $eF'{=}eF{\pm}\Delta_{eF}$, and so on for nF, eS and nS. Because noise perturbs all basic elements, it impacts all scoring functions.

But the extent of perturbation differs because each scoring function combines these elements differently (Table 4.3). FailureOnly is the most impacted because it uses only failure elements (eF and nF). These elements can change significantly as noise turns successful transactions, which are more common, into apparent failures. FailureSuccess is the least impacted as it uses both failure and success information (eF+eS), which is perturbed less. InBetween falls between these extremes. In general, using more evidence and all available elements reduces sensitivity to noise.

For example, suppose in the ground truth before noise, there are 5 failed transactions and 100 successful transactions. Due to 5% noise, say 5 of the successful transactions are turned to failures and 1 of the failures is turned into a success. Now there are 5 incorrectly observed failures to add to 4 true failures. A component C that explained a single failure before noise could easily explain 3 failures (1 real plus 2 noise-induced) failures after noise. This could triple C's score if FailureOnly is used, incorrectly boosting C's diagnostic rank. On the other hand, suppose the same component explained 20 successes before noise and 22 after noise. Then measures like FailureSuccess will be less affected because they equally weight explained failures and successes; the (typically) larger number number of successes will be less sensitive to noise than the (typi-

cally) smaller number of failures.

Figure 4.7(a) confirms this behavior. We inject single failures in Abilene and introduce 0-50% noise. We run 100 trials for each noise level and plot the median diagnostic rank for each level. This graph uses DTL and Independent as the system model and state space explorer; the relative trends are similar with other combinations.

State space exploration

Finding 3 *Iterative state space explorers, Gsc and Hierarchical, are highly sensitive to noise.* This sensitivity stems directly from the iterative nature of these methods. An erroneous inference (due to noise) made in an early iteration can cause future inferences to falter. Independent and Joint$_k$, which are not iterative, do not have this shortcoming.

Figure 4.7(b) confirms this behavior. In this experiment, we introduced two independent failures in Abilene and 0-50% observation noise. The experiment uses DTL and FailureSuccess while varying the state space explorer; other combinations of model and scoring function produce similar trends. Figure 4.7(b) plots the median diagnostic rank across 100 trials. We see that Gsc and Hierarchical deteriorate with even small amounts of noise.

Finding 3 helps explain the extreme sensitivity of SCORE, which uses FailureOnly and Gsc, to noise, that prior work [KYGCS07] empirically observed but did not fully explain. The earlier paper [KYGCS07] tried to alleviate the impact of noise by running multiple instances of fault localization on different topologies (which has high overhead) while retaining FailureOnly and Gsc, methods inherently sensitive to noise.

4.7.3 Covering relationships

Recall that a component C covers a component D if the set of transactions that D impacts is a subset of the set of transactions that C impacts. In other words, when an transaction that D impacts fails, it is impossible to distinguish a failure of C from that of D by looking only at failures.

Finding 4 *For covering relationships, FailureOnly scoring functions should not be used.* Other scoring functions (FailureSuccess and InBetween) can better disambiguate

the failures of the covering and covered component because they use successful transactions (eS, nS) as well, and not only failed ones. For instance, consider a failed link. All failed transactions due to the link can also be explained by the failure of the attached routers. However, by using successful transactions that include the routers but not the failed link, the scoring function can assign a higher likelihood to link failure than router failure.

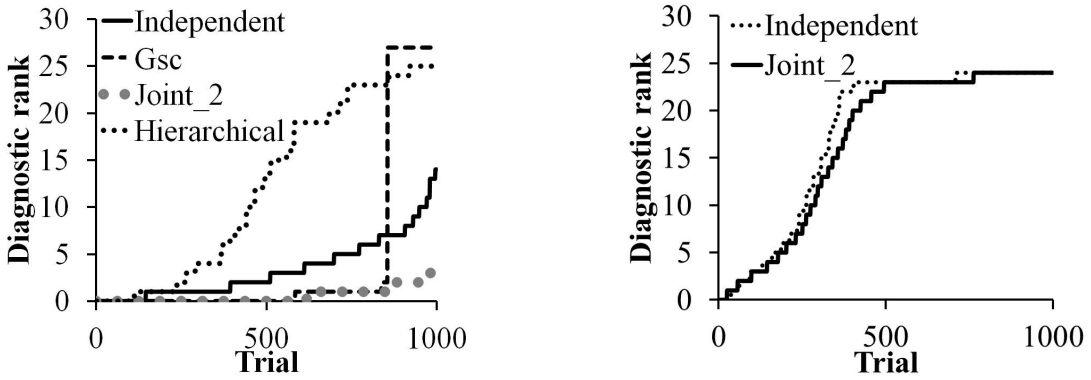
Figure 4.6 provides empirical evidence for Finding 4 by showing the results of an experiment using Abilene, which has many covering relationships. We randomly introduced a single failure in the network and diagnosed it using different scoring functions (combined with DTL and Independent). We see that FailureOnly has the worst performance with non-zero diagnostic rank in 60% of the trials while the other two methods have rank 0 most of the time.

We note that FailureOnly has been used by several tools to diagnose ISP backbones [KYGCS05,KYGCS07,DTDD07], which have many covering relationships. Finding 4 suggests that the localization accuracy of these tools can be improved by changing their scoring function.

4.7.4 Simultaneous failures

We now discuss simultaneous failures of components that have *independent* impact on transactions. The next section discusses *collective* impact.

Finding 5 *For a small number of simultaneous failures ($s \leq k$), Joint$_k$ is best and Hierarchical is worst.* The effectiveness of Joint$_k$ follows because it directly examines all system states with k or fewer failures. Hierarchical does poorly because its clustering approach forces it to explain more failures than needed. Suppose three transactions O_1, O_2, O_3 have failed and component C explains the first two failures and no other component explains more failures. Suppose, however, that C also impacts transaction O_4. Then Hierarchical will add C to the cluster but will also add transaction O_4 as a new failed transaction to be explained by subsequent iterations. Intuitively, the onus of explaining more failures than those observed can lead Hierarchical astray in later iterations. Independent is less susceptible because it evaluates each component independently.



(a) 2 simultaneous failures

(b) 3-4 simultaneous failures

Figure 4.8: Ability of state space explorers to handle simultaneous failures.

Gsc presents an interesting case study. As long as the failed components are diverse, it is more effective than Independent because it chooses the smallest set of failed components that explain the failures. However, Gsc can fail badly if multiple sets of component failures can explain the failed transactions.; this can happen, for instance, when the network has many covering relationships. (The analysis of covering relationships in §4.7.3 considered single failures, which Gsc can handle well.)

For example, consider a network with two routers $R1$ and $R2$ with a link L between them. If both $R1$ and $R2$ fail, Gsc will prefer the more parsimonious explanation that L failed. Worse, Gsc (unlike $Joint_2$) will never consider the joint failure of $R1$ and $R2$, making the diagnostic rank of the actual failure extremely high. On the other hand, if two other routers $R3$ and $R4$ fail simultaneously but do not have a link between them, Gsc will do very well.

Figure 4.8(a) shows the performance of different state space explorers when diagnosing two (randomly picked) simultaneous failures in Abilene. The graph uses PML and FailureSuccess; other combinations produce similar trends. We see that Joint$_k$ is highly effective (rank 2 or less), and Hierarchical is poor (rank > 20 in 25% of trials). Gsc has bimodal behavior with a rank > 25 in a small fraction of trials. Closer investigation confirms that these trials involve the simultaneous failures of two components who together cover a third component.

Finding 5 explains why Pinpoint [CKFF02], which uses Hierarchical, has poor

performance (see Figure 4 in [CKFF02]) for even two simultaneous failures, despite the handling of simultaneous failures being an explicit goal of Pinpoint. It suggests that replacing Hierarchical state space exploration in Pinpoint (with, say, $Joint_2$) while keeping the same system model and scoring function would improve Pinpoint's diagnosis of simultaneous failures.

More broadly, Table 4.4 shows that the performance of Hierarchical is similar or worse than Independent and Gsc in all cases. We thus recommend that future algorithms not consider this method.

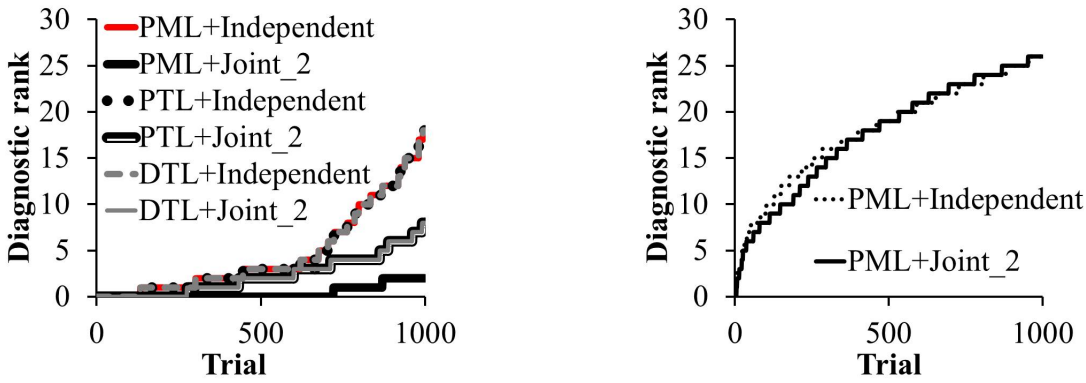
Finding 6 *$Joint_k$ handles simultaneous failures poorly in large networks.* First, $Joint_k$'s computation scales poorly with network size because considering every subset of k components among n components takes $O(n^k)$ time. As we demonstrate later (Figure 4.12(c)), running Joint$_3$ with $k = 3$ takes 21 minutes even when run on a small 67 component network. In practice, our Lync network has 8000 components but other considerations allow limiting the number of components to be considered in a failure to be around 600. Scaling to this size would require three orders of magnitudes more time (which is many days) to run $Joint_3$ in which case a manager may as well conduct manual localization.

Then, we also find that if the number of simultaneous failures s is greater than k, Joint$_k$ is in fact no better than Independent or another scoring function. That is, the high cost of Joint$_k$ is not worthwhile unless one can afford to use a $k \geq s$. Figure 4.8(b) shows an example experiment over Abilene, in which $k < s$. Joint$_2$ is no more effective than Independent.

Thus, while $Joint_k$ does better than Independent and GSC in handling noise and collective impact, it cannot handle simultaneous failures well in large networks.

4.7.5 Collective impact

We now study simultaneous failures of components that have a collective impact on transactions by being, for instance, in a load balancing or failover relationship. We find that in such cases, the choice of system model and state space explorer should be jointly made. We explore two cases: when the number s of failed components in a collection is small ($s \leq$k), and when it is large ($s > k$).



(a) 2 collective failures

(b) 3 collective failures

Figure 4.9: Ability of a model, state space explorer combination to handle failures with collective impact.

Finding 7 *For diagnosing a small number of simultaneous failures in a collection ($s \leq k$), combining PML and Joint$_k$ is most effective; any other system model or state space explorer leads to poor diagnosis.* This is because, among existing models, only PML can encode collective impact relationships. Other models represent approximations that can be far from reality. However, picking the right model is not enough. The state explorer must also consider simultaneous failure of these components. Among existing state space explorers, only Joint$_k$ has this property. Independent does not consider simultaneous failures, and Gsc and Hierarchical assume that components have independent impact.

Figure 4.9(a) demonstrates this behavior. We modeled failures among components with collective impact in Abilene as follows. Each trial randomly selects a pair of nodes that has two vertex-disjoint disjoint paths between them. For messages between these nodes, the two paths can be considered to be in a failover relationship with collective impact. We then introduced a randomly selected failure along each path. Thus, all messages sent between the pair of nodes will now fail. For 1000 such trials, the graph plots the diagnostic ranks of several combinations of system model and state space explorer. It uses the FailureSuccess scoring function, but other functions yield similar results. We omit results for Gsc and Hierarchical; they had worse performance than Independent. As we can see, only PML+Joint$_2$ is effective.

This result implies that half-way measures are insufficient for diagnosing collective impact failures. We must both model relationships (PML) and explore joint failures (Joint$_k$). Localization suffers severely if either choice is wrong. For example, Shrink [KKV05] uses PTL with Joint$_k$ even though it targets IP networks which may have potentially many failover paths. Finding 7 suggests that Shrink would do better to replace PTL with PML.

Large number of failures

Finding 8 *For s simultaneous failures with collective impact, PML+Joint$_k$, $k < s$ provides no advantage.* Finding 7 may seem to imply that $Joint_2$ suffices for failures with collective impact. However, intuitively Joint$_2$ works well in Figure 4.9(a) because there are only two simultaneous failures with collective impact. How well does Joint$_2$ do when there are 3 simulataneous failures with collective impact?

Figure 4.9(b) answers this question. We articially introduced a few additional links in the Abilene topology to allow three (one primary plus two backups) disjoint backup paths for some source-destination pairs. We then failed a (randomly selected) component along each of the three paths and diagnosed the failure by combining PML with Independent and Joint$_2$. As we can see, PML+Joint$_2$ is as poor at diagnosing these failures as PML+Independent.

As with independent, simultaneous failures this result implies that with current methods there are no half-way measures in diagnosing simultaneous failures with collective impact. To be able to diagnose s failures, we must either use Joint$_s$ or some other lower overhead method that considers combinations of k faults.

4.8 Gestalt

The insights from the analysis above led us to develop Gestalt. It combines ideas from existing algorithms and also includes a new state space exploration method.

For the system model, Gestalt uses a hybrid between DTL and PML that combines the simplicity of DTL (fixed number of levels, deterministic edges) with the expressiveness of PML (ability to capture complex component relationships). Our model

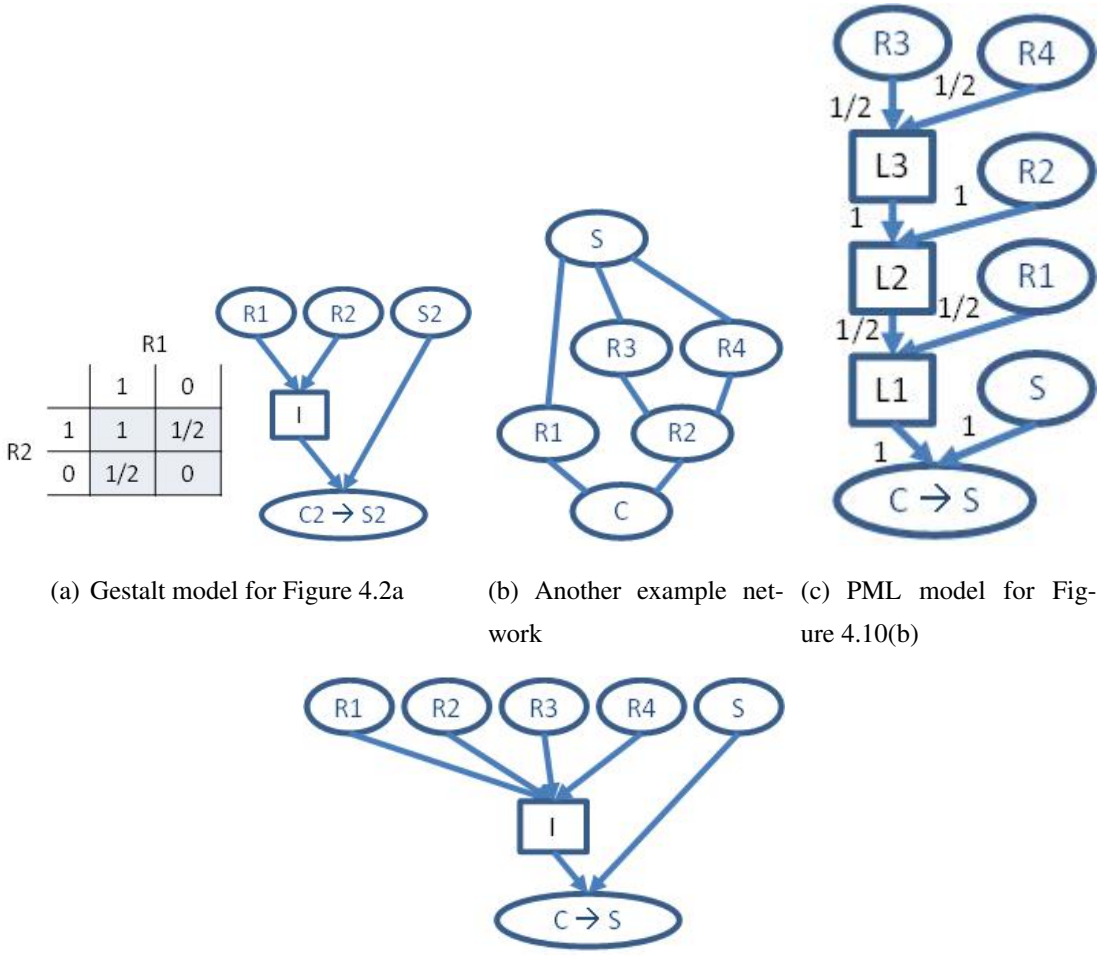
has three levels, where the top level corresponds to system components that can fail independently and the bottom level to transactions. An intermediate level captures collective impact of system components. Instead of encoding probabilistic impact on the edges, the intermediate node encodes the *function* that captures the nature of the collective impact. The domain of this function is the combinations of states of the parent nodes, and the range is the impact of each combination on the transaction. Figure 4.10(a) shows how Gestalt models the example in Figure 4.2a. The intermediate node I encodes the collective impact of R_1 and R_2. The function represented by I is shown in the figure, which shows values only for p_{up} (p_{down}=1–p_{up}).

While for this example, PML too has only three levels, Figure 4.10(b) illustrates the difference between PML and Gestalt. Here, to reach S, C spreads packets across $R1$ and $R2$, and $R2$ spreads across $R3$ and $R4$. Figures 4.10(c) and 4.10(d) show PML and Gestalt models for this network.

Another difference between PML and our model is how we capture single components with uncertain impact on a transaction (e.g., a DNS server whose responses may be cached). Gestalt models these with 3 levels too. An intermediate node captures the uncertainty from the component's state to its impact on the transaction. It may deem, for instance, that the transaction will succeed with some probability even if the component fails.

As scoring function, we use FailureSuccess because of its robustness to noise and covering relationships (Findings 2 and 4). Further, because we explicitly models uncertainty (unlike DTL), the combination of our model and FailureSucess will be robust to uncertainty as well (Finding 1).

For state space exploration, we develop a method that has the localization accuracy of Joint$_k$ and the low computational overhead of Gsc. It is based on the following observations. Gsc is susceptible to covering relationships because many failure combinations can explain the observations and Gsc explores only a subset, ignoring others (Finding 5). Gsc is susceptible to noise because noise can make it pick a poor candidate and rule out other possibilities (Finding 3). The diagnostic accuracy of Joint$_k$ for collective impact failures stems from the fact that it explore combinations of at most k failures; exploring a smaller number does not help (Finding 7, 8). But because its exploration is



(a) Gestalt model for Figure 4.2a

(b) Another example network

(c) PML model for Figure 4.10(b)

(d) Gestalt model for Figure 4.10(b)

Figure 4.10: Gestalt Model

fully combinatorial, it has a high computational overhead.

Our new exploration method is shown in Algorithm 4.1. It takes two parameters as input. The first is $Noise_{thresh}$, the percentage of observation noise expected in the network, which can be estimated from historical data. Given ground truth (post resolution) about a failure and the transaction logs, the percentage of transactions that cannot be explained by the ground truth reflects the level of observation noise. In Lync, we found this to be around 10%. The second parameter is k, the maximum number of simultaneous failures expected in the network. It can also be gleaned from historical failure data.

The candidate failures that we explore are single component failures and combinations of up to k components with collective impact. This candidate pool explicitly accounts for collective impact failures (making them diagnosable, unlike in Gsc). It is also much smaller than the pool considered by Joint$_k$ which includes all possible combinations of up to k failures. The output of the exploration is a ranked list of hypotheses, where each hypothesis is a set of at most k candidates from the pool.

These sets are computed separately for different thresholds of hit ratio [KYGCS05]. The hit ratio of a candidate is the ratio of number of failed versus total transactions in which the component(s) participated. Iterating over candidates in decreasing order of hit ratios gives us a systematic way of exploring failures while focusing on more likely failures first because actual failures are likely to have larger hit ratios. Hit ratios are not used in the scoring function.

For a given hit ratio threshold, the hypothesis sets are built iteratively (i.e., not all possible sets are considered) in k steps. We start with the empty set. At each step, each set is forked into a number of child sets, where each child set has one additional candidate than the parent set.

The child candidates are computed as follows. Let O_{unexp} be the set of observations whose status cannot be explained by the parent set (i.e., the status does not match what would be predicted by the system model). Initially, when the parent set is empty, this set equals O_{all}, the set of all observations. Then, we first compute the score of each candidate in the entire pool with hit ratio higher than the current threshold. This computation uses the scoring function (FailureSuccess) and is done with respect to O_{unexp}. Candidates more likely to explain the as yet unexplained observations will have higher scores.

If there were no observation noise, candidates with the maximum score can be used as child candidates because they best explain the remaining unexplained observations. But it is not robust to noise. Due to noisy observations, the score of actual failures may go down and the score of some other candidates may go up. By focusing only on candidates with the maximum score, we run the risk of excluding actual failures from the set. In fact, this is a key reason why Gsc is not robust to noise.

We thus cast a wider net, with the width of the net proportional to expected noise.

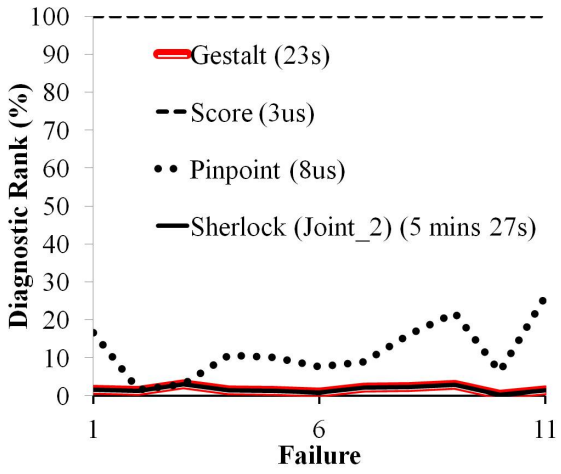


Figure 4.11: Comparison of diagnostic efficacy of different algorithms for real failures in a Lync deployment.

The quantity by which the score of the actual culprit can reduce due to observation noise $score_{noise} = Noise_{thresh} \times |O_{unexp}|$. The selected child candidates are those with scores higher than $score_{max} - score_{noise}$, where $score_{max}$ is the maximum score across all candidates. This guarantees that we will not miss actual failures in our iterations. We will, however, pick more candidates, but the eventual cost of that is significantly lower.

4.9 Gestalt Evaluation

We now evaluate Gestalt and compare it to three existing algorithms that use very different techniques. We start with the Lync network and use the algorithms to diagnose real failures using real transactions available in the system logs. Based on information from days prior to the failures we diagnose, we set $Noise_{thresh}$=10% and k=2 for Gestalt.

Figure 4.11 shows the results for a number of failures seen in a two month period (the actual failure count is hidden for confidentiality). The legend shows the median running time for the algorithms on a 3 GHz dual-core PC. We see that SCORE and Pinpoint perform poorly. Gestalt and Sherlock perform similarly, but the running time of Gestalt is lower by more than an order of magnitude. This is despite the fact that we ran Sherlock with Joint_2. Using Joint_3, which was the recommendation in the original

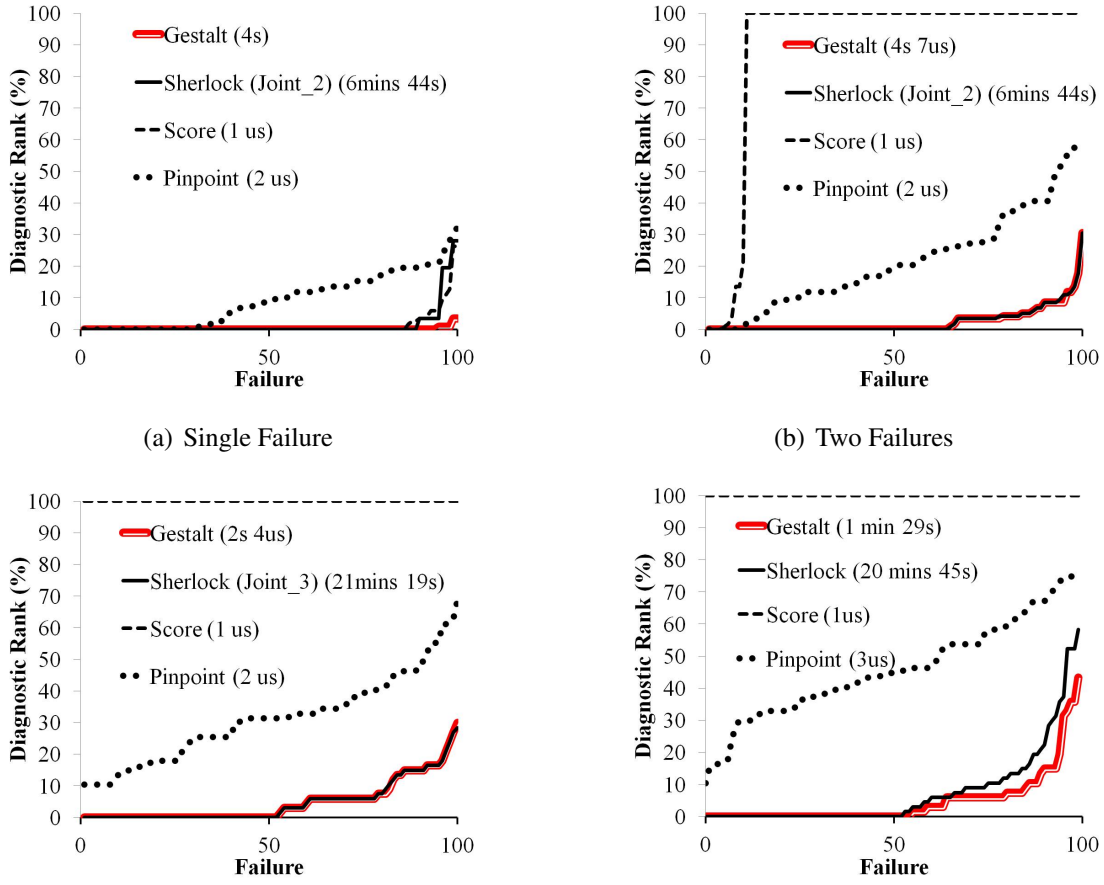
Sherlock paper [BCG+07], would have taken around 20 hours per failure.

Figure 4.5 provides more details for ten sample failures in the logs. We see that the time it took for the operators to manually diagnose these failures (original recovery delay) was very high. The median time was around 8 hours, though it took more than a day for two failures. The primary reason for slow manual diagnosis time is the large number of network components that must be manually inspected. The table lists the number of components involved in failing transactions as an estimate of the number of possible components that might need to be checked. Of course, using domain knowledge and expertise, an operator will only check a subset of these components; but the estimate underscores the challenge faced by operators today. We see that using Gestalt, the operator will have to check only 3-13 components before identifying the real culprits compared to 196-655 components for manual diganosis, significantly reducing diagnosis time. Note that the run time for Gestalt to whittle down the list of suspects by 1-2 orders of magnitude is at most a few minutes.

We next consider failures in the Exchange network. Figures 4.12(a) and 4.12(b) show results for diagnosing one and two component simulated failures. We again used Joint_2 for Sherlock and k=2 and $Noise_{thresh}$=0 for Gestalt. As expected based on our earlier analysis, Score does very well for single failure scenarios, but suffers in two-failure scenarios due to covering relationships. Sherlock and Gestalt do well for both cases, but Sherlock takes two orders of magnitude more time.

In order to experiment with more simultaneous failures and Joint_3, we reduced the size of the Exchange network by half (to 67 components). Figures 4.12(c) and 4.12(d) show the results for three failures and for four failures with 1% observation noise. In the latter case, we run Gestalt with $Noise_{thresh}$=1%. We see that Gestalt matches Sherlock's diagnostic accuracy for three failures, with running time that is two orders of magnitude faster. For four failures, Gestalt has better diagnostic accuracy than Sherlock because it accounts for noise. Its running time is still better by 20x, even though noise makes it explore more combinations of component failures.

Due to space constraints, we omit results for the Abilene, but we found those results to be qualitatively similar to those above. Gestalt had better diagnostic efficacy than SCORE and Pinpoint for all cases. Gestalt matched Sherlock's accuracy for most



(a) Single Failure

(b) Two Failures

(c) Three Failures

(d) Four Failures, 1% Noise

Figure 4.12: Diagnostic efficacy of different algorithms with Exchange network with different number of failures.

cases and exceeded it in the presence of noise and more than three simultaneous failures. Its running time was 1-2 orders of magnitude lower than Sherlock.

4.10 Summary and Future Work

In this chapter, we presented a framework that helps understand the design space of practical fault localization algorithms. Using this framework, we analyzed the effectiveness of different algorithms at handling six characteristics of large, complex networks that pose a challenge to fault localization. We also found that no existing algo-

rithm simultaneously provides high diagnostic accuracy and low computational cost for a range of networks.

Based on the insights from our analysis, we designed Gestalt, a new fault localization algorithm that borrows ideas from existing algorithms but also includes a new state space exploration method. This method represents a continuum between greedy, low-accuracy exploration and combinatorial, high-overhead exploration. For three very different networks (messaging, email, ISP), Gestalt has higher diagnostic accuracy or lower overhead than existing algorithms.

We believe even better performance can be obtained by exploiting more refined fault models; for example, there should be locality among simultaneous failures in a global network. But beyond the specific algorithm, we hope Gestalt takes a modest step towards understanding the *gestalt* of fault localization.

4.11 Acknowledgment

Chapter 4, in part, contains material submitted for publication as ”Gestalt: Unifying fault localization for networked systems” Niranjan Mysore, Radhika; Mahajan, Ratul; Vahdat, Amin; Varghese, George. The dissertation author was the primary investigator and author of this paper.

Table 4.3: Different fault localization algorithms mapped to our framework.

Tool	**Target system**	**System Model**	**Scoring Function**	**State Space Exploration**	**Aspects not captured**
Codebook [KYY+95]	Satellite comm. network	DTL, PTL	FailureSuccess ($\Sigma(eF + eS)$)	Independent	Codebook selection
MaxCoverage [KYGCS07]	ISP backbone	DTL	FailureOnly ($\frac{\Sigma eF}{TF}$)	Gsc	Candidate post-selection, Hypothesis selection
NetDiagnoser [DTDD07]	Intra-AS, multi-AS internetwork	DTL	FailureOnly ($\frac{\Sigma eF}{TF}$)	Gsc	Candidate pre-selection
NetMedic [KMV+09]	Small enterprise network	PTL	FailureOnly (ΣeF)	Independent	Re-ranking
Pinpoint [CKFF02]	Internet services	DTL	InBetween ($\frac{\Sigma eF}{TF+\Sigma nS}$)	Hierarchical	
SCORE [KYGCS05]	ISP backbone	DTL	FailureOnly ($\frac{\Sigma eF}{TF}$)	Gsc	Threshold based hypothesis selection
Sherlock [BCG+07]	Large enterprise network	PML	FailureSuccess ($\prod(eF + eS)$)	Joint_3	Statistical significance test
Shrink [KKV05]	IP network	PTL	FailureSuccess ($\prod(eF + eS)$)	Joint_3	
WebProfiler [ALMP10]	Web applications	DTL	InBetween ($\frac{\Sigma eF}{\Sigma nS+\Sigma eF}$)	Joint_2	Re-ranking

Algorithm 4.1: Pseudocode for Gestalt

1: $\overline{H}_{all}$ = {}
2: **For each** $hitRatio$ **in** $1, 0.95, \cdots 0$ **do**
3: H_{curr} = (); //current hypothesis
4: $\overline{O}_{unexp}$ = $\overline{O}_{all}$; //unexplained observations
5: $\overline{H}_{all}$ += `GenHyp` (*1,* $\overline{O}_{unexp}$*,* $hitRatio$*,* H_{curr})
6: **Return** $\overline{H}_{all}$

`GenHyp` (i, $\overline{O}_{unexp}$*,* $hitRatio$*,* H_{curr})

1: $\overline{H}_{return}$ = {H_{curr} }
2: $\overline{C}_{new}$ = `NewCandidates` ($hitRatio$*,* $\overline{O}_{unexp}$)
3: **For each** c **in** $\overline{C}_{new}$
4: hyp_{new} = (hyp, c)
5: **If** (i $==$ k)
6: $\overline{H}_{return}$ += hyp_{new}
7: **Else**
8: $\overline{O}_{exp}$ = `ExpObs` (hyp_{new},$\overline{O}_{unexp}$)
9: $\overline{H}_{return}$ += `GenHyp` (*i+1,* $\overline{O}_{unexp} - \overline{O}_{exp}$*,* $hitRatio$*,* hyp_{new})
10: **Return** $\overline{H}_{return}$

`NewCandidates` ($hitRatio$*,* $\overline{O}_{unexp}$)

1: $\overline{C}_{new}$ = {}
2: **For each** c **in** CandidatePool
3: **If** (`HitRatio` (c) $\geq hitRatio$)
4: $\overline{C}_{new}$ += c
5: $score_{max}$ = `MaxScore` ($\overline{C}_{new}$, $\overline{O}_{unexp}$)
6: $score_{noise}$ = $Noise_{thresh} \times$ —$\overline{O}_{unexp}$ —
7: **For each** c **in** $\overline{C}_{new}$
8: **If** (`Score` (c) $\geq score_{max} - score_{noise}$)
9: $\overline{C}_{new}$ −= c
10: **Return** $\overline{C}_{new}$

Table 4.5: Statistics for a sample of real failures in Lync.

	Original recovery delay (days, hh:mm)	# potential failed comps.	Gestalt diagnostic rank	Gestalt run time (mm:ss)
1	0, 01:50	196	11	4:02
2	0, 00:50	625	7	2:59
3	0, 01:55	552	6	0:05
4	0, 22:05	608	9	0:05
5	1, 23:45	521	7	0:12
6	0, 10:55	655	6	0:21
7	14, 06:25	676	12	2:43
8	0, 01:45	571	13	1:06
9	0, 20:15	562	13	0:23
10	0, 08:20	455	3	1:03

Chapter 5

Conclusion

In conclusion, we propose new automated management techniques tailored for data center requirements namely: PortLand, FasTrak and Gestalt. In this chapter we first summarize our contributions. This dissertation ends with a discussion of open issues and future work.

5.1 Summary of contributions

The goal of this dissertation is to build a management system that is tailored to meet specific data center requirements. Most of the challenges we encountered through this work trivially arose from challenges of scale.

The first challenge we tackled was that plug-and-play fabrics though desirable for data center networks do not scale. To address this challenge we built PortLand, a self-configuring network fabric. PortLand leverages underlying structure of data center networks, the key insight being that data centers are usually built as multi-rooted trees. Using this, PortLand switches determine their location in the network with some help from the Portland fabric manager. This enables PortLand switches to give out location based MAC addresses to all VMs in the network. PortLand also leverages network structure to re-define how broadcast works in the system. Flooding and loops are completely eliminated in PortLand; as such frequently encountered network problems such as broadcast storms, spanning tree misconfigurations, blackholes are completely avoided with PortLand. As a layer 2 network, PortLand can support workload mobility.

By adopting hierarchical addressing and forwarding from Layer 3 networks, PortLand traffic can take advantage of path diversity within data center fabrics.

Second, we examined the trade-off between scale and performance, under practical hardware constraints. Today, multi-tenant networks give up network performance by introducing slow software queues at source and destination while ensuring tenant isolation. With FasTrak we show that this trade-off is not fundamental. FasTrak moves isolation rules between hypervisors and switch hardware. The application traffic that is migrated to hardware sees near bare-metal performance. Such offloading comes with additional benefits; Due to the offload, the hypervisors are less loaded, and average latency perceived by traffic that is not offloaded also reduces. More importantly, CPU cycles used for network processing are freed up, allowing more multiplexing. FasTrak shows that multi-tenant data centers can both support large scale and performance sensitive applications.

Finally, we examined the trade-off between exhaustive state space exploration for accurate diagnosis and localization time. With Gestalt, we go beyond just solving for this challenge. We first focus on distilling the connection between network properties and localization design choices, which gives us a framework that can be used to compare existing algorithms. This framework then helps build Gestalt, a fault localization algorithm that adapts to monitoring noise, dependency uncertainty. Gestalt is also significantly faster than existing algorithms that are known to be most accurate, and more accurate than existing algorithms that are known to be fast. With Gestalt we hope that the time-to-recovery reduces, allowing for higher availability. Gestalt is not restricted to diagnosis performance or control problems. It can be applied at any level of granularity, from lower level system failures, to application failures. We have used Gestalt on a corporate deployment of Lync at Microsoft.

We have prototyped, evaluated Portland, FasTrak and Gestalt extensively, and the results verify our claims.

5.2 Concluding remarks

Data center networks are revolutionizing the network world in many ways. The fact that the entire network stack, from hardware to the software stack on servers is under the control of one operator, has led to new ways in designing networks. Network management and control is being revisited at multiple levels. Undoubtedly, data centers enable very powerful applications; search, social networks, ability to anayze huge amounts of data, providing users with uniform experience with mobility. From the networking perspective, a lot can be done to make data center networks faster, more efficient, less expensive to enable more such applications at lower costs. We are at the beginning of this revolution.

This dissertation explores three specific aspects of management and control of networks: the network fabric, policy enforcement in multi-tenant environments and fault localization. For future work we aim to combine insights from PortLand and Gestalt to understand how network fabrics can be made more diagnosable. FasTrak makes a first step toward utilizing hardware partially for isolating a subset of network flows in the data center. Ideas in this work show promise in exploring hybrid hardware-software techniques to achieve network performance isolation for data center workloads, an area of active research.

Bibliography

[AA06] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. In *ASPLOS*, 2006.

[abi05] Abilene Topology. http://totem.run.montefiore.ulg.ac.be/files/examples/abilene/abilene.xml, 2005.

[acl] Access Control List. http://en.wikipedia.org/wiki/Access_control_list.

[AFLV08] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A Scalable, Commodity Data Center Network Architecture. In *SIGCOMM '08: Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, pages 63–74, New York, NY, USA, 2008. ACM.

[Agg01] Charu C. Aggarwal. Re-designing distance functions and distance-based applications for high dimensional data. In *SIGMOD Record*, 2001.

[ALMP10] Sharad Agarwal, Nokitas Liogkas, Prashanth Mohan, and Venkata N. Padmanabhan. Webprofiler: Cooperative diagnosis of web failures. In *COMSNET*, 2010.

[amaa] Amazon Elastic Compute Cloud (EC2) Pricing. http://aws.amazon.com/pricing/ec2/.

[amab] Amazon Virtual Private Cloud. http://aws.amazon.com/vpc/.

[AMW+03] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Weiner, Patrick Reynolds, and Athicha Muthitacharoen. Performance debugging for distributed systems of black boxes. In *SOSP*, 2003.

[ari] Arista 7150 Series Ultra-Low Latency Switch. http://www.aristanetworks.com/en/products/7150-series/7150-datasheet.

[azu] Microsoft Azure. http://www.windowsazure.com/.

[BCG+07] Paramvir Bahl, Ranveer Chandra, Albert Greenberg, Srikanth Kandula, Dave Maltz, and Ming Zhang. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *SIGCOMM*, 2007.

[BGRS99] Kevin Beyer, Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. When is "nearest neighbor" meaningful? In *ICDT*, 1999.

[bon] Linux Ethernet Bonding Driver. http://www.kernel.org/doc/Documentation/networking/bonding.txt.

[CCF+05] Matthew Caesar, Donald Caldwell, Nick Feamster, Jennifer Rexford, Aman Shaikh, and Jacobus van der Merwe. Design and Implementation of a Routing Control Platform. In *USENIX Symposium on Networked Systems Design & Implementation*, 2005.

[CCK+06] Matthew Caesar, Tyson Condie, Jayanthkumar Kannan, Karthik Lakshminarayanan, Ion Stoica, and Scott Shenker. ROFL: Routing on Flat Labels. In *Proceedings of ACM SIGCOMM*, 2006.

[CCN+06] Matthew Caesar, Miguel Castro, Edmund B. Nightingale, Gerg O, and Antony Rowstron. Virtual Ring Routing: Network Routing Inspired by DHTs. In *Proceedings of ACM SIGCOMM*, 2006.

[CFH+05] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen Eric Jul Christian Limpach, Ian Pratt, and Andrew Warfield. Live Migration of Virtual Machines. In *USENIX Symposium on Networked Systems Design & Implementation*, 2005.

[CFP+07] Martin Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: taking control of the enterprise. In *SIGCOMM*, 2007.

[cisa] Cisco Data Center Infrastructure 2.5 Design Guide. www.cisco.com/application/pdf/en/us/guest/netsol/ns107/c649/ccmigration_09186a008073377d.pdf.

[cisb] Configuring IP Unicast Layer 3 Switching on Supervisor Engine 2. www.cisco.com/en/US/docs/routers/7600/ios/12.1E/configuration/guide/cef.html.

[cisc] Configuring Virtual Routing and Forwarding. http://www.cisco.com/en/US/docs/net_mgmt/ciscoworks_lan_management_solution/4.0/user/guide/configuration_management/vrf.html.

[cisd] OSPF Design Guide. www.ciscosystems.com/en/US/tech/tk365/technologies_white_paper09186a0080094e9e.shtml.

[CKFF02] Mike Chen, Emre Kiciman, Eugene Fratkin, and Armando Fox. Pinpoint: Problem determination in large, dynamic, internet services. In *IPDS*, 2002.

[CKR08] Matthew Caesar Changhoon Kim and Jennifer Rexford. Floodless in SEATTLE: A Scalable Ethernet Architecture for Large Enterprises. In *SIGCOMM '08: Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, 2008.

[CKRS10] Martín Casado, Teemu Koponen, Rajiv Ramanathan, and Scott Shenker. Virtualizing the network forwarding plane. In *PRESTO*, 2010.

[CMT+11] Andrew R. Curtis, Jeffrey C. Mogul, Jean Tourrilhes, Praveen Yalagandula, Puneet Sharma, and Sujata Banerjee. Devoflow: scaling flow management for high-performance networks. In *SIGCOMM*, 2011.

[CTFD09] Italo Cunha, Renata Teixeira, Nick Feamster, and Christophe Diot. Measurement methods for fast and accurate blackhole identication with binary tomography. In *IMC*, 2009.

[CZMB08] Xu Chen, Ming Zhang, Morley Mao, and Paramvirn Bahl. Automating network application dependency discovery:experiences, limitations, and new solutions. In *OSDI*, 2008.

[DG04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[DTDD07] Amogh Dhamdherey, Renata Teixeira, Constantine Dovrolis, and Christophe Diot. Netdiagnoser: Troubleshooting network unreachabilities using end-to-end probes and routing data. In *CoNEXT*, 2007.

[DYL+10] Yaozu Dong, Xiaowei Yang, Xiaoyong Li, Jianhui Li, Kun Tian, and Haibing Guan. High performance network virtualization with sr-iov. In *High Performance Computer Architecture (HPCA)*, 2010.

[DYR08] Yaozu Dong, Zhao Yu, and Greg Rose. Sr-iov networking in xen: architecture, design and implementation. In *WIOV*, 2008.

[ec2] Amazon EC2. http://aws.amazon.com/ec2/.

[ecu12] Sizing Amazon Server Instances. http://ec2dream.blogspot.com/2012/04/sizing-amazon-server-instances.html, 2012.

[emu] Taking NVGRE to the Next Level. http://o-www.emulex.com/blogs/labs/2012/06/13/nvgre-next-level/.

[fac09] Personal communication with Donn Lee, Facebook, 2009.

[flo] Floodlight. http://floodlight.openflowhub.org/.

[GAH+12] Abel Gordon, Nadav Amit, Nadav Har'El, Muli Ben-Yehuda, Alex Landau, Assaf Schuster, and Dan Tsafrir. Eli: bare-metal performance for i/o virtualization. In *ASPLOS*, 2012.

[gce] Google Compute Engine. https://cloud.google.com/products/compute-engine.

[GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. *ACM SIGOPS Operating Systems Review*, 37(5), 2003.

[GLM+08] Albert Greenberg, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. Towards a Next Generation Data Center Architecture: Scalability and Commoditization. In *PRESTO '08: Proceedings of the ACM Workshop on Programmable Routers for Extensible Services of Tomorrow*, pages 57–62, New York, NY, USA, 2008. ACM.

[gre] GRE. http://tools.ietf.org/html/rfc1701.

[GWT+08] Chuanxiong Guo, Haitao Wu, Kun Tan, Lei Shi, Yongguang Zhang, and Songwu Lu. DCell: A Scalable and Fault-tolerant Network Structure for Data Centers. In *Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, pages 75–86, New York, NY, USA, 2008. ACM.

[Hec89] David Heckerman. A tractable inference algorithm for diagnosing multiple diseases. In *Uncertainty in Artificial Intelligence*, 1989.

[Hop00] C. Hopps. Analysis of an Equal-Cost Multi-Path Algorithm. RFC 2992, Internet Engineering Task Force, 2000.

[hyp] Hyper-V Network Virtualization Overview. http://technet.microsoft.com/en-us/library/jj134230.aspx.

[iom] IOMMU. http://en.wikipedia.org/wiki/IOMMU.

[ioz] IOZone Filesystem Benchmark. http://www.iozone.org.

[KBMJ+08] Ethan Katz-Bassett, Harsha V. Madhyashta, John P. John, Arvind Krishnamurthy, David Wetherall, and Thomas Anderson. Studying black holes in the internet with hubble. In *NSDI*, 2008.

[KCG+10] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, and Scott Shenker. Onix: a distributed control platform for large-scale production networks. In *OSDI*, OSDI'10, 2010.

[KKV05] Srikanth Kandula, Dina Katabi, and Jean-Philippe Vasseuri. Shrink: A tool for failure diagnosis in ip networks. In *MineNet workshop*, 2005.

[KMV+09] Srikanth Kandula, Ratul Mahajan, Patrick Verkaik, Sharad Agarwal, Jitendra Padhye, and Victor Bahl. Detailed diagnosis in computer networks. In *SIGCOMM*, 2009.

[KYGCS05] Ramana Rao Kompella, Jennifer Yates, Albert Greenberg, and Alex C. Snoeren. IP fault localization via risk modeling. In *NSDI*, 2005.

[KYGCS07] Ramana Rao Kompella, Jennifer Yates, Albert Greenberg, and Alex C. Snoeren. Detection and localization of network blackholes. In *Infocom*, 2007.

[KYY+95] S Klinger, S Yemini, Y Yemini, D Ohsie, and S Stolfo. A coding approach to event correlation. In *International Symposium on Integrated Network Management*, 1995.

[LCD04] Anukool Lakhina, Mark Crovella, and Christophe Diot. Diagnosing network-wide traffic anomalies. In *SIGCOMM*, 2004.

[LCR+07] Karthik Lakshminarayanan, Matthew Caesar, Murali Rangan, Tom Anderson, Scott Shenker, Ion Stoica, and Haiyun Luo. Achieving Convergence-Free Routing Using Failure-Carrying Packets. In *Proceedings of ACM SIGCOMM*, 2007.

[Lei85] Charles E. Leiserson. Fat-Trees: Universal Networks for Hardware-Efficient Supercomputing. *IEEE Transactions on Computers*, 34(10):892–901, 1985.

[Liu10] Jiuxing Liu. Evaluating standard-based self-virtualizing devices: A performance study on 10 gbe nics with sr-iov support. In *Parallel Distributed Processing*, 2010.

[LMW+07] John W. Lockwood, Nick McKeown, Greg Watson, Glen Gibb, Paul Hartke, Jad Naous, Ramanan Raghuraman, and Jianying Luo. NetFPGA–An Open Platform for Gigabit-Rate Network Switching and Routing. In *Proceedings of the 2007 IEEE International Conference on Microelectronic Systems Education*, pages 160–161, Washington, DC, USA, 2007. IEEE Computer Society.

[lyn12] Microsoft Lync. http://en.wikipedia.org/wiki/Microsoft_Lync, 2012.

[meg] Inside Microsoft's $550 Million Mega Data Centers. www.informationweek.com/news/hardware/data_centers/showArticle.jhtml?articleID=208403723.

[mem] Memcached. http://memcached.org/.

[MGS+09] Ajay Mahimkar, Zihui Ge, Aman Shaikh, Jennifer Yates, Yin Zhang, and Qi Zhao. Towards automated performance diagnosis in a large iptv network. In *SIGCOMM*, 2009.

[mid] Midokura. http://www.midokura.com/.

[MN06] R. Moskowitz and P. Nikander. Host Identity Protocol (HIP) Architecture. RFC 4423 (Proposed Standard), 2006.

[MNZ04] Andy Myers, T. S. Eugene Ng, and Hui Zhang. Rethinking the Service Model: Scaling Ethernet to a Million Nodes. In *ACM HotNets-III*, 2004.

[Moy98] J. Moy. OSPF Version 2. RFC 2328, Internet Engineering Task Force, 1998.

[MSWA03] Ratul Mahajan, Neil Spring, David Wetherall, and Thomas Anderson. User-level internet path diagnosis. In *SOSP*, 2003.

[MWJ99] Kevin P. Murphy, Yair Weiss, and Michael I. Jordan. Loopy belief-propagation for approximate inference: An empirical study. In *Uncertainty in Artificial Intelligence*, 1999.

[MYM+11] Jayaram Mudigonda, Praveen Yalagandula, Jeff Mogul, Bryan Stiekes, and Yanick Pouffary. Netlord: a scalable multi-tenant network architecture for virtualized datacenters. In *SIGCOMM*, 2011.

[MYSG13] Masoud Moshref, Minlan Yu, Abhishek Sharma, and Ramesh Govindan. Scalable rule management for data centers. In *NSDI*, 2013.

[net] Netperf manual. http://www.netperf.org/netperf/training/Netperf.html.

[NKN12] Karthik Nagaraj, Charles Killian, and Jennifer Neville. Structured comparative analysis of systems logs to diagnose performance problems. In *NSDI*, 2012.

[NMMVV] Radhika Niranjan Mysore, Ratul Mahajan, Amin Vahdat, and George Varghese. Gestalt: Unifying fault localization for networked systems. In *Under submission*.

[NMPF+09] Radhika Niranjan Mysore, Andreas Pamboris, Nathan Farrington, Nelson Huang, Pardis Miri, Sivasankar Radhakrishnan, Vikram Subramanya, and Amin Vahdat. Portland: A scalable fault-tolerant layer 2 data center network fabric. In *SIGCOMM*, 2009.

[NMPV] Radhika Niranjan Mysore, George Porter, and Amin Vahdat. Fastrak: Enabling express express lanes in multi-tenant data centers. In *Under submission*.

[OA11] Adam J. Oliner and Alex Aiken. Online detection of multi-component interactions in production systems. In *DSN*, 2011.

[ope] OpenFlow. www.openflowswitch.org/.

[otICS01] LAN/MAN Standards Committee of the IEEE Computer Society. IEEE Standard for Local and Metropolitan Area Networks, Common Specifications Part 3: Media Access Control (MAC) Bridges Ammendment 2: Rapid Reconfiguration, June 2001.

[ovs] OPEN VSWITCH. http://openvswitch.org/.

[PED+09] Radia Perlman, Donald Eastlake, Dinesh G. Dutt, Silvano Gai, and Anoop Ghanwani. Rbridges: Base Protocol Specification. Technical report, Internet Engineering Task Force, 2009.

[PKC+12] Lucian Popa, Gautam Kumar, Mosharaf Chowdhury, Arvind Krishnamurthy, Sylvia Ratnasamy, and Ion Stoica. Faircloud: sharing the network in cloud computing. In *SIGCOMM*, 2012.

[PYK+10] Lucian Popa, Minlan Yu, Steven Y. Ko, Sylvia Ratnasamy, and Ion Stoica. Cloudpolice: taking access control out of the network. In *Proc. Workshop on Hot Topics in Networks*, 2010.

[rfc] Address Allocation for Private Internets. http://tools.ietf.org/html/rfc1918.

[Ris05] Irina Rish. Distributed systems diagnosis using belief propagation. In *Allerton Conf. on Communication, Control and Computing*, 2005.

[RS07] Himanshu Raj and Karsten Schwan. High performance and scalable i/o virtualization via self-virtualized devices. In *High Performance Distributed Computing*, 2007.

[RTA01] Thomas L. Rodeheffer, Chandramohan A. Thekkath, and Darrell C. Anderson. SmartBridge: A Scalable Bridge Architecture. In *Proceedings of ACM SIGCOMM*, 2001.

[RVR+07] Barath Raghavan, Kashi Vishwanath, Rama Ramabhadran, Kenneth Yocum, and Alex Snoeren. Cloud control with distributed rate limiting. In *SIGCOMM*, 2007.

[RWM+06] Patrick Reynolds, Janet L. Weiner, Jeffrey C. Mogul, Marcos K. Aguilera, and Amin Vahdat. Performance debugging for distributed systems of black boxes. In *WWW*, 2006.

[SBB+91] Michael D. Schroeder, Andrew D. Birrell, Michael Burrows, Hal Murray, Roger M. Needham, Thomas L. Rodeheffer, Edwin H. Satterthwaite, and Charles P. Thacker. Autonet: A High-Speed, Self-Configuring Local Area Network Using Point-to-Point Links. In *IEEE Journal On Selected Areas in Communications*, 1991.

[SC08] Malcolm Scott and Jon Crowcroft. MOOSE: Addressing the Scalability of Ethernet. In *EuroSys Poster session*, 2008.

[SKG+11] Alan Shieh, Srikanth Kandula, Albert Greenberg, Changhoon Kim, and Bikas Saha. Sharing the data center network. In *NSDI*, 2011.

[sri] PCI-SIG SR-IOV Primer: An Introduction to SR-IOV Technology. http://www.intel.com/content/www/us/en/pci-express/pci-sig-sr-iov-primer-sr-iov-technology-paper.html.

[SS04a] Malgorzata Steinder and Adarshpal Sethi. Probabilistic fault localization in communication. In *IEEE/ACM Trans. Networking*, 2004.

[SS04b] Malgorzata Steinder and Adarshpal Sethi. A survey of fault localization techniques in computer networks. In *Science of Computer Programming, Special ed. on Topics in System Administration*, 2004.

[str] stress. http://linux.die.net/man/1/stress.

[stt] A Stateless Transport Tunneling Protocol for Network Virtualization. http://tools.ietf.org/html/draft-davie-stt-02.

[tc] tc. http://linux.die.net/man/8/tc.

[TP09] J. Touch and R. Perlman. Transparent Interconnection of Lots of Links (TRILL):Problem and Applicability Statement, 2009.

[VPC] Amazon VPC Limits. http://docs.aws.amazon.com/AmazonVPC/latest/UserGuide/VPC_Appendix_Limits.html.

[vrf] Virtual Routing and Forwarding. http://en.wikipedia.org/wiki/Virtual_Routing_and_Forwarding.

[vxl] VXLAN. http://datatracker.ietf.org/doc/draft-mahalingam-dutt-dcops-vxlan/.

[YRFW10] Minlan Yu, Jennifer Rexford, Michael J. Freedman, and Jia Wang. Scalable flow-based networking with difane. In *SIGCOMM*, 2010.