TOWARDS SYSTEMATIC DIAGNOSIS FOR CLOUD NETWORKS

by

Wenfei Wu

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

Doctor of Philosophy
(Computer Sciences)

at the
UNIVERSITY OF WISCONSIN–MADISON
2015

Date of final oral examination: 9/29/2015

The dissertation is approved by the following members of the Final Oral Committee:

Srinivasa A. Akella, Associate Professor, Computer Science
Remzi H. Arpaci-Dusseau, Professor, Computer Science
Michael M. Swift, Associate Professor, Computer Science
Shan Lu, Associate Professor, Computer Science, University of Chicago
Xinyu Zhang, Assistant Professor, Electrical and Computer Engineering

To my parents.

ACKNOWLEDGMENTS

Pursuing a Ph.D. at the University of Wisconsin-Madison has been one of the most wonderful experiences of my life. I would like to take this opportunity to thank the people who guided, inspired, and accompanied me to go through this memorable experience.

First and foremost, I would like to thank my adviser Professor Aditya Akella. In addition to teaching me how to conduct research, his enthusiasm and focus on research set a good example for me. I am especially grateful for his support and patience during my difficulties research.

I would like to thank the members of my thesis committee, Remzi Arpaci-Dusseau, Michael Swift, Shan Lu, and Xinyu Zhang. Their valuable suggestions and comments greatly improved this thesis.

I would like to thank my mentors during my several internships, Chuanxiong GPO, Yoshio Turner, Michael Schlansker, Anees Shaikh, Guohui Wang, Alex Tessmer, and Li Erran Li. I appreciate the industrial internship experience I gained while working with them, particularly the chance to learn about and solve practical problems.

I would like to thank my fellow graduate students and postdocs, Ashok Anand, Theophuius Benson, Shan-Hsiang Shen, Aaron Gember-Jacobson, Robert Grandl, Junaid Khalid, Chaithan Prakash, David Tran-Lam, Raajay Vishwanathan, Xiaoyang Gao, Ramakrishnan Durairajan, and Brent Stephens. Working with these brilliant people is a great gift to me. I am also quite fortunate to have great friends around me, who constantly supported and cheered me: Keqiang He, Linhai Song, Yizheng Chen, Ao Ma, Yupu Zhang, Lanyue Lu, and Suli Yang.

Last but not least, I would like to thank my family. My parents always stand behind me with their love, support, and encouragement, and have been great sources of inspiration

throughout the highs and lows of my Ph.D. work. I would also like to thank my relatives, who inspired and supported me.

TABLE OF CONTENTS

Page

Appendix

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

Appendix
Figure Page

ABSTRACT

Cloud outage can result in bad user experiences for cloud tenants and revenue loss to the provider. This makes cloud network diagnostic solutions invaluable. Despite the various existing network diagnostic solutions, few of them are designed specifically for cloud networks. Current state-of-the-art cloud network diagnosis falls short in three aspects: (1) there is no clear way to distinguish whether an observed problem is in the tenant's virtual network or in the provider's infrastructure. As a result, the interaction between tenants and the provider leads to a longer problem-solving time and higher maintenance costs; (2) for cloud tenants, there are only rudimentary troubleshooting tools (e.g., ping, VM monitoring) that can be deployed. However, diagnosing a distributed system with these tools depends heavily on skill and experience, which is not always feasible for tenants; (3) for the cloud provider, new trends such as network function virtualization make the infrastructure more complex than the traditional network, which could lead to new problems arising. Thus, the provider requires new diagnostic tools to help cover this range of problems.

In this thesis, we design two systems for cloud network diagnosis: (A) *VND: a Virtual Network Diagnostic Service.* VND is a service offered by the provider to its tenants. Using VND, a tenant can determine whether a problem is in its virtual network or not; VND's interfaces also simplify tenants' troubleshooting operations. A tenant could use VND to collect, parse and query its packet traces. Here, the trace collection cooperates within the tenant's view of its own virtual network without exposing the cloud infrastructure. Trace parse and query interfaces are design to ease the tenant's troubleshooting operations. VND provides a SQL interface for tenants to perform diagnosis. We show several typical network diagnostic

use cases where troubleshooting solutions can be easily implemented using VND. We also measure VND's overhead and show its feasibility. (B) *PerfSight: Performance Diagnosis for Software Data Planes.* Increasingly, modern network data planes have complex software involved in packet processing (e.g., virtual switches, VM hypervisors and software middleboxes). We refer to these software parts as the software data plane. We argue that there are at least three new classes of performance problems that arise in software data planes: bottlenecks, contentions and bugs. We propose a system named PerfSight to target these three problems. PerfSight instruments the software data plane, gathers basic statistics (e.g., packet count, byte count, I/O time) and analyzes the statistics comprehensively. We obtained two key insights by running PerfSight: (1) packet drop is the best indicator of bottlenecks, and location of packet drop can give information on the resources in contention (e.g. CPU, network bandwidth); (2) software middlebox's states can be defined by basic statistics, and these states propagate in the network in certain patterns. These patterns can be used to infer which middlebox has performance bugs. Our evaluation shows PerfSight introduces little overhead to the existing system, and thus it is feasible to deploy.

Together, we believe VND and PerfSight provide diagnostic solutions to both tenants and the provider. They form an integral basis for cloud network diagnosis.

Chapter 1

Introduction

Over the past few years, cloud-based networking solutions have been gaining widespread acceptance and deployment for organizations such as enterprises and institutions. Today's data centers are infrastructures of various services, including search, content distribution, big data analysis, and virtual networks. The cloud providers obtain revenue by multiplexing the infrastructure among the cloud tenants and the tenants save the trouble of needing to manage the network by themselves. Recent studies have forecast that global data center traffic (both inter and intra) in 2018 will be triple the traffic in 2013 [3].

Despite their importance, cloud networks still face the risk of performance, availability and reliability issues. In April 2011, AWS performed a network change for scaling, and shifted the traffic to a low capacity EBS network, which caused a fault whereby the EBS nodes were unable to find replicas. As such the EBS nodes became "stuck" in searching for replicas —the outage lasted for 12 hours! In April 2013, a misconfiguration with an invalid address for authentication servers was released to the Google API production environment causing the internal monitoring systems to become blocked and all of the serving threads to be consumed; as a result, Google API experienced a one-hour long outage.

Performance, availability and reliability issues affect the productivity and revenue of the cloud providers and tenants. For example, in 2009, AWS experienced an 24-hour outage, which cost a revenue loss of about $4,320,000 [35]. In August 2013, an AWS outage causes its high profile tenants Netflix and Airbnb to go down as well. Motivated by these observations, this thesis proposes two diagnostic solutions—VND and PerfSight —for cloud network problems. They together form an integral basis of cloud network diagnosis.

In the rest of this chapter, we first describe the cloud infrastructure (Section 1.1). Then, in section 1.2, we discuss the new challenges for troubleshooting in cloud networks compared with the traditional networks. Section 1.3 briefly presents our network troubleshooting approaches for the new challenges. Section 1.4 provides a road map for the rest of thesis.

1.1 Cloud Infrastructure Organization

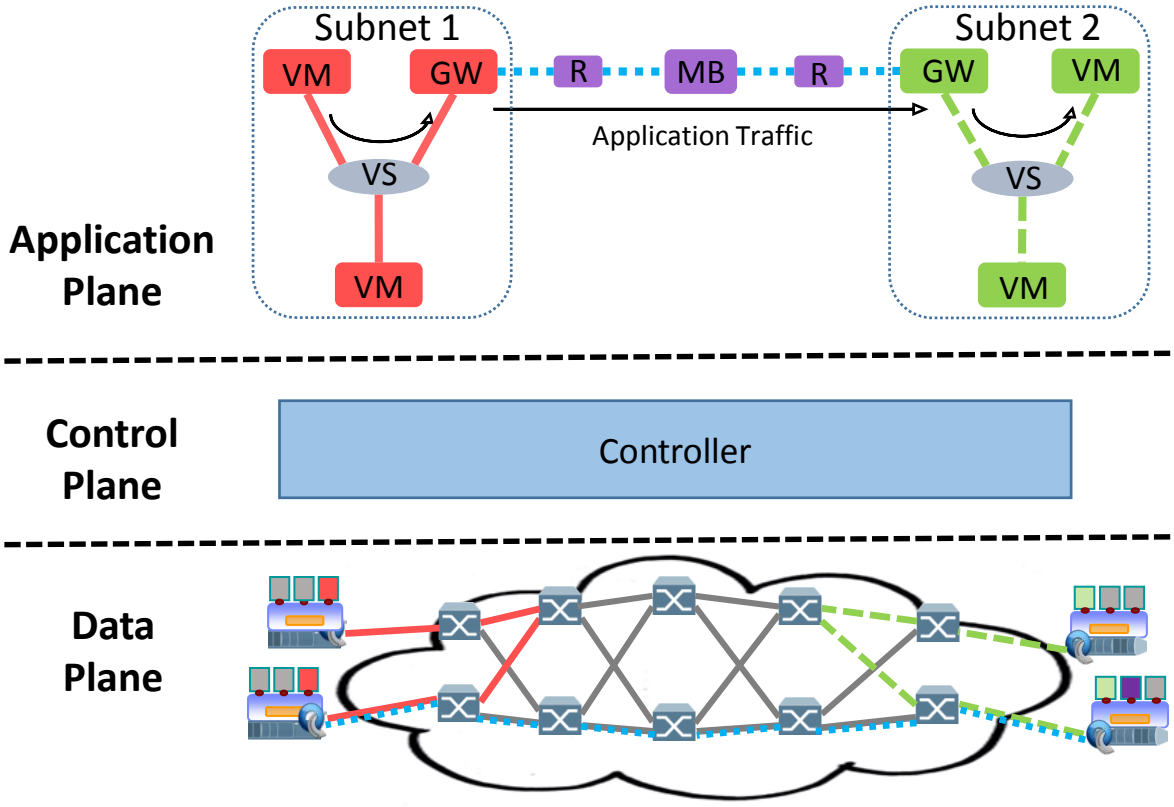


Figure 1.1: Three planes in the cloud architecture

In this thesis, we focus on multi-tenant cloud data centers. In this setting, tenants deploy *virtual private clusters* composed of application end-points (this could be service software in the case of private data centers, or VMs in the case of public clouds), network function services (or middeboxes), and logical links between subsets of them. Network functions

improve network performance or security, and tenants increasingly desire to deploy sophisticated sets of middlebox functionality within their clusters [52].

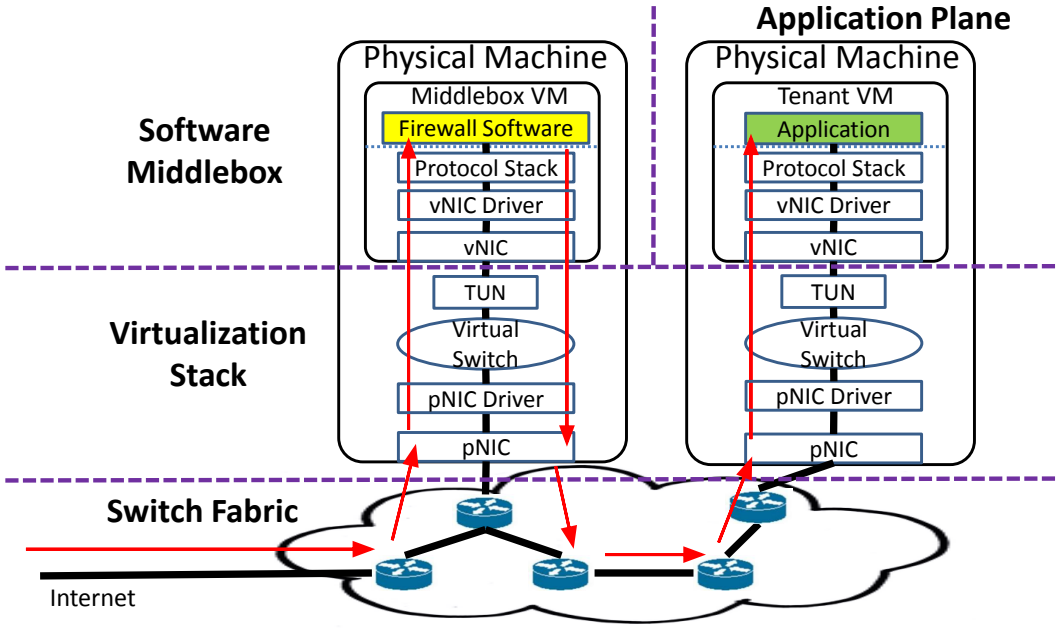
This setting can be viewed logically as being composed of three planes (Figure 1.1): the *control plane*, *application plane* and *data plane*. Tenants interact with the application plane, requesting (re)deployments of virtual private clusters. The control plane, which the cloud operator runs, responds to such requests by computing suitable deployment policies, e.g., determining where VMs and middleboxes ought to be placed, instantiating virtual links between VMs and middleboxes (using tunneling schemes or encapsulation policies [52]), and computing the forwarding state configuration to determine how traffic flows between VMs/middleboxes and over virtual links. Then the data plane for the tenant's virtual cluster, where fast path actions are performed on the tenant's traffic, follows the configurations provided by the control plane to deliver network traffic between the appropriate end-points in each virtual cluster.

In this thesis, we focus on middleboxes that are implemented as software and deployed in VMs attached to virtual switches, which is an increasingly popular trend also known as *network functions virtualization (NFV)* [30]. In NFV, the middlebox VMs, similar to application VMs, are allocated fixed resources (e.g., CPU, memory, network bandwidth), and the controller deploys all VMs to physical machines that have sufficient resources.

Figure 1.2(a) shows a tenant with a simple virtual cluster, consisting of an application VM and a firewall; furthermore, the tenant requires all Internet traffic to traverse through the firewall. While this is a simple example, virtual clusters can have much more complex sequences of middleboxes, where a given sequence may only apply to a specific traffic substream.

The physical deployment computed by the controller is shown in Figure 1.2(b). Consider the path that Internet-originated traffic would traverse to arrive at the tenant VM (and vice versa for the outgoing traffic), each packet is forwarded by the cloud gateway to the physical NIC (pNIC) of a physical server hosting the middlebox VM first; then it traverses the pNIC driver, the virtual switch, the hypervisor I/O handler, the virtual NIC (vNIC), the vNIC driver and the VM guest OS network stack, and finally arrives at the software firewall. After being

(a) An example of a virtual network



(b) Deployment in a data plane

Figure 1.2: Data plane organization

processed by the software firewall, the packet traverses all the layers back down to the pNIC. Then it is delivered by the physical network to the next hop in the virtual network (a tenant VM or another middlebox).

1.2 Challenges in Diagnosing Cloud Networks

Troubleshooting network problems is difficult. A network is a distributed system, where the states are distributed across devices and software components. With current rudimentary tools (trace, ping, SNMP, NetFlow, sFlow), troubleshooting network problems depends heavily on the operators' skill and experience, which lead to a long problem-solving time and high operational costs [42].

In addition to this common difficulty, cloud networks have their own characteristics, which makes troubleshooting even more difficult.

- Cloud networks have higher complexity than traditional networks. In data planes, network virtualization introduces more software components, including software switches, hypervisors, and so on; software middleboxes are introduced to support network function virtualization. In control planes, the cloud controllers need to perform more functions to set up virtual networks; the controllers map tenants' requirements from a logical view to a physical view, and finally to devices rules. Cloud networks involve a greater number of components involved, and these components may have logical or physical dependency (e.g., exchanging data or sharing hardware resources). This complexity in cloud networks makes them not only more error-prone but also difficulty to manage and diagnose.

- The two roles in cloud networks makes the management trickier than traditional network. The two roles are the provider (or operator) and the tenants, and their information is isolated from each other. In the case of network problems, the interaction between the provider and the tenants is crucial for problem solving; however, this process is usually of low efficiency. The tenants observe misbehaviors of their applications directly, but they lack diagnostic tools. The provider does not know the applications' performance, so they can only wait for tenants' tickets. The two parties need to exchange observations to figure out the problem. This manual process increases the problem-solving time and maintenance cost.

- The visibility of cloud networks is not well provided to customers and operators. In traditional networks, network devices and components provide various information for operators to monitor or troubleshoot, e.g., packet drop statistics in switches and the network protocol stack. However, by our study, this kind of visibility is not well preserved in cloud networks. For customers, the cloud provider does not provide visibility of how their packets traverse the network for security reasons. For the provider,

some virtualization components are introduced to cloud networks without keeping the visibility for diagnosis—there are several silent packet drops in VM hypervisors and software middleboxes. This is partially due to the fact that developers typically focus on functionality instead of diagnostic features in the first few versions of the software.

1.3 An Overview of Our Approach

Ever since the birth of networks, network problems have been appearing and upseting network operators. There have been numerous proposals aiming to solve various problems. For example, there are standards or technologies such as SNMP, sFlow, and NetFlow to collect states in networks; there are network models and diagnostic algorithms to discover culprits in network problems; and there are various network diagnostic solutions for traditional networks and software-defined networks, which combine the technologies and algorithms. These approaches are undoubtedly valuable in their specific scenarios.

However, these solutions do not overcome the challenges in diagnosing cloud networks (Section 1.2). In this thesis, we make a study of problems in cloud networks. According to the cloud organization, we look into each planes and summarize new problems. In more details, we found that in application planes, the isolation between tenants and the provider causes tenants unable to diagnose their virtual networks, thus we propose, design and implement a virtual network diagnostic service in application planes. We found that in data planes, increasing complexity and lack of visibility cause performance problems difficult to be identified, thus we modify software data planes, collect statistics and leverage the statistics to perform accurate diagnosis. We also find inconsistency issues in control planes, where tenant requirements may not be consistency with physical device states; we leave this problem in future works.

Decomposing cloud networks into three planes and reconsidering diagnostic challenges in cloud networks is an efficient way to discover, abstract and solve cloud network problems. We feel that, owing to our approach, several cloud-specific problems are discovered. Therefore, our works improve the reliability of cloud networks.

1.4 Thesis Outline

The rest of this thesis is organized as follows. In Chapter 2, we make a study of cloud network problems and briefly describe our approaches. In Chapter 3, we describe our design of the virtual network diagnostic service. In Chapter 4, we present our solution for software data plane diagnosis. In Chapter 5, we discuss the related work, Finally in Chapter 6, we conclude this thesis and discuss options for future work.

Chapter 2

Background and Approach

Network diagnosis has been a research topic ever since the appearance of networking systems. A number of network diagnostic solutions have been proposed to solve network problems. These approaches provide visibility to network internal states [20, 13, 40, 65], locate faulty components in networks [66, 33, 31], and verify network-wide invariants [47, 46, 56].

Cloud infrastructures have gained wide deployment and acceptance in recent years. The cloud networks are more complex than traditional networks, have two roles involved in network management, and lacks visibility of their internal states. These properties make the existing diagnostic approaches difficult to deploy. For example, due to security issue, tenants do not have access to the infrastructure, and thus cannot collect state information in the network or perform diagnosis. In the cloud, network virtualization introduces several software components without providing enough visibility. Thus cloud operators lack a means of understanding how packets traverse the networks and cannot diagnose these software components efficiently.

In this chapter, we describe our methodology for collecting network problem logs from various sources, and analyzing and summarizing problems in different planes in cloud networks (Section 2.1). Then we briefly describe our approaches to solving these problems (Section 2.2).

2.1 Cloud Network Problems

2.1.1 Data Set

Device inventory/configurations and trouble ticket logs. We collected this set of data from a large online service provider (OSP) [37]. A device inventory records the vendor, model, location, and role (switch, router, load balancer, etc.) of every device in their deployment, and the network it belongs to. Device configuration snapshots are taken whenever device configurations are changed. When users report network problems, or monitoring systems raise alarms, a trouble ticket is created in an incident management system. The ticket is used to track the duration, symptoms, and diagnosis of the problem. In total we collected device configurations and ticket logs from over 850 networks over a period spanning 17 months.

Bug and outage reports. We collected the software bug tracks of several software components in the software data plane. The software includes KVM, OVS, balance, MySQL, and NFS [6, 11, 9, 63, 10]. We collected 57 bugs in total. We also collected the outage reports of Microsoft Azure and Amazon Web Service, resulting in 9 outage reports [7, 8].

2.1.2 Analysis

We categorize the records in our data set according to the planes where the problem of a record belongs to. We find management problems in application planes, performance problems in data planes, and consistency issues in control planes. In this thesis, we focus on application and data plane issues, and leave the control plane issues for future works.

Application Plane. In public clouds, the application plane runs tenants' virtual networks. As described in Section 1.1, virtualization abstracts the underlying details, so the virtual networks are isolated from the operator and from each other. To set up a virtual cluster, a tenant needs to input the virtual network topology (VMs, middleboxes and virtual links), configure network function services and deploy its own applications. However, among these inputs, the tenant only has access to its own VMs. That is, the tenant lacks the necessary visibility

to perform troubleshooting. For example, if a network flow which is designed to traverse a network service (e.g., cache) fails to get through, there is no way for the tenant to discover whether there is a misconfiguration in the network service or a fault in the underlying switch fabric drops packets [1].

The solutions currently deployed in tenant VMs are usually simple point tools (e.g., ping, VM resource monitoring). To diagnose a distributed system with these tools requires strong skills and experiences, which is not always applicable for cloud tenants. Tenants can also ask for help from the cloud provider's customer service. However, this takes extra time and typically incurs additional costs to both parties, which hinders the agility of the cloud computing.

Data Plane. In the cloud infrastructure, a part of the network data plane is implemented in software, e.g., software middleboxes and virtual switches. The software data plane is more complex than the traditional network stack and presents new challenges in fault diagnosis. We argue that software data planes are much more vulnerable to network problems due to three reasons.

First, the more number of software components (compared with the traditional network stack) increases the risk of problems and the difficulty to locate a problem. In current implementation, when a packet is dropped while traversing the software data plane, there is little clue as to where it is dropped. Second, the software components run on a shared computing platform (i.e., the physical machine), because of which they potentially could contend for computing resources (e.g., CPU, memory, disk). To make things worse, some contentions are implicit, which makes the detection trickier. For example, network traffic and memory traffic both go through the bus, but there is no way to detect their contention. Third, packet processing in middleboxes is usually stateful and middleboxes may logically depend on each other. As a result, problems arising in one middlebox may propagate along the end-to-end path to other middleboxes, which complicates root cause diagnosis.

[1] Traceroute does not work. Because the tenant's packets are encapsulated while traversing the network, the inner packet's TTL is not changed.

2.2 Our Approaches

In this section, we first focus on the management problem in application planes. We propose a virtual network diagnostic service offered by the cloud provider to tenants, namely *VND*. With VND, a tenant can determine whether a problem is in its virtual network, and the tenant can also perform systematic diagnosis to its own virtual network. Then we target performance problems in data planes with the goal of aiding the cloud operator to diagnose and fix them. To this end, we design a new system, called PerfSight.

VND: Virtual Network Diagnostic Service. The Virtual Network Diagnostic service enables a cloud provider to offer sophisticated virtual network diagnosis as a service to its tenants. When diagnosing a fault, a tenant uses VND interfaces to perform trace collection, trace parsing and trace analysis. In trace collection, a tenant specifies a set of flows to be gathered in its virtual network; then the setting is translated and deployed in the infrastructure and packet traces are collected. In trace parsing, the tenant submits packet header fields of its interest so that those fields are extracted and stored in tables. In the trace query, VND provides a SQL interface to tenants for them to analyze their trace and to perform meaningful diagnosis.

In the whole troubleshooting process, the trace collection interface retains the existing virtualization abstraction without leaking information of the infrastructure and other tenants' virtual networks, while the trace parsing and query interfaces provide a more systematic view of the virtual network and simplify tenants' troubleshooting operations. VND distributes trace collection, parsing, storage and query to improve scalability; VND also leverages the current SDN switch design to avoid interfering with the existing network rules (e.g. routing). We give several use cases to show how VND helps troubleshooting and cloud network management.

PerfSight: Performance Diagnosis for Software Data Planes. We first identify three classes of performance problems that may arise in software data plane. They are resource

mis-allocation, contention and performance bugs. We address the challenges in solving them. Then we design a general system, called PerfSight, for accurate and quick diagnosis of these performance problems.

In PerfSight, we decouple information collection and problem analysis. To collect traffic information in the software data plane, we view it as a *pipeline of elements*, where an element is a logical unit that reads traffic from or writes traffic to another element via buffers or function calls. For each element, we instrument it to collect statistics such as packet count, byte count and I/O time.

Based on the statistics, we develop applications to diagnose the three performance problems. By identifying the exact packet drop location, we can infer whether an element is facing resource limitation or whether elements are contending for some resources; by inferring the middlebox states based on statistics, we can narrow down the root cause middlebox in the virtual topology.

The PerfSight design keeps the statistics light-weight so that little overhead is introduced into the existing system. The interface design also makes it easy to extend to add custom statistics. Our PerfSight diagnostic applications also accurately detect performance problems.

Chapter 3

Virtual Network Diagnostic Service in Application Planes

Recent progress on network virtualization has made it possible to run multiple virtual networks on a shared physical network, and decouple the virtual network configuration from the underlying physical network. Today, cloud tenants can specify sophisticated logical network topologies among their virtual machines (VMs) and other network appliances, such as routers or middleboxes, and flexibly define policies on different virtual links [27, 36]. The underlying infrastructure then takes care of the realization of the virtual networks by: for example, deploying VMs and virtual appliances, instantiating the virtual links, setting up traffic shapers/bandwidth reservations as needed, and logically isolating the traffic of different tenants (e.g., using VLANs or tunnel IDs).

While virtual networks can be implemented in a number of ways, we focus on the common overlay-based approach adopted by several cloud networking platforms. Examples that support such functionality include OpenStack Neutron [18], VMware/Nicira's NVP [4], and IBM DOVE [55]. Configuring the virtual networks requires setting up tunnels between the deployed VM instances and usually includes coordinated changes to the configuration of several VMs, virtual switches, and potentially physical switches and virtual/physical network appliances. Unfortunately, many things could go wrong in such a complicated system. For example, misconfiguration at the virtual network level might leave some VMs disconnected, or receiving unintended flows, rogue VMs might overload a virtual network with broadcast packets on a particular virtual or physical switch.

Because virtualization abstracts the underlying details, cloud tenants lack the necessary visibility to perform troubleshooting. More specifically, tenants only have access to their

own virtual resources, so there is no way for them to know how their packets traverses the network. And more crucially, each virtual resource may map to multiple physical resources, e.g., a virtual link may map to multiple physical links; as a result, extracting information of a virtual appliance needs to be mapped to actual physical location in the physical infrastructure. When a problem arises, there is no way today to systematically obtain the relevant data from the appropriate locations and expose them to the tenant in a meaningful way to facilitate diagnosis. In addition, depending on tenants to deploy diagnostic solutions also requires network operating skills and experience. We argue that this breaks the agility of cloud network service and is not a proper solution.

In this chapter, we make the case for VND, a framework that enables a cloud provider to offer sophisticated virtual network diagnosis as a service to its tenants. Extracting the relevant data and exposing it to the tenant forms the basis for VND. Yet, this is not trivial because several requirements must be met when extracting and exposing the data: we must preserve the abstracted view that the tenant is operating on, ensure that data gathering and transfer do not impact performance of ongoing connections, preserve isolation across tenants, and enable suitable analysis to be run on the data, while scaling to large numbers of tenants in a cloud.

VND exposes interfaces for configuring diagnosis and querying traffic traces to cloud tenants for troubleshooting their virtual networks. Tenants can specify a set of flows to monitor, and investigate network problems by querying *their own* traffic traces. VND controls the appropriate software switches to collect flow traces and distributes traffic traces of different tenants into "table servers". VND co-locates flow capture points with table servers to limit the data collection overhead. All the tenants' diagnosis queries run on the distributed table servers. To support diagnosis requests from many tenants, VND moves data across the network only when a query for that data is submitted.

Our design of VND leverages recent advances in software defined networking to help meet the requirements of maintaining the abstract view, ensuring low data gathering overhead and isolation. By carefully choosing how and where data collection and data aggregation happens, VND is designed to scale to many tenants. VND is a significant improvement over

existing proposals for enterprise network diagnosis, such as NDB [40], OFRewind [65], Anteater [56] and HSA [46], which expose all the raw network information. This not only leads to obvious scale issues, but also weakens isolation across tenants and exposes crucial information about the infrastructure that may open the provider to attack.

We show that several typical network diagnosis use cases can be easily implemented using the query interface, including throughput, RTT and packet loss monitoring. We demonstrate how VND can help to detect and scale the bottleneck middlebox in a virtual network. Our evaluation shows that the data collection can be performed on hypervisor virtual switches without impacting existing user traffic, and the queries can be executed quickly on distributed table servers. For example, throughput, RTT and packet loss can be monitored in real time for a flow with several Gbps throughput. We believe our work demonstrates the feasibility of providing a virtual network diagnosis service in a cloud.

The contributions of VND can be summarized as follows:

- our work is the first to address the problem of virtual network diagnosis and the technical challenges of providing such a service in the cloud;
- we propose the design of a VND framework for cloud tenants to diagnose their virtual network and application problems, including its architecture, interfaces and operations.
- we propose optimization techniques to reduce overhead and achieve scalability for the VND framework;
- we demonstrate the feasibility of VND through a real implementation, and conduct experiments measuring overhead along with simulations to show scalability.

The rest of this chapter is organized as follows. Section 3.1 introduces the challenges and necessity of a virtual network diagnosis framework. Section 3.2 discusses the overall view of how VND works. Section 3.3 and 3.4 give our VND design, which describe VND architecture and applications respectively. Section 3.5 presents our VND implementation.

We evaluate VND feasibility in Section 3.6, discuss the scope of VND in Section 3.7, and conclude this chapter in Section 3.8.

3.1 Background

In this section, we discuss the challenges in virtual network diagnosis in details and review the current network diagnostic approaches.

3.1.1 Challenges of Virtual Network Diagnosis

When setting up a virtual network, the tenant needs to coordinate multiple elements (e.g., firewall rules, routing adjacencies) in the virtual topology. A number of things could go wrong in configuring such a complex system, including incorrect virtual machine settings, or misconfigured gateways or middleboxes. To complicate matters further, failures can also occur in the underlying physical infrastructure elements which are not visible in the virtual networks. Hence, diagnosing virtual networks in a large scale cloud environment introduces several concomitant technical challenges described further below.

Challenge 1: Preserving abstractions. Tenants work with an abstract view of the network, and the diagnosis approach should continue to preserve this abstract view. Details of the physical locations from which data is being gathered should be hidden, allowing tenants to apply analyze data that corresponds to their logical view of the network.

Challenge 2: Low overhead network information collection. Most network diagnostic mechanisms collect information by tracing flows on network devices [40, 65]. In traditional enterprise and ISP networks, operators and users rely on the built-in mechanisms on physical switches and routers for network diagnosis such as NetFlow, sFlow or port mirroring. In the cloud environment, however, the virtual network is constructed on software components, such as virtual switches and virtual routers. Trace capture for high throughput flows imposes significant traffic volume into the network and switches. As the cloud infrastructure is shared among tenants, the virtual network diagnostic mechanisms must limit their impact on switching performance and the effect on other tenant or application flows.

Challenge 3: Scaling to many tenants. Providing a network diagnosis service to a single tenant requires collection of flows of interest and data analysis on the (potentially distributed) flow data. All these operations require either network bandwidth or CPU cycles. In a large-scale cloud with a large number of tenants who may request diagnosis services simultaneously, data collection and analysis can impose significant bottlenecks impacting both the speed and effectiveness of troubleshooting and also affecting prevalent network traffic.

Challenge 4: Disambiguating and correlating flows. To provide network diagnosis services for cloud tenants, the service provider must be able to identify the right flows for different tenants and correlate them among different network components. This problem is particularly challenging in cloud virtual overlay networks for two reasons: (1) Tunneling/encapsulation makes tracing tenant-specific traffic on intermediate hops of a tunnel difficult; (2) middleboxes and other services may transform packets, further complicating correlation. For example, NATs rewrite the IP addresses/ports; a WAN optimizer can "compress" the payload from multiple incoming packets into a few outgoing packets, etc.

3.1.2 Limitations of Existing Tools

There are many network diagnosis tools designed for the Internet or enterprise networks. These tools are designed to diagnose network problems in various settings, but due to the unique challenges of multi-tenant cloud environments, they cannot be used to provide virtual network diagnosis service. We discuss existing diagnosis tools in two categories: tools deployed in the infrastructure and tools deployed in the virtual machines.

Solutions deployed on network infrastructure, such as NDB [40], OFRewind [65], Anteater [56], HSA [46], Veriflow [47] and Frenetic [34] could be used in data centers to troubleshoot problems in network states. However, these tools expose all the raw network information in the process. In the context of the cloud, this violates isolation across tenants and may expose crucial information about the infrastructure that introduces vulnerability to potential attacks. In addition, these solutions are either inefficient or insufficient for virtual network diagnosis.

For example, OFRewind collects all control and data packets in the network, which introduces significant overhead in the existing network. NDB's trace collection granularity is constrained by the existing routing rules, which is not flexible enough for cloud tenants to diagnose specific application issues. Anteater, HSA, and Veriflow model the network forwarding behavior and can check the reachability or isolation, which is limited to analyzing routing problems; Frenetic focuses on operating each single switch without considering the virtual network wide problems.

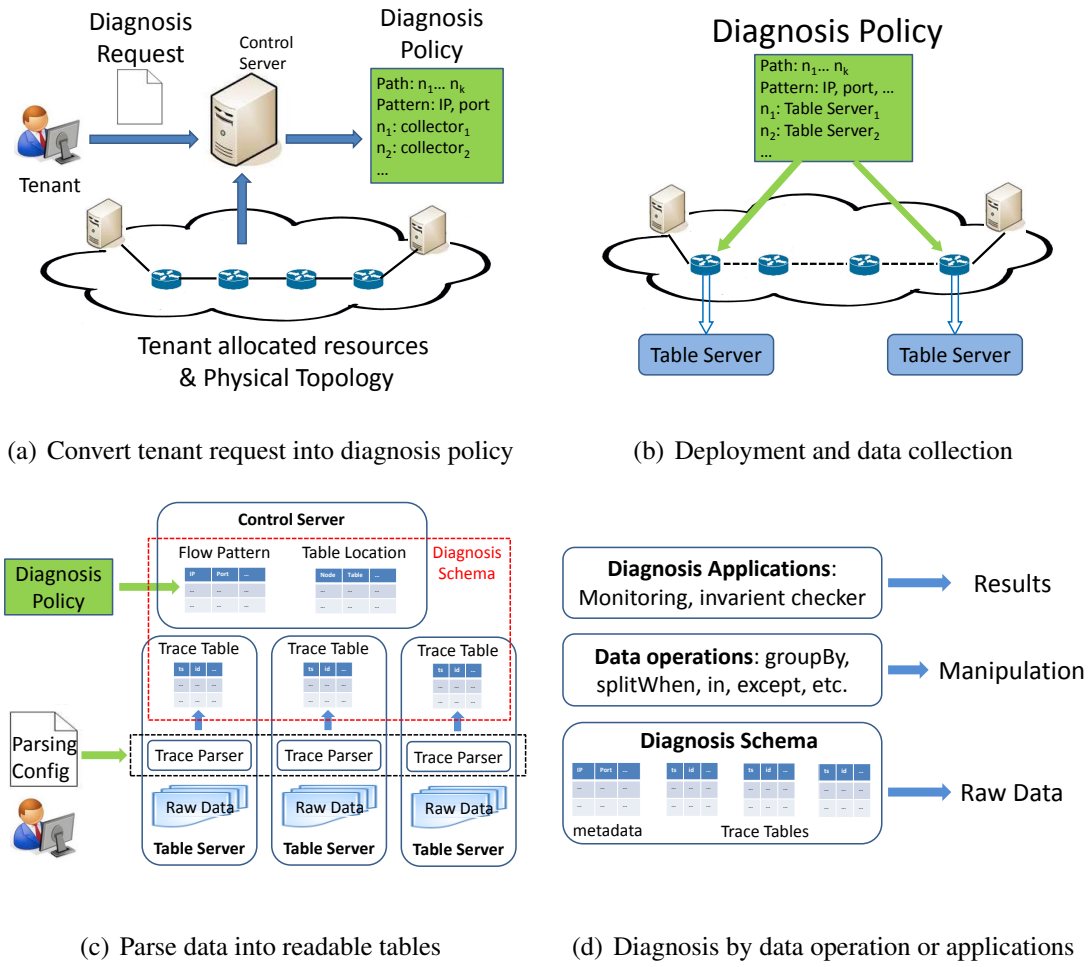
Many network monitoring or tracing tools, such as tcpdump, SNAP [66] and X-Trace [33] can be deployed in client virtual machines for network diagnosis. These tools are usually heavy-weight, however, and it may not be possible to apply these tools on virtual appliances, such as a distributed virtual router or a firewall middlebox. Second, and more importantly, simply collecting traffic traces is not enough to provide a virtual network diagnosis service. In such a service, tenants also need to be able to perform meaningful analysis that helps them tie the observed problem to an issue with their virtual network, or some underlying problem with the provider's infrastructure.

Thus, we need a new approach to enable virtual network diagnosis, which involves trace collection and analysis. This new approach should overcome the challenges in Section 3.1.1

3.2 VND Overview

Figure 3.1 illustrates the operations of VND's diagnosis service, which takes input from the tenants and produces the raw data, operational interfaces, and initial analysis results. We assume the cloud has the architecture as described in Section 1.1. There is a network controller that knows the physical topology and all tenants' virtual network embedding information (i.e., an SDN controller).

First, when a tenant observes poor performance or failure of his virtual network, he submits a diagnostic request to the VND control server (Figure 3.1(a)). The request describes the flows and components experiencing problems. The control server, which is deployed by the cloud administrator, accepts the tenant request and obtains the physical infrastructure



(a) Convert tenant request into diagnosis policy

(b) Deployment and data collection

(c) Parse data into readable tables

(d) Diagnosis by data operation or applications

Figure 3.1: VND service operations

details like topology and the tenant's allocated resources. The control server then translates the diagnosis request into a diagnosis policy. The diagnosis policy includes a flow pattern to diagnose (flow granularity such as IP, port, protocol, etc.), capture point (the physical location to trace the flow), and storage location (the physical server location for storage and further analysis).

Then, the cloud controller deploys this diagnosis policy into the physical network to collect the flow traces (Figure 3.1(b)). This deployment includes three aspects: 1) mirroring problematic flows' packets at certain capture points (physical or virtual switches), 2) setting up trace collectors to store and process the packet traces, and 3) configuring routing rules

from the capture point to the collector for the dumped packets. Now the tenant can monitor his problematic trace of his network applications.

Next, the tenant supplies a parse configuration that specifies packet fields of interest and the raw packet trace is parsed (Figure 3.1(c)), either offline after the data collection, or online as the application runs. The raw trace includes packets plus timestamps. The raw traces are parsed into human-readable tables with columns for each packet header field and rows for each packet; each trace table denotes a packet trace at a certain capture point. There is also a metadata table transformed from the diagnosis policy. All of these tables collectively form a diagnosis schema.

Finally, the tenant can diagnose the virtual network problem based on the trace tables (Figure 3.1(d)). The control server provides an interface to the tenants through which they can fetch the raw data, perform basic SQL-like operations on the tables and even use integrated diagnosis applications from the provider. This helps tenants diagnose problems in their own applications or locate problems in the virtual network. If the physical network has problems, tenants can still use VND to find the abnormal behavior (packet loss, latency, etc.) in observations of the virtual components, so that they can report the abnormality to the cloud administrator.

3.3 VND Architecture

In this section, we describe the design of our virtual network diagnosis framework (VND) to address the challenges outlined in the previous section. We show how the VND architecture preserves data isolation and abstraction, and demonstrate VND's applicability to existing cloud management platforms.

VND is composed of a *control server* and multiple *table servers* (Figure 3.2). Table servers collect flow traces from network devices (both physical and virtual), perform initial parsing, and store data into distributed data tables. The control server allows tenants to specify trace collection and parse configurations, and diagnose their virtual networks using

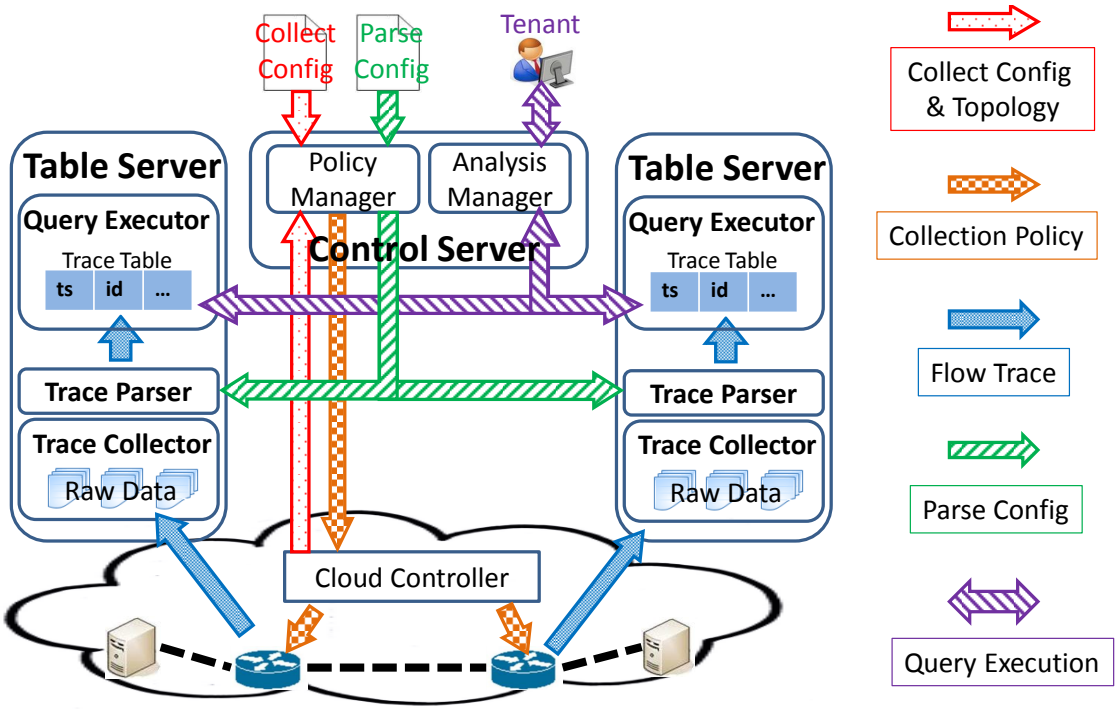


Figure 3.2: Virtual network diagnostic service framework

abstract query interfaces. To reduce overhead, trace collection and analysis begin only in reaction to the tenant's diagnosis requests.

3.3.1 Control Server

The control server is the communication hub between tenants, the cloud controller, and table servers. Its configuration and query interfaces allow cloud tenants to "peek into" problems in their logical networks without having the provider to expose unnecessary information about the infrastructure or other tenants. To decide how to collect data, the control server needs interfaces from the cloud controller to request virtual-to-physical resource mapping (e.g., placement of VMs or middleboxes, tunnel endpoints) and interfaces to set up data collection policies (e.g., flow mirroring rules, collector VM setup, and communication tunnels between all VND components).

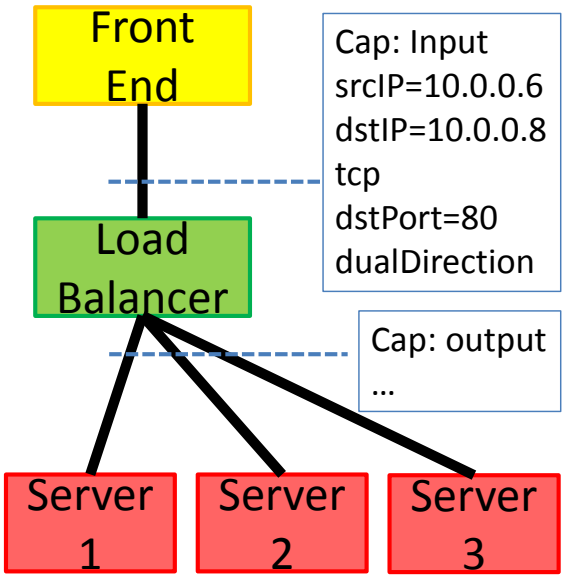
The *policy manager* in the control server manages the trace collection and parse configuration submitted by cloud tenants. When a tenant encounters problems in its virtual network,

```
1)  Virtual Appliance Link : l1
2)      Capture Point node1
3)          Flow Pattern field = value, ...
4)      Capture Point node2
5)  ...
6)  Virtual Appliance Node : n1
7)      Capture Point input, [output]
8)  ...
```

Figure 3.3: Trace collection configuration format

```
1)  Virtual Appliance Node : lb
2)      Capture Point input
3)          srcIP = 10.0.0.6/32
4)          dstIP = 10.0.0.8/32
5)          proto = TCP
6)          srcPort = *
7)          dstPort = 80
8)          dualDirect = True
9)      Cap output
10)         ...
11) Virtual Appliance ...
12)     ...
```

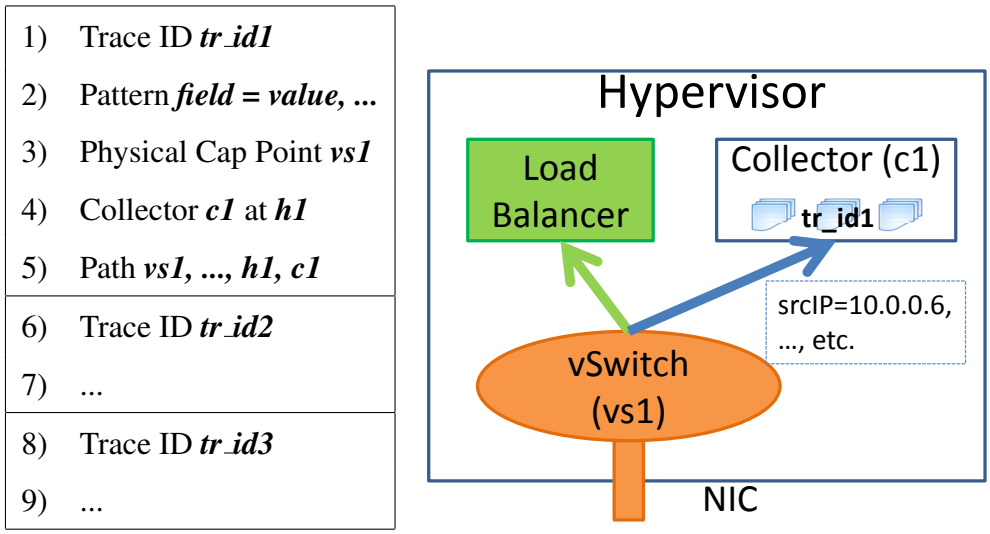
(a) Configuration



(b) Tenant's view

Figure 3.4: An example of trace collection

it can submit a *trace collection configuration* (Figure 3.3) that specifies the flow of interest, e.g., flows related to a set of endpoints, or application types (line 1, 2, 4, 6, 7). The pattern may be specified at different granularity, such as a particular TCP flow or all traffic to/from a particular (virtual) IP address (line 3).



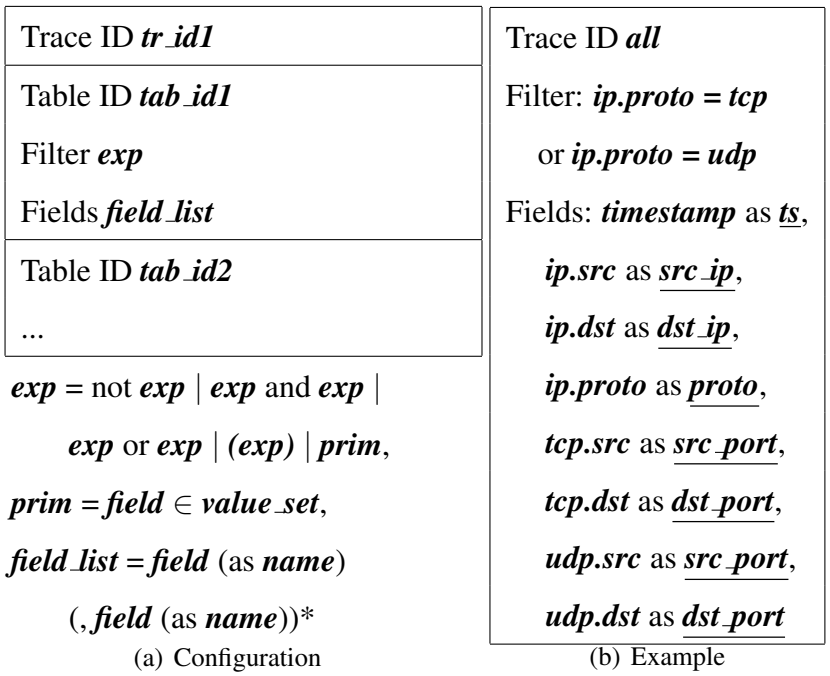
(a) Policy (b) Deployment

Figure 3.5: Trace collection policy

Figure 3.4 shows a trace collection configuration example. A tenant deploys a load balancer and multiple servers in his virtual network, and now he wants to diagnose the load balancer. He describes the problematic appliance to be the *node lb* (line 1), and captures both the input and output (line 2, 9). The flow of interest is the web service flow (port 80) between the host 10.0.0.6 and 10.0.0.8 (line 3-8). In the configuration, the tenant only has the view of his virtual network (Figure 3.4(b)) and the infrastructure is not exposed to the tenant.

The policy manager combines the trace collection configuration with network topology and the tenant's logical-to-physical mapping information. This is assumed to be available at the SDN controller, e.g., similar to a network information base [53] (not shown in the figure). The policy manager then computes a *collection policy* (Figure 3.5(a)) that represents how flow traces should be captured in the physical network. The policy includes the flow pattern (line 2), the capture points in the network (line 3), and the location of *trace collectors* (line 4), which reside in the table servers to create local network taps to collect trace data. The policy also has the routing rules to dump the duplicated flows for the capture point into the collector (line 5). We discuss the capture point and table server allocation algorithm in

Section 3.3.4.1. Based on the policy, the cloud controller sets up corresponding rules on the capture points to collect the appropriate traces (e.g., matching and mirroring traffic based on a flow identifier in OpenFlow), and it starts the collectors in virtual machines and configures routing rules between capture points and collectors (Figure 3.5(b)). We discuss how to avoid interference between diagnostic rules and routing rules in Section 3.3.4.2.



(a) Configuration (b) Example

Figure 3.6: Parse configuration and an example

Cloud tenants also submit a *parse configuration* in Figure 3.6(a) to perform initial parsing on the raw flow trace. It has multiple parsing rules, with each rule having a filter and a field list. The filter specifies the packets of the tenant's interest, and the field list describes the header fields to extract as well as the table columns to store the field values. Based on the parse configuration, the policy manager configures the *trace parser* on table servers to parse the raw traffic traces into multiple text tables, called trace tables, which store the packet records with selected header fields. Figure 3.6(b) shows an example parse configuration, in which all traces (line 1) in the current diagnosis are parsed. All the layer-4 packets including TCP and UDP (line 2, 3) are the packets of interest. The packets' 5-tuple fields,

i.e. source/destination IP, source/destination port and protocol, are extracted and stored in tables. In this configuration, the TCP and UDP's source/destination ports are stored in the same columns.

<ts, src_ip, dst_ip, proto, src_port, dst_port>.

Based on trace tables, tenants can perform various diagnosis operations through a query interface provided by the control server. The *analysis manager* in the control server takes the tenant's query, schedules its execution on distributed *query executors* on table servers, and returns the results to the tenant. In Section 3.4.1, we discuss typical diagnosis tasks that can be easily implemented using the query interface.

3.3.2 Table Server

A table server has three components, a trace collector, a trace parser and a query executor. The raw packets from the virtual NIC pass through these three components in a stream. The trace collector dumps all packets and transmits them to the trace parser. The trace parser which is configured by the policy manager, parses each packet to filter out packets of interest and extracts the specified fields. The extraction results are stored by the query executor in trace tables.

A query executor can itself be viewed as a database with its own tables; it can perform operations such as search, join, etc. on data tables. Query executors in all table servers form a distributed database which supports inter-table operations. We choose a distributed approach over a centralized one for two reasons. First, with distributed storage, VND only moves data when the query requires it, so it avoids unnecessary data movement and reduces network traffic overhead. Second, for all the diagnostic applications discussed in Section 3.4.1, the most common table operations are single-table operations. These operations can be executed independently on each table server, so distributed storage helps to parallelize the data queries and avoid overloading a single query processing node.

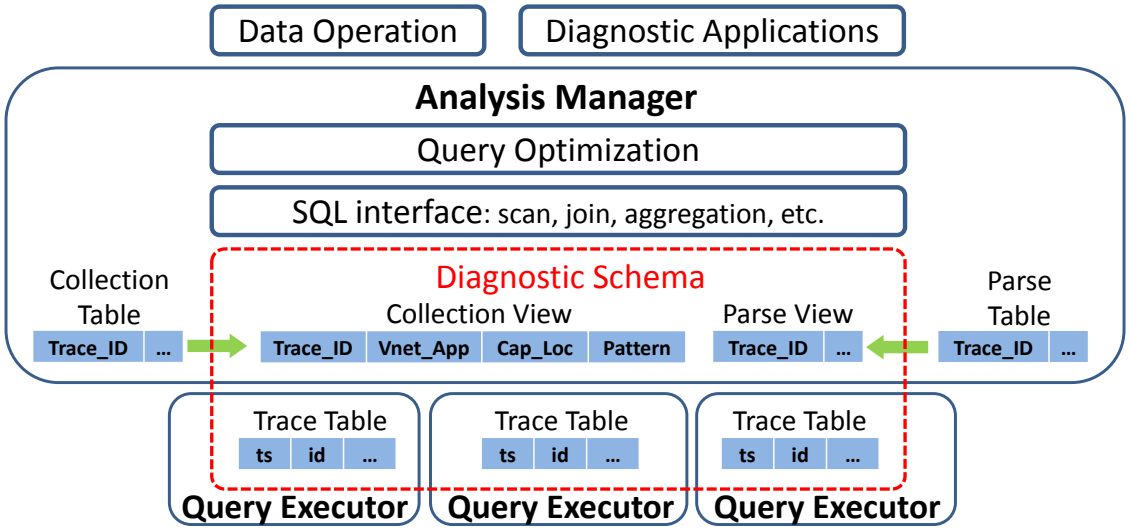


Figure 3.7: Analysis framework

3.3.3 Distributed Query Execution

The data analysis framework in Figure 3.7 can be viewed as running atop a distributed database. Each tenant's diagnosis forms a schema, including the metadata table in the analysis manager and trace tables in the query executors. The metadata table records how the tenant's traces are collected and parsed, and each tenant can only access its own information as a view. The trace tables are parsed traces which are distributed to query executors.

When a query is submitted to the analysis manager, the query is optimized to an execution plan as in typical distributed databases [54]. In VND, each table is placed locally at a query executor. This benefits the query execution: single-table operations do not need to move data across the network, and multiple table operations can predict the traffic volume introduced into the network, so the analysis manager is able to decide each executor's task to complete a query and make better execution plans, for example, using dynamic programming.

3.3.4 Optimizations

Below, we describe a number of optimizations to improve the performance and scalability of VND as the size of the data center and number of virtual network endpoints grow.

3.3.4.1 Local Table Server Placement

Replicating traffic from capture points to table servers is a major source of both bandwidth and processing overhead in VND. Flow capture points can be placed on either virtual or physical switches. Assuming all appliances (including tenant VMs, middleboxes and network services) participate in the overlay as a VM, the physical network works as a carrier of virtual links (tunnels) between these VMs. In this case, VND can always place capture points on hypervisor virtual switches. Virtual switches are the ends of virtual links, so it is easier to disambiguate traffic for different tenants here because packets captured there have been decapsulated from tunnels. Also, if the capture point is placed on a physical switch, the trace traffic must traverse the network to arrive at the trace collector, adding to the bandwidth overhead. Finally, current virtual switches, such as Open vSwitch (OVS), can support flexible flow replication using OpenFlow rules, which is supported in a relatively smaller (though growing) number of physical network devices. If a virtual network service is implemented in physical appliances, the trace capture points can be placed in the access switch or a virtual network gateway.

VND also places a table server locally on the same hypervisor with its capture point, which helps keep all trace collection traffic *local to the hypervisor*. Data movement across the network is needed only when a distributed query is executed. By allocating table servers in a distributed way around the data center, all data storage and computation are distributed in the data center. So VND's trace collection is scalable with the cluster and the virtual network size.

3.3.4.2 Independent Flow Collection Rules

Open vSwitch (OVS) allows us to capture a specific flow by installing OpenFlow rules to replicate the flow to its local table server. However, there may already be OpenFlow rules installed on the OVS for forwarding or other purposes. We have to make sure that the flow collection rules do not interfere with those existing rules. Similar problems have been addressed by tools like Frenetic [34].

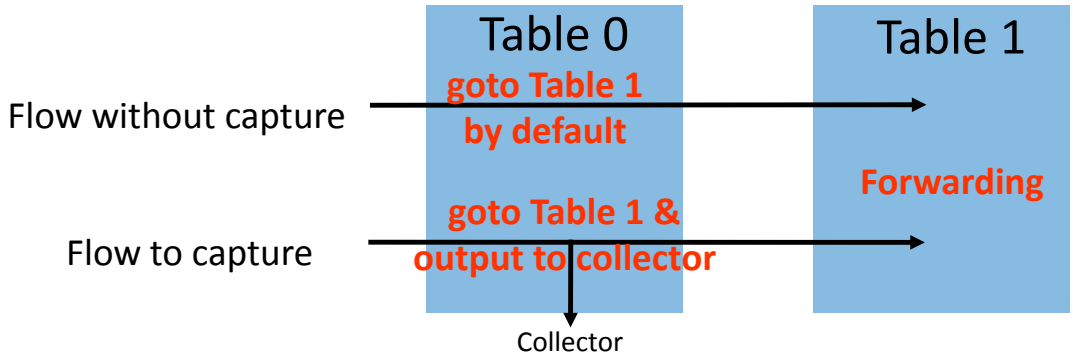


Figure 3.8: Flow capture with multiple tables

For example, if existing rules route flows by destination IP, and the tenant wants to monitor the port 80 traffic, the administrator needs to install (1) the monitoring rule for port 80, and (2) the overlapping flow space (IP, port 80) of both requirements. Otherwise the switch only uses one rule for the overlap part and ignores the other. To make things worse, when the cloud controller updates a routing rule, it must check whether there are diagnostic rules overlapping with it; if so, the cloud controller needs to update both the original rules and the overlapping rules. This way of managing the diagnostic routing rules not only causes excessive use of the routing table entries, but also adds complexity to the existing routing policy and other components.

VND solves this problem by using the multi-table option in Open vSwitch (Figure 3.8). We use two tables in VND with flow collection rules installed in Table 0 and forwarding rules written into Table 1. Table 0 is the first consulted table for any packet, where there is a default rule to forward packets to Table 1. When the administrator wants to capture a certain flow, new rules are added into Table 0 with actions that send packets to the table server port and also forward to Table 1. Using this simple approach, we avoid flow capture rules impacting existing rules on the same switch.

3.4 VND Applications

The tenant sends virtual network diagnostic requests via a SQL interface, and the diagnostic query is executed on the distributed query executors with distributed query execution

optimizations. In this section, we first present several network monitoring applications that can be easily implemented VND interfaces, and then illustrate a complicate example of flow correlation when a flow traverses middleboxes.

3.4.1 Network Monitoring Applications

VND provides a SQL interface to tenants, on which various network diagnosis operations can be developed. Tenants can develop and issue diagnostic tasks themselves or use diagnostic applications available from the cloud provider. VND makes use of existing SQL operations on relational databases, so that it supports a wide variety of diagnostic applications. Some of the queries are single-table queries, and others need to combine multiple tables. Single table queries are useful to identify anomalies in the input/output path of an appliance as is shown in the following examples.

Filter: With filters, the tenant can focus on packets of interest. For example, tenants may want to check ARP packets to find address resolution problem, they may want to check DNS packets for name resolution problems, or they may be interested in a certain kind of traffic such as ssh or HTTP. These filters are actually matching a field to a value and are easily described by a standard SQL query of the form:

```
select * from Table where field = value
```

Statistics: The tenants may need distributions of traffic on a certain field, such as MAC address, IP and port. These distributions can be used to identify missing or excessive traffic. Distribution computation first gets the count of records, and then calculate the portion of each distinct field value. These are described as:

```
var1 = select field, count(*) from tab group by field
var2 = select count(*) from tab
for each record r in var1
    Output <r.field, r.count/var2>
```

Groups: The unique groups among all packets records gives a global view of all traffic types. For example, identifying unique TCP connections of a web service helps identifying client IP distribution. In SQL, it is equivalent to finding the distinct field groups. Finding unique group query is described as:

```
select distinct field1, field2, ... from Table
```

Throughput: Throughput has a direct impact on application performance and is a direct indicator of whether the network is congested. To monitor a flow's throughput we first group the packet records by timestamp and then output the sum of payload lengths in each time slot. It can be implemented as follows:

```
# assume the timestamp unit is second
select ceil(ts), sum(payload_length)
    from table group by ceil(ts)
```

Combining or comparing multiple tables can help to find poorly behaving network appliances.

RTT: RTT is the round-trip delay for a packet in the network. Latency is caused by queuing of packets in network buffers, so RTT is a good indicator of network congestion. To determine RTT, we need to find a packet and its ACK, then use the difference of their timestamps to estimate the RTT. Assume the trace tables have the following format:

<ts, id[1] , srcIP, dstIP, srcPort, dstPort, seq, ack, payload_length>.

RTT monitoring is designed as follows:

```
1) create view T1_f as select * from T1
   where srcIP=IP1 and dstIP = IP2
2) create view T1_b as select * from T1
   where dstIP=IP1 and srcIP = IP2
3) create view RTT as select f.ts as t1, b.ts as t2
   from T1_f as f, T1_b as b
   where f.seq + f.payload_length = b.ack
4) select avg(t2-t1) from RTT
```

Note that the RTT computation discussed here is a simplified version. The diagnostic application could handle the more complicated logic of RTT computation in real networks. For example, retransmitted packets can be excluded from the RTT computation; in the case of SACK, a data packet's acknowledgment may be in the SACK field.

[1]Packet ID is used to identify each packet, and does not change with hops. This ID can be calculated from unchanged fields in the packets such as identification number in the IP header, sequence number in TCP header or hash of the payload.

Delay at a hop: Delay time of a packet on a hop indicates the packet processing time at that hop, which implies whether that hop is overloaded. To find the one-hop delay, we correlate input and output packets, and then calculate their timestamp difference. The SQL description is:

```
1) create view DELAY as select In.ts as t1,
   Out.ts as t2 from In, Out
   where In.id = Out.id
2) select avg(t2-t1) from DELAY
```

Packet loss: Packet loss causes TCP congestion window decrease, and directly impacts application performance. Finding packets loss at a hop requires identifying the missing packet records between the input/output tables of that hop. It is described as:

```
select * from In where In.id not in
   (select id from Out)
```

All the examples above are one-shot queries, and the applications can periodically pull new data from the VND executors. If an application wants to get a continuous data stream (e.g., traffic volume or RTT in each second), a proxy can be added between the distributed database and the application, which queries the database periodically and pushes data to the application.

3.4.2 Flow Correlation

In a virtual network, a flow usually traverses several logical hops, such as virtual switches and middleboxes. When cloud tenants experience poor network performance or incorrect behaviors in their virtual networks, the ability to correlate flows and trace the flows along their paths is necessary to locate the malfunctioning components. For example, when multiple clients fetch files from a set of back-end servers, and one of the servers provides corrupted files, with flow correlation on its path, one can follow the failed client's flow in reverse to the server to locate the malfunctioning server.

It is easy to identify a flow based on the packet header if packets are simply forwarded by routers or switches. However, middlebox devices may change the packet header or even

payload of incoming flows, which makes it very difficult to correlate the traffic flows on their paths. We summarize several flow trajectory scenarios and discuss how flows can be correlated in these cases.

(1) Packets pass a virtual appliance with some of its header fields unchanged. Examples of such appliances are firewalls or intrusion detection systems. We define a packet's fingerprint (packet ID) on those fields to distinguish it from other packets. We use SQL to describe the flow correlation:

```
select * from T1, T2 join by id
```

For example, the IP header has an identification field which does not change with hops; the TCP header has a sequence number which is unique in a flow if the packet is not retransmitted. We can define a packet ID by IP.id + TCP.seq $\ll$ 16. We add a field id in data tables to describe the packet ID. This ID can be used to correlate the packets into and out of a middlebox.

(2) Some appliances, such as NAT and layer-4 load balancers, may change the entire packet header but do not change packet payloads. In this case, a flow can be identified using its payload information. We define a packet's fingerprint (packet ID) as hash(payload) in the trace table. So packets in both input and output traces can still be joined by packet ID.

Recent work [32] proposes to add tags to the packets and modify middleboxes to keep the tag, so that a middlebox's ingress and egress packets can be mapped by tags. Another approach is to treat middleboxes as opaque and use a heuristic algorithm to find the mapping [60]. In the view of VND, both methods are giving the packets a fingerprint (in the latter case the fingerprint is not strictly unique) – VND can support both methods.

(3) There are still cases where the packet header is changed and the payload is not distinguishable from the input and output of certain appliances. For example, multiple clients fetch the same web pages from a set of backend servers via a load balancer. A layer-4 load balancer usually breaks one TCP connection into two, that is, the load balancer accepts the connection request from the client and starts another connection with backend servers. In

this case, a flow's header is totally changed; the input and output packet headers have no relation. If all clients fetch the same file, then the payload is also not distinguishable among all flows.

In this case, we use the flows' creation time sequence to correlate them. Usually, the load balancer listens to a port continuously. When a connection from the client is received, the load balancer creates a new thread or folk a new process in which the load balancer connects to one of the backend servers. So the first ACK from the client to the load balancer (the 3rd step in the 3-way shake) indicates that the client successfully connects with the load balancer; then the load balancer creates a new thread/process to connect to servers; the first SYN (1st step in 3-way shake) from the load balancer to the servers indicates the load balancer has started to connect with the servers. So if these two packets are ordered by arriving time sequence respectively. The two flows of these two packets should be in the same position in both sequences. This application is implemented in the following code.

```
create table inbound as fields, order
create table outbound as fields, order
var1 = select min(ts), fields from INPUT where
   ackflag = 1 group by srcIP, dstIP, srcPort, dstPort
index = 0
for record in var1
   insert into inbound <record, index++>
var2 = select min(ts), fields from OUTPUT where
   synflag = 1 group by srcIP, dstIP, srcPort, dstPort
index = 0
for record in var2
   insert into outbound <record, index++>
```

3.5 Implementation

We prototyped the VND on a small layer-2 cluster with three Dell T5500 workstations and one HP Procurve switch. Each workstation has 2 quad-core CPUs, a 10Gbps NIC and 12GB memory. The Open vSwitch and KVM hypervisor are installed in each physical server to simulate the cloud environment.

A table server is a virtual machine with a trace collector, a trace parser and a query executor. We implement a table server as a virtual machine image which can be deployed

easily in the cluster. The trace collector and trace parser are implemented in python using the pcap and dpkt package.

The query executor and the analysis manager in the control server are actually a distributed database system. We use MySQL Cluster to achieve their functions. We use MySQL daemon as the analysis manager and the MySQL Cluster data node as the query executor.

Policy manager is designed as a component integrated with the existing cloud management platform. We have not implemented this because the current platform (e.g., OpenStack) does not support the Openflow multi-table feature (Openflow 1.3). Without multi-table supported Openflow protocol, the routing control becomes very complicated as discussed in Section 3.3.4.2. Currently, we use shell scripts to set up cloud clusters and VND. In our experiment setup, we make use of the OVS's multi-table features.

VND cluster (composed of a control server and table servers) can be integrated with existing cloud platforms. VND cluster can be implement as a virtual cluster in the cloud, with the table servers as virtual machines and overlay communication among table servers and the control server. The difference between this virtual diagnostic cluster and a tenant's virtual cluster is that 1) VND is deployed by a network administrator, and 2) the VND control server can send trace duplication request to the cloud controller to dump the flow of interest.

3.6 Evaluation

Our evaluation is divided into the following sections: (1) Functional validation where we show how VND helps virtual network diagnosis. (2) Benchmark of our prototype where we evaluation VND's performance. (3) Overhead which is the impact introduced to the system by VND. (4) Scalability of VND.

3.6.1 Functional Validation

We first show the results of network monitoring, and then the flow correlation. Finally, we show a case how VND benefits middlebox management.

3.6.1.1 Network Monitoring

We show the results of networking monitoring applications. The virtual network topology is the same as in Figure 3.4 and 3.5. The applications are described in Section 3.4.1. The results are from the screenshot of VND interfaces.

In Figure 3.9, we *filtered* the first 5 packets of a flow in a time interval. In Figure 3.10, we searched for distinct source/destination IP pair *groups* of all flows. Figure 3.11 shows the *packet loss* when a flow's packets traverses the middlebox. Figure 3.12 is the result of *RTT* monitoring, where the capture point is the input of the middlebox. Figure 3.13 shows the *statistics* of packet count of each source/destination IP pair. Figure 3.14 shows the result of *throughput* of a flow.

```
mysql> select ts, src_ip, dst_ip, src_port from server
    -> where dst_port=5001 and ts<2 and ts>1 limit 5;
+----------------+-----------+-------------+----------+
| ts             | src_ip    | dst_ip      | src_port |
+----------------+-----------+-------------+----------+
| 1.00108504295  | 10.10.50.1 | 10.10.50.72 |    45587 |
| 1.00122404099  | 10.10.50.1 | 10.10.50.72 |    45587 |
| 1.00149011612  | 10.10.50.1 | 10.10.50.72 |    45587 |
| 1.00164294243  | 10.10.50.1 | 10.10.50.72 |    45587 |
| 1.00193595886  | 10.10.50.1 | 10.10.50.72 |    45587 |
+----------------+-----------+-------------+----------+
5 rows in set (0.00 sec)
```

Figure 3.9: Filter a flow

```
mysql> select distinct src_ip, dst_ip from server;
+-------------+-------------+
| src_ip      | dst_ip      |
+-------------+-------------+
| 10.10.50.1  | 10.10.50.72 |
| 10.10.50.72 | 10.10.50.1  |
+-------------+-------------+
2 rows in set (0.11 sec)
```

Figure 3.10: Group flows

```
mysql> select ceil(ts) as ts, src_ip, dst_ip, src_port,
    -> dst_port from client where client.id not in
    -> (select id from server);
+----+-------------+-------------+----------+----------+
| ts | src_ip      | dst_ip      | src_port | dst_port |
+----+-------------+-------------+----------+----------+
|  1 | 10.10.50.1  | 10.10.50.72 |    45587 |     5001 |
|  1 | 10.10.50.1  | 10.10.50.72 |    45587 |     5001 |
|  3 | 10.10.50.1  | 10.10.50.72 |    45587 |     5001 |
|  6 | 10.10.50.1  | 10.10.50.72 |    45587 |     5001 |
|  8 | 10.10.50.1  | 10.10.50.72 |    45587 |     5001 |
| 12 | 10.10.50.1  | 10.10.50.72 |    45587 |     5001 |
| 13 | 10.10.50.1  | 10.10.50.72 |    45587 |     5001 |
| 14 | 10.10.50.1  | 10.10.50.72 |    45587 |     5001 |
| 16 | 10.10.50.1  | 10.10.50.72 |    45587 |     5001 |
| 18 | 10.10.50.1  | 10.10.50.72 |    45587 |     5001 |
+----+-------------+-------------+----------+----------+
10 rows in set (0.23 sec)
```

Figure 3.11: Packet loss

```
mysql> create view server_f as select * from server where
    -> src_ip = '10.10.50.1' and dst_ip = '10.10.50.72';
Query OK, 0 rows affected (0.04 sec)

mysql> create view server_b as select * from server where
    -> src_ip = '10.10.50.72' and dst_ip = '10.10.50.1';
Query OK, 0 rows affected (0.04 sec)

mysql> create view RTT as select f.ts as t1, b.ts as t2
    -> from server_f as f, server_b as b
    -> where f.tcp_seq+f.payload_length = b.tcp_ack;
Query OK, 0 rows affected (0.05 sec)

mysql> select avg(t2-t1) from RTT;
+--------------------+
| avg(t2-t1)         |
+--------------------+
| 0.000483895305344  |
+--------------------+
1 row in set (0.26 sec)
```

Figure 3.12: Compute RTT

```
mysql> select src_ip, dst_ip, count(*) from server
    -> group by src_ip, dst_ip;
+--------------+--------------+----------+
| src_ip       | dst_ip       | count(*) |
+--------------+--------------+----------+
| 10.10.50.1   | 10.10.50.72  |    55221 |
| 10.10.50.72  | 10.10.50.1   |    28943 |
+--------------+--------------+----------+
2 rows in set (0.10 sec)
```

Figure 3.13: Statistics

```
mysql> select ceil(ts), sum(payload_length)
    -> from server group by ceil(ts);
+-----------+-----------------------+
| ceil(ts)  | sum(payload_length)   |
+-----------+-----------------------+
|         0 |                     0 |
|         1 |               4255696 |
|         2 |               4620568 |
|         3 |               4231056 |
|         4 |               4132592 |
|         5 |               4209336 |
|         6 |               4293320 |
|         7 |               4465632 |
|         8 |               4514864 |
|         9 |               4448256 |
|        10 |               4789984 |
|        11 |               4488800 |
|        12 |               4426536 |
|        13 |               4688624 |
|        14 |               4455496 |
|        15 |               4145624 |
|        16 |               4458392 |
|        17 |               4349792 |
|        18 |               4361376 |
|        19 |                618000 |
+-----------+-----------------------+
20 rows in set (0.09 sec)
```

Figure 3.14: Throughput

3.6.1.2 Flow Correlation

We now test the methods to correlate flows as described in Section 3.4.2. First, we use packet fingerprints to correlate the input and output packets of middleboxes. We route

a flow to traverse an redundancy elimination middlebox (RE) and an intrusion detection middlebox (IDS), then use packet id (ip.Identification + tcp.Sequence $\ll$ 16) to correlate packets between each logical hops. We compare two 0.5 GB traces and find that all packets are correlated unless dropped at the hop.

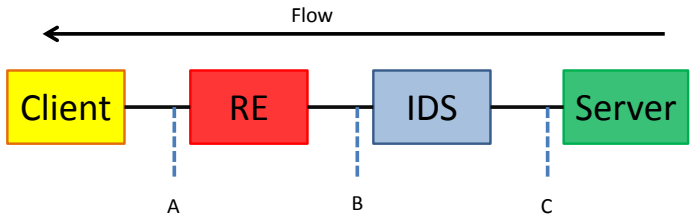
Then we look into the load balancer case, in which packets have no fingerprints. We use a load balancer named haproxy, which breaks one TCP connection into two. In haproxy, we use round robin to balance the flows. We use iperf to generate traffic, whose payloads have a high probability of being the same. So the packets have no fingerprints to be distinguished from each other. We sort the time when connections are built at the client side and server side, i.e., the 1st ACK packet from the client to the load balancer and the 1st SYN from the load balancer to the server, and correlate inbound and outbound flows by the time sequence. We start 4000 iperf client flows to 10 iperf servers via haproxy; the connections are set up by the haproxy as soon as possible, which takes 12 seconds. We use haproxy logs to check the accuracy. We find that with the load of 330 connections per second in haproxy we can achieve 100% accuracy on flow correlation. This is the fastest rate for a haproxy in our VM to build connections. The result reveals that it is feasible to use time sequence to correlate flows and VND provides flexible APIs to correlate flows for a layer-4 load balancer.

3.6.1.3 Bottlenecked Middlebox Detection

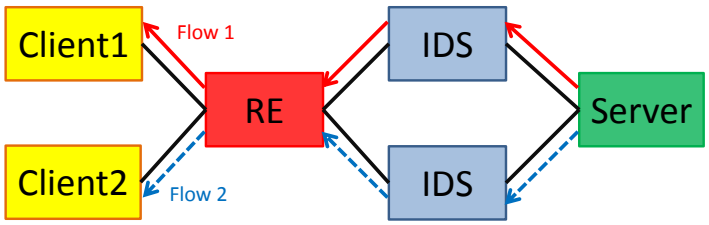
In virtual networks, middleboxes are usually used by a cloud provider to achieve better network utilization or security. In the cloud, middleboxes are also provided to the tenants as services [62] for their virtual networks. In these cases, the tenant does not have direct access to the middlebox, which makes its diagnosis difficult. In a virtual topology with multiple middleboxes, especially when the middleboxes form a chain, if a large amount traffic traverses the middlebox chain, one of the middleboxes may become a bottleneck. The bottlenecked middlebox needs scaling up. However, there is no general way to determine the bottlenecked middlebox.

One solution is to try to scale each middlebox to see whether there is performance improvement at the application [36]. But this solution needs the application's support and is not

prompt enough. Another solution is to monitor VM (we assume a tenant is using a software middlebox in the VM) resource usage, which is still not feasible due to middlebox heterogeneity [38]. Also some resources, such as memory throughput, is hard to measure. Network administrators can also check middlebox logs to find out problems. However, this requires too much effort to become familiar with various middleboxes, moreover, the problem may be in the OS kernel.



(a) Logical chain topology with middleboxes

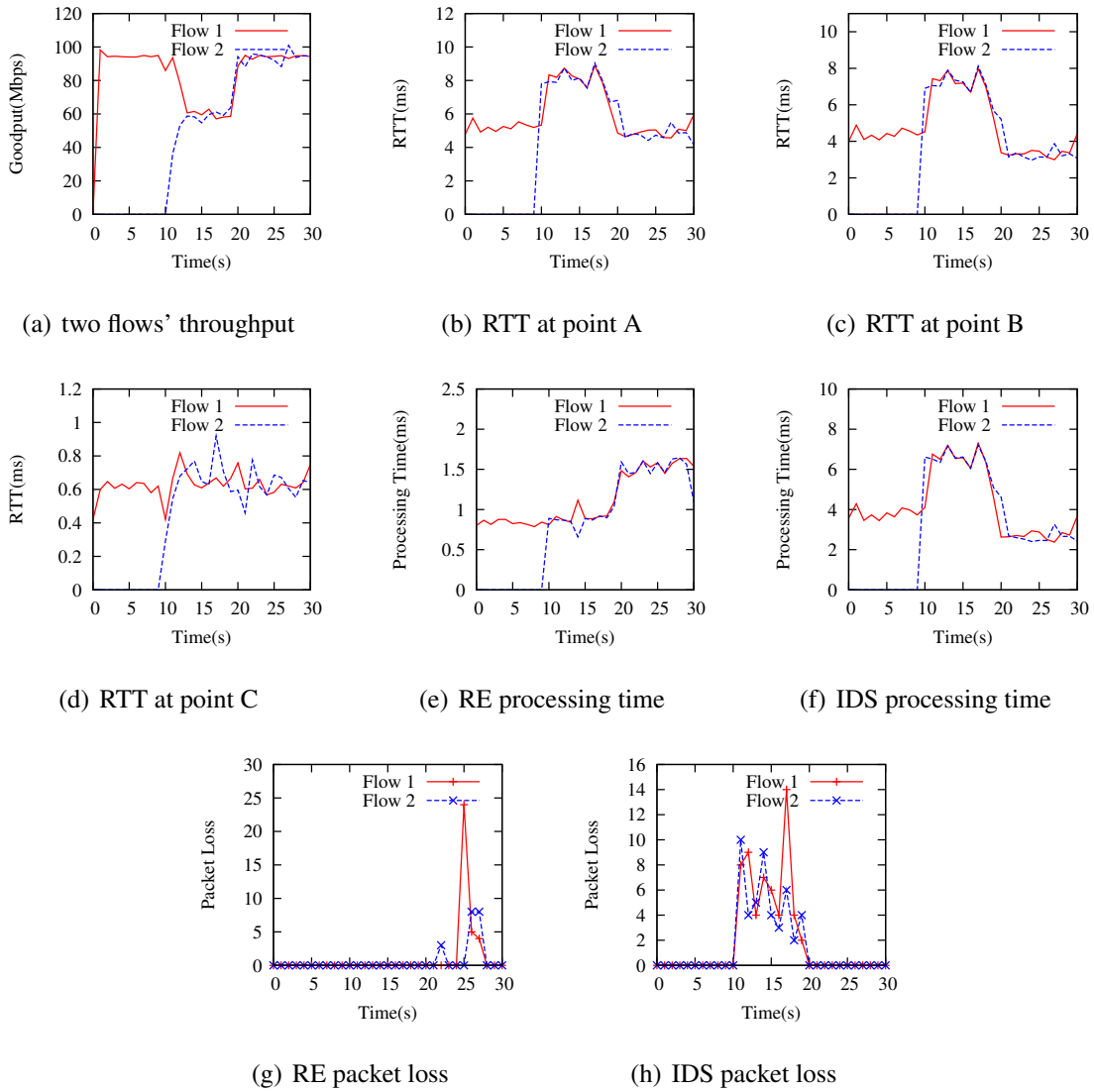


(b) Actual topology after scaling

Figure 3.15: Chain topology with middlebox scaling

Here we use VND to diagnose the bottleneck. We assume a flow from a client to the server traverses a middlebox chain with a Redundancy Elimination (RE)[23] and an Intrusion Detection System (IDS)[16]. In the case that traffic volume increases, one of the two middleboxes becomes the bottleneck and requires scale up.

At first, a client fetches data from the server at the rate of about 100Mbps. Then at the 10th second, a second client also connects to the server and starts to receive data from the server. Then client 1's throughput drops to about 60Mbps, and client 2's throughput is also about 60Mbps (Figure 3.16(a)). To find the bottleneck of the whole chain topology,



(a) two flows' throughput (b) RTT at point A (c) RTT at point B

(d) RTT at point C (e) RE processing time (f) IDS processing time

(g) RE packet loss (h) IDS packet loss

Figure 3.16: Bottleneck middlebox locating

we use VND to deploy trace capture at points A, B and C in the topology. We capture all traffic with the server IP address. We start the diagnosis application in Section 3.4, and check the RTT at each point. Figure 3.16(b) 3.16(c) 3.16(d) show that at point A and B the RTT increases significantly when the second flow joins, and at point C the RTT does not change too much. We use $RTT_A - RTT_B$ as the processing time at the RE middlebox and $RTT_B - RTT_C$ as that of the IDS. It is obvious that when traffic increases, the processing time at the IDS increases by about 90% (Figure 3.16(e) 3.16(f)). We deploy the packet loss

diagnostic application to observe the packet loss at each hop. Figure 3.16(g) 3.16(h) indicate that when the second client joins, packet loss happens at the IDS and no packets are lost at the RE. These observations indicate that the IDS becomes the bottleneck of the whole chain. So the IDS should be scaled up as in Figure 3.16(b). Then we can see that the throughput of both flows increases to nearly 100Mbps, the delay at the IDS decreases back to 3ms, and there is no packet loss at the IDS. The RE middlebox has some packet loss, but it does not impact the application performance. The logical chain topology with middleboxes is thus successfully scaled.

3.6.2 Performance Benchmark

We evaluate VND's performance in our testbed. We measure the processing capacity of each component in VND and that of the diagnostic applications.

3.6.2.1 Trace Collection and Parsing

Table 3.1: Processing capacity of components

Components	Trace Collector	Trace Parser	Query Executor
Throughput (pkt/s)	68k	37k	84k

We first evaluate the processing capacity of each components in VND. We perform trace collection and parsing as fast as we can in our prototype. The result is shown in Table 3.1. The trace collector can dump 68k packets per seconds, the trace parser can process 37k packets per seconds [2] and the query executor can store 84k packets per second [3].

3.6.2.2 Trace Query

The trace query processing capacity depends on the complexity of query itself. We use the trace in the bottleneck middlebox detection experiment (Section 3.6.1.3). We monitor

[2]The trace parser is single thread.

[3]We optimize the query executor's storage by using ramdisk.

throughput, RTT and packet loss, and observe its overhead and performance in terms of response time. These three diagnostic applications represent different trace table operations: aggregation, single-table join and multi-table join.

In throughput and RTT monitoring, we check them periodically at different time granularity; we control the checking period to observe the overhead and performance. In packet loss monitoring, we sample packets in one table and search for it in another; we control the sample rate to observe the overhead and performance. The result is shown in Table 3.2, 3.3, and 3.4

Table 3.2: Throughput query

Period(s)	1	3	5	7	9
Execution(s)	0.03	0.1	0.16	0.22	0.29
Traffic(MB)	<0.1	<0.1	<0.1	<0.1	<0.1

Table 3.3: RTT query

Period(s)	1	3	5	7	9
Execution(s)	0.1	0.29	0.49	0.69	0.9
Traffic(MB)	<0.1	<0.1	<0.1	<0.1	<0.1

Table 3.4: Packet loss query

Samples/s	1E0	1E1	1E2	1E3	1E4
Execution(s)	<0.01	<0.01	0.01	0.03	0.2
Traffic(MB)	0.1	0.1	0.2	0.5	3.4

In throughput and RTT monitoring, the response time shows strong linear relations with the checking period. In throughput monitoring, one second's traffic volume of a 100Mbps flow can be processed in 0.03 second, so we predict that at most 3Gbps flow's throughput

can be monitored in real time. Similarly, at most 1Gbps flow's RTT can be monitored in real time.

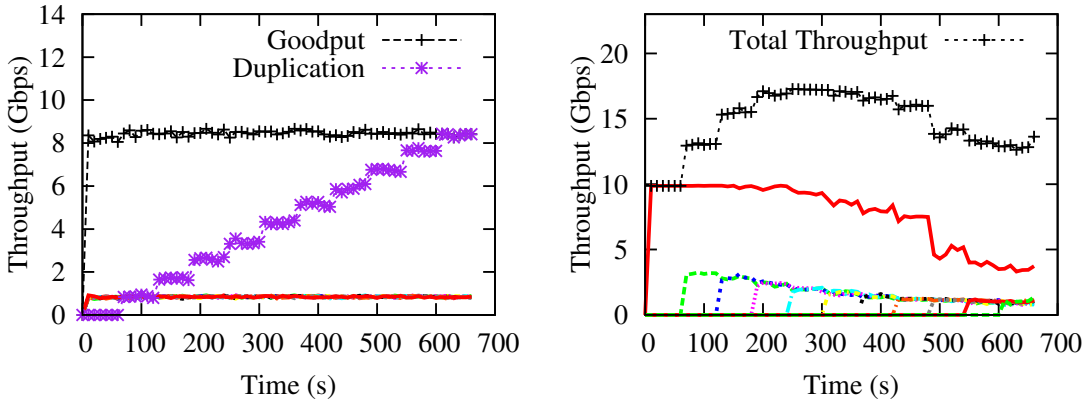
In the packet loss case, VND can process 10,000 records in 0.2 second. Each record costs a fixed amount of time; scaling this linearly, we predict that with 2-3 Gbps throughput, the packet loss can be detected in real time.

3.6.3 Overhead

We first evaluate whether trace collection impacts the existing system in terms of network overhead and memory overhead. We then measure whether trace query introduces obvious network and storage overhead.

3.6.3.1 Trace Collection

VND makes use of the extra processing capability of virtual switches, so that flow capture does not impact the existing tenant network traffic. However, it consumes memory I/O throughput on servers, so flow capture could possibly impact some I/O intensive applications with rapid memory access in virtual machines. We measure and model this overhead in our experiment.



(a) Flow capture (b) OVS capacity

*Curves without a legend are TCP flows

Figure 3.17: Network overhead of the trace collection

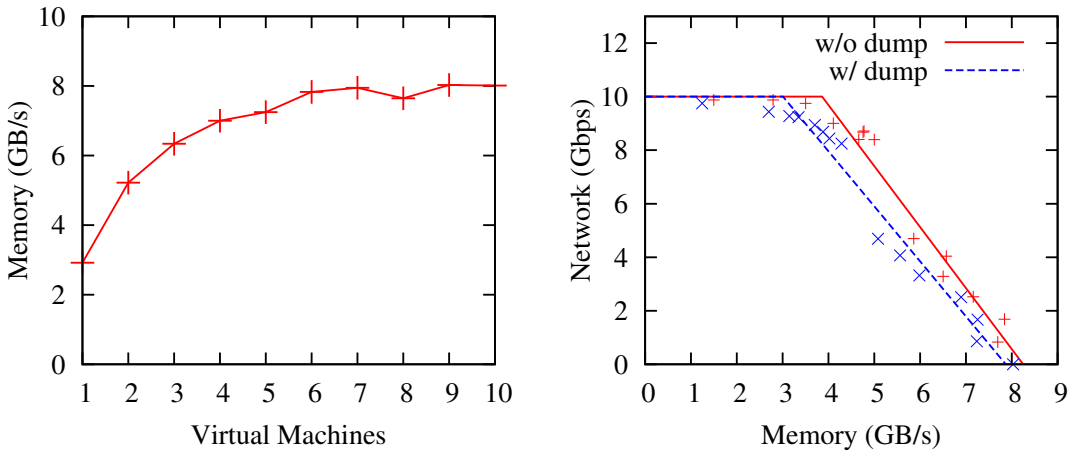
Network Overhead: With our optimization, trace duplication is performed by a virtual switch and the table server is set up locally. This introduces extra network traffic volume from the virtual switch to the table server. We evaluate whether this trace duplication impacts existing network flows.

We set up 8 virtual machines on 2 hypervisors, and start eight 1Gbps TCP flows between pairs of VMs running on the 2 hypervisors. We then use VND to capture one of them every minute. Figure 3.17(a) shows that when the flows are mirrored into table servers, the original flow's throughput is not impacted by the flow capture on OVS. The reason is that the total throughput of VM traffic is limited by the 10Gbps NIC capacity. However, the packet processing capacity of OVS is larger than 10Gbps, which makes it possible to perform extra flow replication even when OVS is forwarding high throughput flows.

We conduct an experiment to further understand the packet processing capacity of OVS. In Figure 3.17(b), we start a background flow between 2 hypervisors, which traverses OVS and saturates the 10Gbps NIC. Then we start one new TCP flow between two VMs on the same hypervisor every minute to measure the left-over processing capacity of OVS. The peak processing throughput of OVS is around 18Gbps. Thus there is a significant amount of packet processing capacity – up to 8Gbps – on OVS to perform local flow replication even when the 10Gbps NIC is saturated.

Memory Overhead: Another overhead concern is memory. The physical server can be more and more powerful with more CPU, larger memory and more peripheral devices. However, the computer architecture makes all internal data transfer go through the memory and the bus, which is a potential bottleneck for cloud servers with multiple VMs running various applications. VND surely takes some memory throughput to dump traces; we evaluate how much impact is introduced to the virtual machine memory access.

In Figure 3.18(a), we run linux mbw benchmark in virtual machines. In that benchmark, we allocate two 100MB memory spaces and call memcpy() to copy one to another. Results show that 1 VM can only make use of about 3GB/s memory bandwidth. As the number of VMs increases, the aggregated memory throughput reaches the upper bound, which is about 8GB/s.



(a) VM memory throughput

(b) Network-memory throughput

Figure 3.18: Memory overhead of the trace collection

We look into the influence of network traffic on memory throughput. We start 20 VMs on the hypervisor, 8 VMs run the memory benchmark, 6 VMs send network traffic by iperf out to another physical server, and 6 VMs are used to dump traces. We control the network throughput and aggregate the memory throughput. The network throughput is constrained by the physical NIC bandwidth, which is 10Gbps. When the network traffic does not saturate the physical NIC, the memory benchmark saturates the remaining memory bandwidth. We fit the memory-network throughput using linear regression. Figure 3.18(b) indicates the relationship between aggregate network throughput and aggregate memory throughput. The solid line is without flow capture; the dash line is when we dump all network traffic. We assume the network throughput is N Gbps, and the memory throughput is M GB/s. Without network traffic dump, the network-memory throughput is

$$N + 2.28M = 18.81,$$

and with network traffic dump, it is

$$N + 2.05M = 16.1$$

This result shows that each 1Gbps network traffic dump costs an extra 59 MB/s of memory throughput.

Memory bandwidth overhead introduced by VND is unavoidable. Our experiment quantifies the performance impact introduced by the VND trace collection on application memory throughput. We advise that the cloud administrator takes memory throughput into consideration when allocating VMs for the tenants.

3.6.3.2 Trace Query

Trace queries require data movement such that it consumes network bandwidth. The VND network traffic can be isolated from the tenant traffic by tunneling, and their bandwidth allocation can also be scheduled together with the tenant traffic by the cloud controller. We still use the same experiments in Section 3.6.2.2 to evaluate VND's overhead, and the results are still in Table 3.2 3.3 3.4.

Storage: At each hop, the total traffic volume is 0.5GB, so the total size of all traces is 1.5GB. After the traces are parsed and dumped into the database, the table storage costs only 10MB for tables and 10MB for logs. The storage for one diagnosis is not a big issue for current cloud storage, and this storage space can be released after the diagnosis.

Network: The result of the throughput and the RTT monitoring experiment in Table 3.2 and 3.3 show little network traffic, because a local data table operation does not cause any traffic and outputting the results generates negligible traffic. Inter-table operations need data movement, e.g., packet loss monitoring in our experiment. The overhead is easy to predict: it is the record size multiplied by the number of records to move in the execution period. Table 3.4 shows that with a 100Mbps flow, the extra traffic generated by packet loss detection is only a few Mbps at the rate of 10,000 samples per second.

3.6.4 Scalability

In this section, we discuss the scalability of the VND framework. In a large-scale cloud environment, the scalability challenge for the VND is to perform trace collection from a large number of VMs and support diagnosis requests for a large number of tenants.

Table servers are co-located with tenants' VMs or middlebox VMs and performs trace collection locally. As we measured in Section 3.6.3.1, tenants' network traffic is not impacted by trace collection, and only a little memory overhead is introduced. According to our performance benchmark, each table server can process about 37k pkt/s traffic (in Section 3.6.2.1), but the table server can be scaled up by assigning more resources (e.g., CPU) or set up more table server VM instances. So trace collection will not be the scalability bottleneck when there is a large number of VMs.

The *control server* generates trace collection policies and passes query commands and results between tenants and table servers. It is easy to add this logic to existing user-facing cloud control servers. Given that existing clouds, such as Amazon EC2 and Microsoft Azure, have been able to support a large number of tenants through web-based control servers, we believe the control server will not be a major scalability bottleneck either.

However, in a large-scale cloud, VND table servers will need to perform real time data processing and table queries for many tenants, which could become a major scalability bottleneck. Therefore, our following scalability discussion is focused on the query performance of table servers.

To evaluate the *table servers'* scalability, we perform simulation analysis based on the statistics of real cloud applications and our query performance measurements. In the simulation, we make the following assumptions:

- The data center network has full bisection bandwidth, so we simplify the physical network by one big switch connecting all physical servers. The physical NIC bandwidth is 10 Gbps.

- Typical enterprise cloud applications (e.g. interactive and batch multi-tiered applications) use 2 to 20 VMs [27]. We assume each application is running in one virtual network, so each virtual network has 2 to 20 VMs.

- The flow throughput between virtual machines follows a uniform distribution in [1, 100] Mbps [27].

- In each physical server, the virtual switch can process up to 18 Gbps network traffic (Section 3.6.3.1).

- Throughput query is common in the network diagnosis. This query is intensive because it needs to inspect all the packets. We assume each tenant issues a throughput query of all the traffic in its virtual network. Each executor can process query for 3 Gbps network traffic at real time (Section 3.6.3.2).

In the simulation, we first generate virtual networks whose size and flow characteristics following our assumptions, then allocate them (greedily to the server with most available resources) to a data center with 10,000 physical servers until the total link utilization reaches a threshold. Then the tenants start to issue diagnostic requests. Each diagnostic request is capturing and querying all the traffic in the tenant's virtual network. If there are enough resources left (trace duplication capability in the virtual switch and query processing capability in the query executor), the tenant's request consumes its physical resources and succeeds; otherwise, the request is rejected and fails. As more and more requests are being issued, there are fewer and fewer resources left for the following diagnostic requests. We stop issuing diagnostic requests when requests start to be rejected. Then we calculate the portion of virtual networks that is successfully diagnosed over the total virtual networks allocated.

Trace Collection: When total link utilization is under 80% (less than 150K tenants), all the virtual network traffic can be captured. Even if the total link utilization is 90% (162K tenants), 98.6% virtual networks can be diagnosed. Given that in a typical data center network, the utilization of 80% links are lower than 10% and 99% links are under 40% utilization [26], we conclude that in a common case (total link utilization is lower than 30%) all virtual network traffic can be captured without impacting existing application traffic.

Trace Query: In Table 3.5, when total link utilization is under 30% (54K tenants), almost all queries succeed. When total link utilization is high, the product of the link utilization and the success query ratio is about 30%, that is, 30% of the total link capacity can be queried at "real time" successfully. If some tenants relax the latency requirement of queries and

do offline data processing, the VND query can make even better use of the spare resources without contending with latency sensitive queries.

Table 3.5: Successful queries in a data center

Tenants Count	18K	54K	90K	126K	162K
Link Utilization(%)	10	30	50	70	90
Successful Query(%)	100	97.85	58.7	41.9	32.5

3.7 Limitations and Opportunities

VND is not a panacea for all application plane problems. There are a number of issues that cannot be solved by VND. Some examples of cases that cannot be solved by VND are provided below, along with discussions of opportunities to use and improve VND.

- **Clock synchronization among multiple capture points.** During one diagnosis, there may be multiple capture points. When a packet is captured, its time stamp is the time when it arrives at the collector. If the collectors are not synchronized, the time stamps of traces from different capture points will not be synchronized, either. In this case, some diagnostic applications may not give accurate results, e.g., processing delay between hops. We argue that using Network Time Protocol (NTP), we can synchronize the clocks of collectors with an error less than 1ms. Our application examples are operated at the time granularity of 1s; therefore the error introduced to the final result is negligible. In addition, we can design the diagnostic application smarter so that it eliminates time difference errors. For example, we can compute the RTT of two points separately and compute the difference of the RTTs as the processing delay between two hops; thus we avoid computing difference between time stamps of two traces.

- **Completeness.** Diagnostic applications cannot cover all problems in virtual networks. For a recent finding, network problems may be transient, which means its cause or

symptoms may change their locations with time. All our diagnostic/monitoring applications are designed to check a snapshot of the network, so we need a new approach to diagnosing transient network problems.

- **Performance Guarantee.** VND's overhead is not small. We measured its overhead by experiments. In the actual deployment, if the overhead is expected to affect the existing system, that diagnostic should be rejected.

- **Service Provision.** VND's diagnostic interfaces includes packet traces and operations, so tenants will have enough flexibility to perform their own diagnosis. However, in some cases, this provision may harm the cloud security. For example, it is possible that a tenant will capture traces before and after a middlebox, and then reverse engineers the logic inside the middlebox. In this case, the cloud provider should limit the operations that can be performed by tenants.

In future work on VND, one direction would be to provide more intelligent diagnostic applications, since VND already builds a powerful infrastructure and interfaces to support these applications. Another direction would be to reduce VND's overhead in order to improve its performance. When integrating VND with current cloud solutions, security should be carefully considered to prevent malicious users from taking advantage of VND.

3.8 Summary

In this chapter, we identify the virtual network diagnosis problem and articulate the challenges involved. We propose VND to overcome these challenges, which is a framework that allows a cloud provider to offer a sophisticated virtual network diagnosis service to its tenants. Our evaluation shows that by co-locating flow capture points and table servers, VND can capture tenant's traffic flows without impacting their performance, and the network diagnosis query can be executed quickly on distributed tables in response to tenants' requests without introducing too much extra network traffic. This architecture scales to the size of a

real data center network. To the best of our knowledge, ours is the first attempt at addressing the virtual network diagnosis problem, and VND is a feasible and useful solution.

Chapter 4

Performance Diagnosis in Software Data Planes

Data plane diagnostic tools are invaluable toward managing and troubleshooting networks. Tools such as ping and traceroute, and frameworks such as NetFlow [13] and sFlow [20], are routinely used by network operators both to understand whether the network is functioning as expected, and, if not, understand what may be causing the underlying problem.

However, in recent years, network data planes have changed in fundamental ways. In addition to simple L2 and L3 devices, they are increasingly composed of a wide range of network functions, or middleboxes, that perform custom packet processing to aid in satisfying various network-wide objectives pertaining to performance, security, and compliance; examples include firewalls, load balancers, application gateways, accelerators, etc. With the advent of software switching, and more importantly, network functions virtualization (NFV), traditional hardware switching elements and middleboxes are being realized using software running on generic compute platforms (e.g., a virtual machine, or VM).

Thus, the "data plane" that packets traverse on end-to-end paths now includes—in addition to hardware L2/L3 devices and links—a variety of software components that reside within compute servers' virtualization stacks and within the VMs running various middlebox software. Examples include physical and virtual NICs and their drivers, various packet processing routines in hypervisors and within middlebox logic, virtual switches, hypervisor I/O handlers, host and guest network stacks, etc. We refer to this new portion of the data plane as the *software data plane*.

Our community has developed a variety of innovative tools and frameworks for diagnosing problems in hardware data planes. Examples include traceroute, path MTU discovery [15], available capacity detection [61], tomography [28], etc. Unfortunately, we don't have similar tools for software data planes.

In fact, software data planes present new challenges to diagnosis. Because they span a variety of software components running on shared compute resources, where each component can perform fairly sophisticated actions, software data planes are much more susceptible to a range of *subtle performance problems*. We argue that there are at least three classes of performance problems—those arising due to mis-allocation of resources to software data plane elements, contention amongst elements for shared resources, and buggy design/implementation. Such problems either don't arise frequently in traditional hardware data planes (e.g., implementations with performance bugs are rare), or they are simpler to diagnose because only a handful of resources are allocated (e.g., bandwidth and router buffering, vs. CPU, disk, memory, network etc. in software data planes) and contention observed manifests at a small number of locations (e.g., buffers building up or link utilization growing vs. drops/buffering at a multitude of possible locations in the virtualization stack—See Section 4.1.1). Furthermore, because of stateful packet processing, problems arising in one middlebox may quickly propagate up- or down-stream to other middleboxes on an end-to-end path, which complicates accurate diagnosis of root causes (See Section 4.1.2).

We design a general system, called PerfSight, for accurate and quick diagnosis of a broad variety of performance problems that may arise in current and future software data planes. Our approach is rooted in viewing the software data plane as a *pipeline of elements*, where an element is a logical unit that reads traffic from or writes traffic to another by buffers or function calls. This abstraction captures a variety of entities on the software data path, including middlebox logic, routines in the hypervisor, and routines in a VM's network stack. We decouple the problem of statistics collection from diagnosis. We identify elements and their input/output methods, and statically analyze their code paths to determine where packets can be buffered or dropped; we then instrument all such locations to collect a suite of statistics in a light-weight fashion. When queried by a controller at run-time, these statistics

are returned in a generic format, and then consumed by diagnostic applications to perform interesting analytics.

We develop two novel applications that aid in diagnosing key software data plane problems. At a high level, these diagnostic applications analyze the gathered element statistics in different "dimensions": e.g., across all VMs on a single physical machine, or across a collection of middlebox VMs that are chained together. We show that by identifying the exact locations in the software data plane where the performance problems are arising—i.e., which particular buffer or element in a VM, hypervisor or the kernel is facing a problem—we can detect contention across VMs vs. problems arising due to resource limitations within a single VM. Similarly, by studying the nature of the performance problem faced by a middlebox—e.g., whether it is stalled for read vs. writes—and in what status its neighboring middleboxes are (i.e., whether they are also stalled not not), we can quickly narrow down the root cause middlebox(es) whose performance problems have propagated throughout a chain.

We have built a prototype of PerfSight in an environment running a Linux 3.2 kernel, Open vSwitch [19], and QEMU /KVM [25]. We conduct several experiments with this prototype on a small scale experimental testbed, using topologies representing chains of middleboxes, and a variety of workloads.

We study the effectiveness of PerfSight and find that in all cases PerfSight's low level instrumentation provides accurate information about the exact location in the software data plane of an observed performance problem. We also find that PerfSight can identify contention for a variety of different shared resources (e.g., memory or network bandwidth, CPU, or NIC capacity), it can accurately detect bottleneck middleboxes irrespective of the specific resource that was under-provisioned, and it can accurately pin-point the root cause of problems in complex multi-chain settings where problems can arbitrarily propagate. We also illustrate how an operator can use PerfSight to implement cross-tenant workload management and elastic scaling in a multi-tenant set-up. Finally, we show that PerfSight's instrumentation imposes an insignificant overhead ($< 1\%$).

The remainder of this chapter is organized as follows. In Section 4.1, we describe the performance problems of software data planes and motivate the need for a better diagnosis

framework. In Section 4.2, we sketch the high level architecture of PerfSight. Then, in Section 4.3 and Section 4.4, we present the design of PerfSight's two major tasks— statistics gathering and diagnostic applications. Section 4.5 describes the implementation details. We present experimental results showing the accuracy and overhead of PerfSight in Section 4.6. We discuss the scope of PerfSight in Section 4.7, and conclude this chapter in Section 4.8.

4.1 Background and Motivation

In this section, we first present software data plane definition and the performance problems associated with it. Then we talk about the challenges in accurate software data plane diagnosis and the need for a better diagnosis approach.

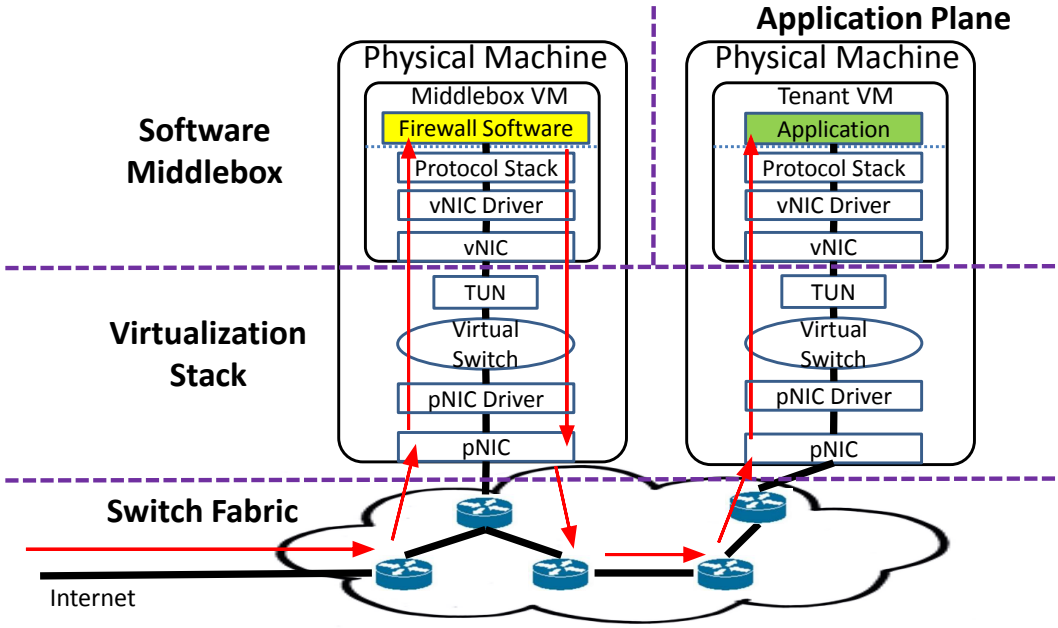
4.1.1 Software Data Plane

We use the same example as in Section 1.1. In Figure 4.1, the tenant traffic flow traverses the physical NIC driver, virtual switch, hypervisor IO handler, virtual NIC driver, and network stack in guest OS. Whereas traditional data planes consisted just of hardware switching elements and network links connecting network end points, the advent of NFV means that we need to rethink what constitutes the data plane. In particular, it now also includes the software components shown above that are traversed within middlebox VMs. We refer to this portion of the data plane as the *software data plane*.

The software data plane is composed of a variety of *elements*, each of which performs a certain logical function. In Figure 4.1, each blue rectangle or oval is an element. The software data plane can thus be viewed as a pipeline of elements with data traversing from one to the next. The output of one element is the input to its successor. Neighboring elements exchange messages via buffers or function calls.

Elements can be further divided into two categories: (a) Those belonging to the *virtualization stack*; such elements are *shared* by multiple VMs, and examples include the pNIC driver, the packet processing routine, virtual switches and the hypervisor I/O handler. (b)

(a) An example of a virtual network



(b) Deployment in the data plane

Figure 4.1: Data plane organization

And those belonging to the *software middlebox*; such elements are confined to one middlebox VM, and examples include the vNIC, vNIC driver, VM guest OS network stack and middlebox software. In a multi-tenant setting, software data planes belonging to different tenants may "overlap" on one or more physical machines.

4.1.2 Performance Problems

In what follows, we argue that software data plane performance problems can arise due to at least three underlying reasons. Furthermore, addressing the problem requires a different approach in each case. A diagnostic approach therefore needs to carefully delineate and accurately identify the root cause.

When the offered load on a middlebox exceeds the capacity allocated to it (along some resource dimensions), the traversing flows' throughput/latency will be **bottlenecked**. Because most middleboxes perform complex actions, such bottlenecks may arise not just due to increased traffic volume but also due to sudden, unexpected changes in the traffic profile. Addressing bottlenecks is up to the tenant (e.g., the tenant can redeploy the middlebox in a "larger" VM).

If multiple elements contend for a shared resource, and their requirements exceed the available resource capacity, all involved elements cannot achieve their expected performance. Such **contention** usually happens in the virtualization stack, because all tenant VMs and middlebox VMs in one physical machine share the same datapath in the virtualization stack. What make things worse is that the shared resource may not be explicitly allocated. For example, it is hard to allocate memory bandwidth to individual VMs; shared buffers in the virtualization stack are similarly not allocated either. To address this, impacted middlebox VMs may have to be migrated to locations with less contention. Often, this requires the cloud operator's involvement.

Design and implementation defects that lead to inefficient computation widely exist in software, and thus software data planes are naturally impacted by such **performance bugs**. Indeed, it has been shown previously [59] that middlebox software bugs can result in "soft failures" (e.g., a significant drop in throughput when middlebox software is "upgraded"). To address such performance bugs, a tenant must reload their VMs with a suitable version of software.

4.1.3 Challenges of Accurate Diagnosis

After a tenant experiences performance problems and submits trouble tickets, we assume that the tenant submits a ticket to its data center operator. Because there are many different kinds of middleboxes and a multitude of elements in software data planes, multiple middleboxes in a virtual cluster can often be operated in a sequence (or "chained"), and multiple tenants' clusters can overlap at arbitrary physical machines; accurately diagnosing data plane performance problems and identifying root causes is not easy for the operator.

A common approach to detect bottlenecks is to monitor the resource utilization on VMs [5]. While this may work in some cases, there are a variety of middleboxes for which resource utilization does not reflect workload intensity. For example, a video stream transcoder [39] may employ non-blocking I/O instead of blocking I/O to avoid context switching. For this middlebox, CPU utilization is always 100%, but we lack a way of distinguishing the portion of CPU cycles spent on processing vs. busy waiting. Another alternative—monitoring traffic volume changes—is also insufficient as bottlenecks may arise due to changes in traffic profiles (e.g., when a middlebox encounters a traffic profile that is different from the one it was optimized for).

Contention is similarly hard to diagnose. Some OS statistics such as packet drops at the pNIC can determine certain forms of contention (e.g., for network bandwidth), but such statistics are not always available today for other key resources. For instance, when multiple VMs on a machine are performing heavy memory copies, they contend with each other over the shared memory bus, which is hard to diagnose due to lack of suitable statistics. Another reason why contention is hard to diagnose is that its effects may only be indirectly felt. For instance, a VM trying to saturate a certain amount of network capacity may be unable to do so in the presence of competing memory intensive workloads, because the two sets of workloads contend for the memory bus (an empirical example is shown in Figure 4.2). Simply monitoring memory utilization or network packet drops does not reveal the root cause of contention.

The final factor that complicates diagnosis is that performance problems in a middlebox—arising due to, e.g., bugs—may *propagate* across a virtual cluster due to the chaining of middleboxes. The upshot is that the wrong middlebox may be identified as being the root cause of poor performance in a chain or cluster, and incorrect/inefficient counter-measures may be adopted. In Section 4.6 we will show such an example—a load balancer, a content filter and an HTTP server form a chain and the content filter writes logs to a shared NFS server (see Figure 4.12). When the NFS server has a bug, performance degrades in the whole chain. In this case, it is challenging to find out the root cause middlebox.

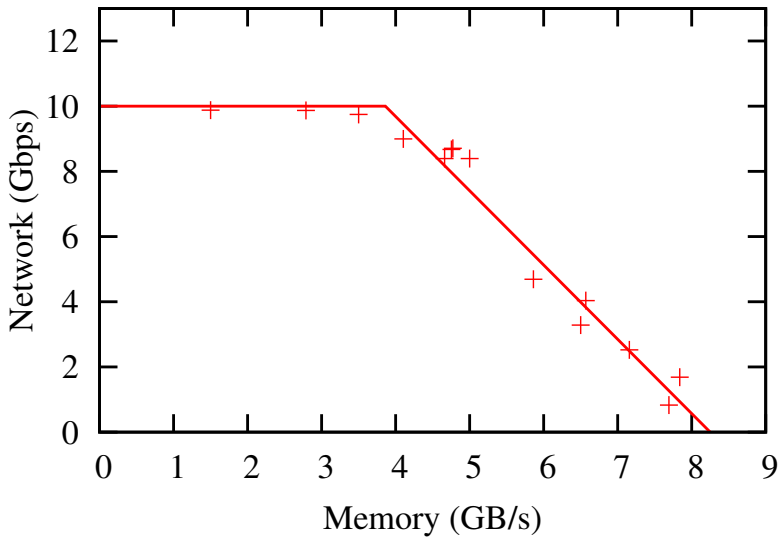


Figure 4.2: There are 8 VMs in a 8-core hypervisor with a 10Gbps NIC. Some of the VMs perform intensive memory copy operations, and the others send traffic to another machine by best effort. We vary the memory copy workload and measure the total throughput of the memory and the network respectively in each case. When memory throughput is low, the NIC capacity is fully saturated (10Gbps). However, when the memory throughput exceeds a threshold, every 1 GB/s increase of memory throughput causes 439 Mbps decrease of network throughput.

4.1.4 Need for a New Approach

Existing troubleshooting techniques such as Anteater, HSA, NICE, Libra, etc. [56, 46, 29, 67] focus on the correctness of the network; thus, they cannot be used to detect performance problems. Tools such as ping and traceroute provide end-to-end tenant-level information which cannot pin-point root causes (e.g., those due to contention) accurately. Other diagnostic frameworks, such as NDB and VND [40, 64], collect traces on the datapath, either at network switches (NDB), or at vNICs or pNICs (VND). This can impact performance (of multiple tenants' virtual clusters) significantly. But, more importantly, such traces don't provide enough information to perform accurate root cause diagnosis; e.g., they cannot determine that packets were dropped due to memory contention.

Thus, we argue for a ground-up comprehensive framework for software data plane diagnosis. We posit that given the complexity of the software data plane, a framework that performs *low-level, broad instrumentation* of various elements in the software data plane, coupled with *suitable analytics*, can provide accurate and useful diagnoses.

4.2 PerfSight Overview

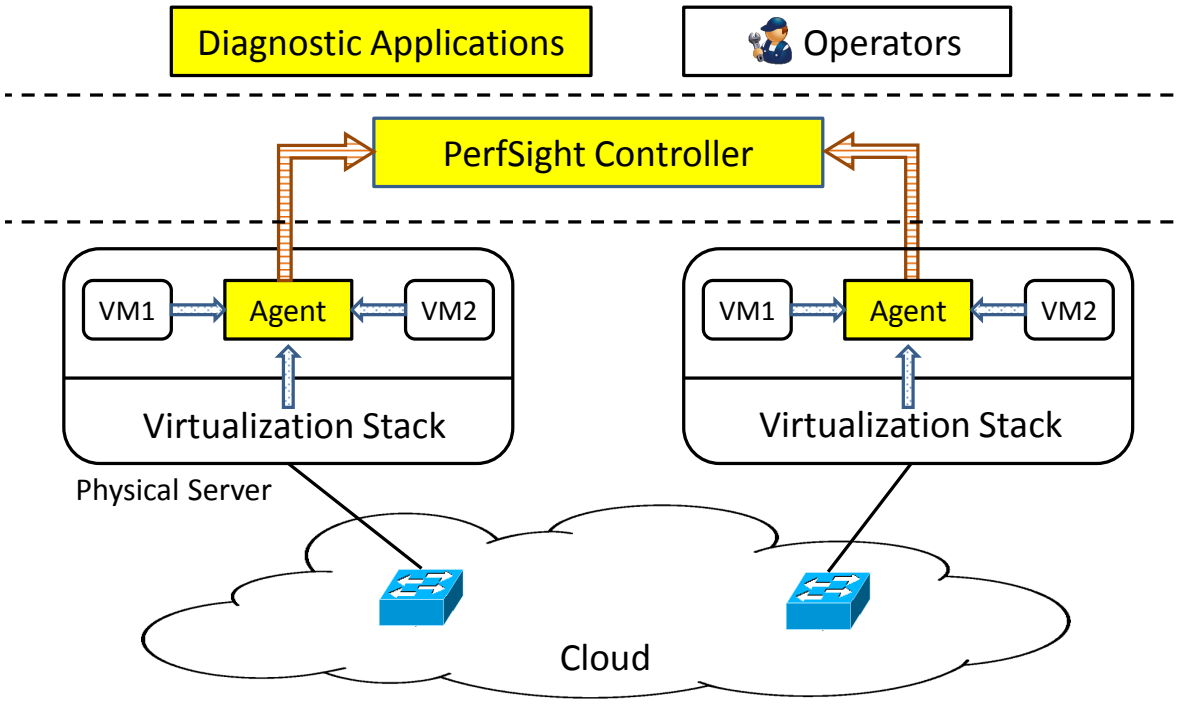


Figure 4.3: PerfSight architecture

We design and implement a system for software data plane diagnosis called PerfSight. Figure 4.3 shows the architecture of PerfSight. It has 3 high level components: an agent running on each physical server, a central controller, and a set of diagnostic applications atop the controller.

Diagnostic applications are developed using the controller-operator interfaces. These applications query the software data plane for various statistics; the controller then translates these to queries issued to agents running on the corresponding physical servers. The agents

interrogate the relevant elements for statistics, and report the gathered statistics back to the application.

PerfSight uses a simple unified low-level interface for recording/retrieving statistics of the software data plane elements. This abstracts the complexity arising from the diversity of the data plane elements, and simplifies the task of designing analytics engines. The interface can be extended so that operators can easily enrich the set of statistics gathered.

PerfSight decouples data collection from analytics so that each can be improved or replaced by more advanced techniques. PerfSight's analytics applications work by performing *aggregate* diagnostics on the gathered statistics; more specifically, these applications jointly analyze data gathered across multiple instances of a middlebox, multiple middlebox VMs deployed on a single server, or multiple middleboxes deployed in one or more virtual clusters, to accurately pin-point the root cause of the observed performance problem.

In what follows, we first describe what statistics we collect and how they are collected. Then, we describe how these statistics are analyzed to obtain insights into software data plane performance problems.

4.3 Statistics Gathering

The key insight we leverage to perform software data plane diagnosis is to view the data plane as a pipeline of low-level *elements*. By instrumenting at the element-level, we can obtain fine-grained statistics at key points in the software data plane. These then form the basis for interesting and accurate diagnostic applications. In what follows, we describe how we obtain element-level statistics on throughput, packet drops, and packet size. The key challenge lies in understanding where and how to instrument elements such that the data collection overhead is minimal.

4.3.1 Element Abstraction and Statistics

Figure 4.4 shows the elements and buffers in a virtualization environment that uses a Linux kernel, QEMU/KVM, and Open vSwitch (OVS). Each element receives packets from

its predecessor, processes them based on internal logic, and delivers them to its successor. The exchange of packets between elements is often implemented by a buffer between them or by function calls. For example, in Figure 4.4, the pNIC driver and the packet processing routine (e.g., the NAPI routine in Linux) use the physical CPU (pCPU) backlog (a buffer) to exchange messages, whereas the NAPI routine and virtual switch use function calls.

Each element has *input methods* and *output methods*. For example, in the pNIC driver, the interrupt handler reads from the pNIC (or DMA mapped memory) and enqueues packets to the CPU backlog queue; the NAPI routine dequeues packets from the CPU backlog and calls the virtual switch frame handling function; the virtual switch frame handling function writes packets to a TAP's socket which is associated with a VM; and, the hypervisor I/O handler reads packets from the TAP and writes packets to the associated vNIC.

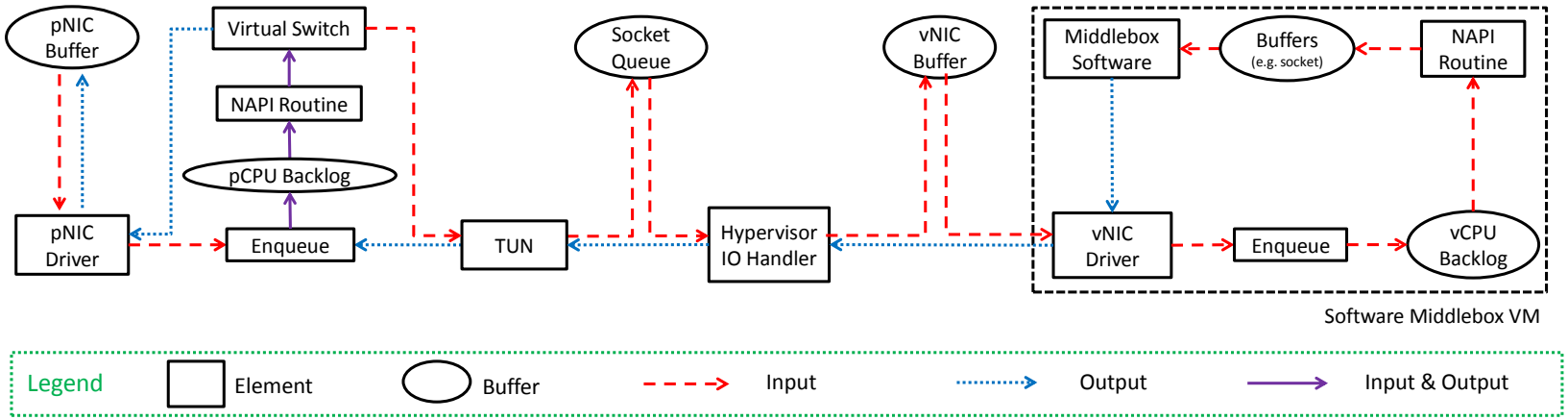


Figure 4.4: Elements in the software data plane (QEMU/KVM, Open vSwitch, Linux)

In each element, by analyzing its code, we can determine the code path that a packet traverses from input to output and possible code branches that might drop it. Statistics—e.g., how many bytes/packets were received/sent/dropped—of the traffic on the datapath between an element's input and output methods can be then recorded by instrumenting the code path and relevant branches. Another key statistic we gather is I/O time. This records the time spent on an I/O method (read and write). This can also be obtained in an easy and light-weight fashion by comparing the timestamps before and after the read/write function in an element. I/O time can reveal whether an element is facing starvation, and it plays a particularly crucial role in the diagnosis of propagation issues, as we show in Section 4.4.

In our current implementation, we perform the instrumentation task manually and exhaustively, but we believe it can be automated using program analysis. Interestingly, we find that many useful statistics actually don't require special instrumentation as they can be readily gathered, and hence statistics gathering imposes low overhead in this case. This is particularly true for elements residing in the host OS kernel. For example, packet count and byte count are basic statistics that already exist in most kernel elements. For other elements, e.g., those in the hypervisor and those inside middlebox software, we perform the instrumentation as indicated above; our extensive evaluation (Section 4.6) shows that the overhead is very low.

Our current prototype of PerfSight extracts/implements the three types of counters in each element: a packet counter, a byte counter, and an I/O time counter. The packet and byte counters are used by all types of elements while I/O time counter is only used for elements that interact with buffers. The counters accumulate as packets are processed.

Aggregate statistics such the instantaneous/average packet drop rates, throughput, and packet size can then be easily derived per element from these statistics. Operators can implement more complicated statistics at an element such as packet size distribution tracking if they can accept the resulting performance impact.

4.3.2 Agent

An agent in each physical server gathers element statistics. To reduce overhead, the agent pulls counter values from elements only when required. Due to the diversity of elements, especially across the kernel, hypervisor and the middlebox software, the element/agent interfaces are tailor-designed for each type of element. In particular, we use custom APIs for elements in the kernel: e.g., in the NIC driver and the TAP, the counters are in net_device, and they can be accessed by reading the corresponding device file in the file system. When the NAPI routine processes packets in CPU backlogs, statistics are stored in the data structure softnet_data, and they can be accessed via /proc in the file system. In virtual switches, each switch rule has its own statistics and can be fetched by virtual switch control channels. For elements in the hypervisor and middleboxes, we design a common generic element-agent API.

After fetching statistics from elements, the agent provides statistics to the controller in a simple, unified format, which is as follows:

```
<TimeStamp, Element, (attr1, value1),
(attr2, value2), (attr2, value3)...>
```

That is, the response of a query to an element returns the timestamp, element ID, and a list of <attribute, value> pairs, where each pair describes a counter and its value at that timestamp. For example, a NIC driver's statistics can be described as

```
<t1, eth0, ("Rx bytes", v1), ("Tx bytes", v2), ... >
```

This interface is generic and simple for infrastructure operators to extend the statistics. When an operator needs to add a customized counter, the operator would need to add the counter into related elements, add logic in the agent to fetch this counter and add the result to the message that is returned to the controller.

4.3.3 Controller

The PerfSight controller delivers statistics requests and responses between operators and agents. When the operator requests a virtual cluster's information, the PerfSight controller

obtains the physical location of the related virtual elements and sends requests to the agents of those elements.

Through controller/operator interfaces, the operator can query the attributes of each element. To retrieve the attribute of an element in a virtual cluster, the PerfSight controller first finds the physical location of the element, i.e.,
vNet[tenantID].elem[elementID], and then sends a request for attributes to the element's agent, and finally receives the requested attributes' values.

This interface can be used by an operator to perform basic monitoring of the performance of specific portions of its infrastructure. Figure 4.5 provides examples of basic utility routines for monitoring element states. As we argue next, interesting diagnostic applications can be built on these statistics.

4.4 Performance Diagnosis

The most interesting aspect of PerfSight is the set of diagnostic applications it enables. In what follows, we show how the statistics gathered in the aforementioned fashion can be used to develop algorithms that help resolve the performance problems central to software data planes that we mentioned in Section 4.1.

4.4.1 Detecting Contention and Bottleneck Middleboxes

Before we describe our algorithms, we give a few comments on how contention and bottlenecks manifest, and what makes detection hard in order.

Contention: VMs on the same physical server can contend for hardware or software resources in the virtualization stack. Returning to the example situation outlined in Sections 4.1.2 and 4.1.3, memory bandwidth was the shared hardware resource in contention; in yet other situations, hardware resources such as CPU and the NIC may be in contention. Likewise, in Figure 4.4, the shared datapath of input traffic and output traffic (purple/solid arrows) can be the software resource in contention; similar contention can arise for other software resources such as shared buffers/queues.

```
function GETATTR(tenantID,elementID,attributes)
    return vNet[tenantID].elem[elementID].attr[attributes]
end function
function GETTHROUGHPUT(tid, eid)
    attr ← [“time”, “bytes”]
    <t1, b1> ← GETATTR(tid, eid, attr)
    sleep(T)
    <t2, b2> ← GETATTR(tid, eid, attr)
    return (b2 − b1) / (t2 − t1)
end function
function GETPKTLOSS(tid, eid)
    attr ← [“inPkts”, “outPkts”]
    < b1_i, b1_o > ←GETATTR(tid, eid, attr)
    sleep(T)
    < b2_i, b2_o > ←GETATTR(tid, eid, attr)
    return (b2_i − b2_o) − (b1_i − b1_o)
end function
function GETAVGPKTSIZE(tid, eid)
    attr ← [“bytes”, “pktCount”]
    <b1, c1> ← GETATTR(tid, eid, attr)
    sleep(T)
    <b2, c2> ← GETATTR(tid, eid, attr)
    return (b2 − b1) / (c2 − c1)
end function
```

Figure 4.5: Basic utility routines

However, not all contentions display explicit symptoms. Among hardware resources, high CPU, memory and network utilization are explicit symptoms of contention, but memory bandwidth contention does not have an explicit manifestation. Whether symptoms of

contention for software resources can be observed depends on the software implementation. For example, OVS provides QoS and statistics of switch rules, which can be used to identify contention directly, but not all elements in the virtualization stack have appropriate instrumentation to enable such direct identification. Thus, on the whole, resource contention in the virtualization stack is difficult to detect.

Bottleneck Middlebox: A bottleneck middlebox is constrained by the amount of resources allocated to it, but, as discussed in Sections 4.1.2 and 4.1.3, whether a middlebox is a bottleneck cannot be judged solely based on its resource utilization.

Accurate Detection: The main idea underlying our diagnostic application here is as follows: Elements in the virtualization stack deliver packets to each other via intermediate buffers or function calls, and they typically use *nonblocking I/O* in doing so. That is, if an element cannot write to its successor, or its target buffer is full, packets get dropped. Thus, we obtain the *packet loss of each element* in the software data plane, and use it to locate where VMs are contending for resources or whether a VM is under-provisioned along some resource.

The application is designed as shown in Algorithm 1. For each element in the virtualization stack, we use the utility routing GetPktLoss() to obtain its packet loss; we sort all elements by their packet loss. Finally, the element with most packet loss is returned. This application monitors all related elements in the virtual network, so its cost is proportional to the size of the virtual network.

Crucially, the location of packet loss reveals the possible resources that VMs are contending for or are in shortage of. To aid in this, we build a simple offline *rule book* that maps packet loss locations to specific resources that may be running low or facing contention.

We construct the rule book as follows: we set up a variety of experiments where VMs contend for different resources, and we exhaustively track possible packet loss locations (details are in Section 4.6.1). The result is summarized in Table 4.1. Some symptoms reveal the resource in contention directly; for example, if incoming traffic exceeds pNIC capacity, packets are dropped at the pNIC.

Algorithm 1 Detect bottleneck and contention

```
1: function FINDCONTENTIONANDMIDDLEBOX()
2:     elements ← { e | e ∈ virtualization stack }
3:     elemLoss← ∅
4:     for e ∈ elements do
5:         loss = GETPACKETLOSS(e)
6:         elemLoss.add(<e, loss>)
7:     end for
8:     return SORTBYLOSS(elemLoss)
9: end function
```

In some cases, different kinds of contentions can have the same symptoms making detection hard: e.g., contention on CPU and memory bandwidth both lead to VMs being unable to fetch packets from TUN to vNIC, causing TUNs to drop packets which is the only externally visible symptom. In such cases, the operator can combine this with other symptoms such as CPU utilization and NIC throughput to distinguish the specific root cause.

Bottleneck middlebox detection is similar: when a tenant's deployment is facing end-to-end performance problems (and the tenant complains about it), the operator first selects middleboxes with high resource utilization and includes them in a "suspicious" set; in the degenerate case (e.g., when no high utilization is apparent) all of the tenants middleboxes could be included in this set. Then, we use our light-weight statistics to distinguish those middleboxes that are facing legitimate issues, such as packet drops, against those whose resources naturally run at a high utilization but are otherwise not bottlenecks (e.g., a video encoder). Thus, we can more accurately pin-point bottleneck middleboxes.

Contention and bottleneck can be distinguished based on whether loss is spread across multiple VMs (contention) or confined to on VM's software data path (bottleneck).

Table 4.1: Resource in shortage and symptom rule book

Resource in Shortage	**Packet Drop Location**
CPU	TUN (aggregated)
Memory Space	pNIC Driver
Memory Bandwidth	TUN (aggregated)
Incoming Bandwidth	pNIC
Outgoing Bandwidth	Backlog Enqueue
pCPU Backlog	Backlog Enqueue
VM Bottleneck (CPU or Bandwidth)	TUN (individual)

4.4.2 Combating Propagation

Performance problems, e.g., due to implementation bugs, in one middlebox in a chain may propagate to others, causing them to appear to perform poorly as well. In particular, as we show below, the other middleboxes may appear to have stalled reading/writing from/to the network or I/O devices. Given this, however, determining the root cause middlebox can be challenging.

In what follows, we show how the statistics gathered by PerfSight can be used to infer the specific *states* that different middleboxes are operating under—e.g., under/over-loaded and read/write-blocked—which can then be used to identify the root cause.

Analysis and key insights: Before describing our algorithm for identifying the root cause middlebox, we first present the underlying insights in identifying the states of different middleboxes in a chain.

In virtual settings, middlebox software exchanges data with the guest OS. Consider some time interval t_{total}; from the middlebox's perspective, this time interval can be split across the middlebox performing input, processing, and output. Thus, we have

$$t_{total} = t_{input} + t_{process} + t_{output}.$$

The input/output time is constituted by block time, t_{block}, and memory copy time, t_{memcpy}; i.e.:

$$t_{input/output} = t_{block} + t_{memcpy}.$$

Input block time is the time spent waiting for new data, and output block time is spent waiting for buffers in kernel to be ready (e.g., waiting for TCP sending window to open up). Memory copy time is the time spent on copying data between user space and kernel space. When the middlebox software is blocked or processing, the kernel is waiting for the packet arrival interrupe from the vNIC or transmitting data from/to the vNIC.

The input function of middlebox software (e.g., recv() in TCP, FromDevice() in Click [49]) fetches data from kernel space to user space. If the buffer in kernel is ready with data, the input function is not blocked ($t_{block} = 0$), so the input method is as fast as a memory copy (which is at least 2 orders of magnitude faster than network transmission). Assume that during the input function reading b bytes data from the kernel, the vNIC capacity is C and the memory copy speed is C_{mem}; in general, we have $C_{mem} \gg C$.

Thus, when there is no input blocking, we have:

$$t_{input} = t_{memcpy} = \frac{b}{C_{mem}} \ll \frac{b}{C}.$$

Thus, we can define a middlebox is *ReadBlocked* if it satisfies

$$\frac{b_{input}}{t_{input}} < C,$$

where b_{input} is the bytes read by the input function.

Similarly, when the output function (e.g., send() in TCP, ToDevice() in Click) sends data, the data (b bytes) would be copied from the user space to kernel space. As before, if the output function is not blocked,

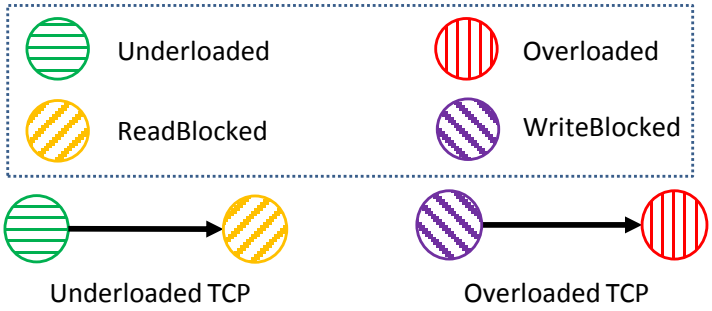
$$t_{output} \ll \frac{b}{C}.$$

Thus, we can define a middlebox is *WriteBlocked* if it satisfies

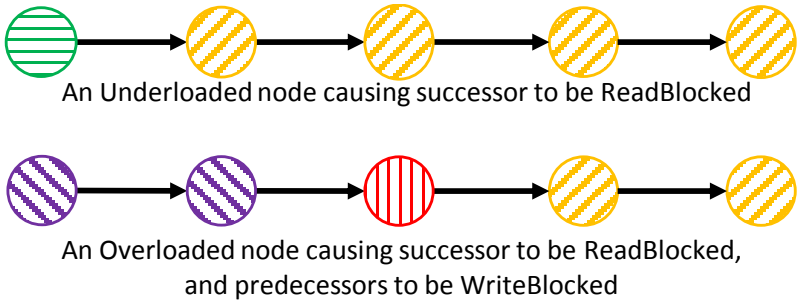
$$\frac{b_{output}}{t_{output}} < C,$$

where b_{output} is the bytes written by the output function.

If two neighboring (in virtual topology) middleboxes use nonblocking I/O methods (e.g., packet-level processing) to exchange messages, their states do not impact each other. If they use TCP, TCP's congestion control makes the middlebox states influence each other, causing propagation. In particular, there are two possibilities: (1) the TCP sender does not send fast enough, causing the receiver to be ReadBlocked, (2) the receiver cannot receive or process data quick enough, causing the sender to be WriteBlocked. We define the sender in (1) as *Underloaded* and the receiver in (2) as *Overloaded* (Figure 4.6(a)). In a chain of TCP connections, an Underloaded source causes all its successors to be ReadBlocked, and an Overloaded middlebox causes all its predecessors to be WriteBlocked and successors to be ReadBlocked (Figure 4.6(b)).



(a) Middlebox and TCP states



(b) State propagation in a Chain

Figure 4.6: Middlebox states and propagation

Accurate detection of root cause(s): Based on the above analysis and insights, we develop the following algorithm (see Algorithm 2) underlying diagnostic application for

determining the root cause middlebox(es): each middlebox's statistics are fetched (including input/output bytes/time and the middlebox vNIC capacity) (line 8). Then their respective states—under/overloaded, and read/write blocked—are computed based on the statistics (line 13- 16). If a middlebox is ReadBlocked, all its successors are waiting for its data and are also ReadBlocked; we filter out the entire such chain of ReadBlocked middleboxes from the suspicious candidates (line 15). Similarly, a WriteBlocked middlebox and all of its predecessors can be filtered out (line 18). The remaining middleboxes in the candidate set are returned (line 21) as the plausible root cause middleboxes. In Figure 4.6(b), this algorithm would identify the green middlebox in the first chain and the red middlebox in the second chain as the root cause. This application monitors all middleboxes in a virtual network, so its cost is linear to the size of the virtual network.

4.5 Implementation

Most elements (e.g., NIC driver and virtual switch) already have their own counters and logs implemented—in these cases, PerfSight simply makes use of the existing ones; if some counters are missing in an element, we instrument the elements and add augmented counters. The communication channel between an element and the corresponding agent on the physical server is implemented specifically according to the element. In the following paragraphs, we will describe the implementation details at each key element that a packet goes through on the software data plane. The environment we use to simulate the cloud is Linux kernel v3.20, Open vSwitch and QEMU.

For the *physical NIC*, the Linux kernel maintains a data structure called net_device which records received/sent bytes/packets. net_device can be accessed via the file system interface from userspace (e.g., ifconfig). The PerfSight agent uses this.

In the *NAPI routine* which processes packets in CPU backlog queues, a data structure called softnet_data is maintained for each queue. softnet_data records packets dropped between the CPU backlog queue and the target callback function (i.e., virtual switch packet handler). This record is accessable from the /proc file system.

Algorithm 2 Locate root cause middlebox

```
1:  function GETSTAT(tid, mb)
2:      attr←["inBytes","inTime","outBytes","outTime"]
3:      return GETATTR(tid, mb, attr)
4:  end function
5:  function GETROOTCAUSE(tid)
6:      midboxes←{mb|GetAttr(tid,mb,"type")="middlebox"}
7:      cand ← midboxes
8:      for mb ∈ midboxes do
9:          C ← GETATTR(tid, mb, "Capacity")
10:         < b1_i, t1_i, b1_o, t1_o > ← GETSTAT(tid, mb)
11:         sleep(T)
12:         < b2_i, t2_i, b2_o, t2_o > ← GETSTAT(tid, mb)
13:         if (t2_i − t1_i > (b2_i − b1_i)/C) then
14:             mb.state = ReadBlocked
15:             cand←cand−GETSUCCESSOR(mb)−mb
16:         else if (t2_o − t1_o > (b2_o − b1_o)/C) then
17:             mb.state = WriteBlocked
18:             cand←cand−GETPREDECESS(mb) −mb
19:         end if
20:     end for
21:     return cand
22: end function
```

In *virtual switches*, each rule has statistics for packets processed and dropped. The statistics are accessable via the OpenFlow control channel.

A *TAP*'s transmit function enqueues a packet into a socket queue. The TAP has a net_device record to keep track of Rx/Tx byte/packet counts, which are accessible via file system.

QEMU, which delivers packets from TAP socket to a vNIC data structure, does not have intrinsic statistics. Thus, we instrument it to obtain them (packet count, byte count, I/O time). We write these counters into logs and PerfSight fetches the counters' values from the logs.

Inside the *middlebox VM* (in the guest OS kernel), the difference from the virtualization stack is that the NAPI routine delivers a packet from vCPU backlog queues to another buffer in the kernel (e.g., socket). The statistics records are then similar to the virtualization stack.

Middlebox software reads packets/messages from the kernel via system calls. After a middlebox performs its own processing, it sends packets/messages out to the guest OS kernel via system calls. We instrument middlebox software to record Rx/Tx byte/packet counts and the time spent on reading from/writing to the kernel. In PerfSight, we use sockets between middlebox software and the agent corresponding to the physical server on which the middlebox is located to fetch these counters.

When a middlebox sends a packet out, the packet is delivered across several elements and finally put into the pCPU backlog. The write function to the kernel calls the vNIC transmit function, which causes an interrupt in QEMU. The interrupt handler in QEMU calls the transmit function in the TAP. The TAP transmit function enqueues the packets into the pCPU backlog queue. When dequeued, the packet is processed by the virtual switch and sent to another device's (another TAP or pNIC according to the packet's destination) transmit function. In each layer that the packet goes through, the instrumentation and communication with the agent are similar to the packet receiving datapath.

Discussion. We assume that the cloud provider offers the virtual network services and the NFV service, so they have the source code. PerfSight is a framework, and it defines interfaces between components. Even if some services (e.g., middlebox) are provided by third party, they can use the interface to provide statistics so that PerfSight can perform diagnosis.

4.6 Evaluation

We first validate PerfSight's functionalities including the utility functions and the performance diagnosis, and then benchmark its performance. Finally we evaluate PerfSight's overhead and discuss its scalability.

To evaluate PerfSight, we set up a cluster with Dell T5500 servers, each of which has 8 cores, 16GB memory and a 10Gbps NIC. Each server runs Ubuntu with Linux kernel 3.2. We use Open vSwitch, QEMU to set up the virtualization stack. We install middlebox software in VMs and route tenant traffic to traverse the VM to simulate NFV-like settings.

In what follows, we first validate the basic functionalities of PerfSight, and then show the effectiveness of using PerfSight by illustrating how it can diagnose performance problems, reflecting the examples outlined in Section 4.1. Finally we show that it imposes negligible overhead and discuss its scalability.

4.6.1 Functional Validation

We first show the output of basic utility functions, and then show how to use basic utility functions to monitor the software data plane state. Then we illustrate that PerfSight can support accurate diagnosis, and finally we give a synthetical example.

4.6.1.1 Utility Functions

We first give two examples of using PerfSight's utility functions to monitor states in elements. Figure 4.7 shows an example using PerfSight's interface to monitor the throughput in a physical NIC. Figure 4.8 shows that PerfSight supports packet size query.

4.6.1.2 Software Data Plane Monitoring

In what follows, we experimentally illustrate that PerfSight works as expected, using the example of the packet loss function (Section 4.3).

In this experiment, we start 8 VMs on a physical machine. Two of these VMs function as middleboxes and the rest as tenant VMs. The middlebox software we use is that of a load

```
def GetThroughput(tid, eid):
    attributes = ['rx bytes', 'tx bytes']
    (rx0, tx0) = GetAttr(tid, eid, attributes)
    time.sleep(1)
    while True:
        (rx, tx) = GetAttr(tid, eid, attributes)
        ts = datetime.now()
        print 'time: ', datetime.strftime(ts, '%H:%M:%S, '),
        print 'TX: ', (tx - tx0)/1000000 * 8, 'Mbps, ',
        print 'RX: ', (rx - rx0)/1000000 * 8, 'Mbps'
        (rx0, tx0) = (rx, tx)
        time.sleep(1)

root@rohan# ./throughput.py
time:  02:36:06,  TX:  504 Mbps,  RX:  0 Mbps
time:  02:36:07,  TX:  504 Mbps,  RX:  0 Mbps
time:  02:36:08,  TX:  400 Mbps,  RX:  0 Mbps
time:  02:36:09,  TX:  352 Mbps,  RX:  0 Mbps
time:  02:36:10,  TX:  360 Mbps,  RX:  0 Mbps
time:  02:36:11,  TX:  360 Mbps,  RX:  0 Mbps
time:  02:36:12,  TX:  352 Mbps,  RX:  0 Mbps
time:  02:36:13,  TX:  344 Mbps,  RX:  0 Mbps
```

Figure 4.7: Utility function: throughput

```
def GetThroughput(tid, eid):
    attributes = ['rx bytes', 'tx bytes']
    (rx0, tx0) = GetAttr(tid, eid, attributes)
    time.sleep(1)
    while True:
        (rx, tx) = GetAttr(tid, eid, attributes)
        ts = datetime.now()
        print 'time: ', datetime.strftime(ts, '%H:%M:%S, '),
        print 'TX: ', (tx - tx0)/1000000 * 8, 'Mbps, ',
        print 'RX: ', (rx - rx0)/1000000 * 8, 'Mbps'
        (rx0, tx0) = (rx, tx)
        time.sleep(1)

root@rohan# ./avg_pkt_size.py
time:  02:52:15,  avg pkt size:  66
time:  02:52:16,  avg pkt size:  66
time:  02:52:17,  avg pkt size:  66
time:  02:52:18,  avg pkt size:  66
time:  02:52:19,  avg pkt size:  66
```

Figure 4.8: Utility function: average pakcet size

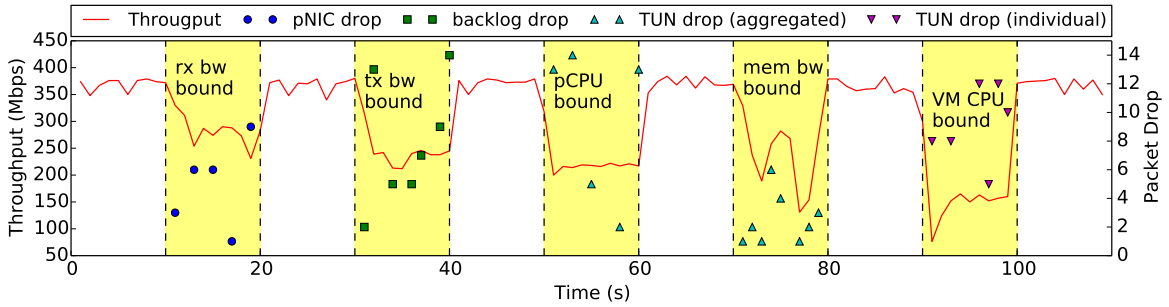


Figure 4.9: Throughput and packet prop in the virtualization stack during performance problems

balancer [1]. We start a handful of long-lived TCP flows that traverse the middlebox VMs, and we monitor their throughput. Over time, we inject various performance problems. All the while, we run PerfSight's packet loss function at various software data plane elements in the 8 VMs to identify if, when and where packet loss happens.

The results are shown in Figure 4.9. We plot the average throughput of the middlebox flows on the left y axis, and the packet drop counts on the right y axis.

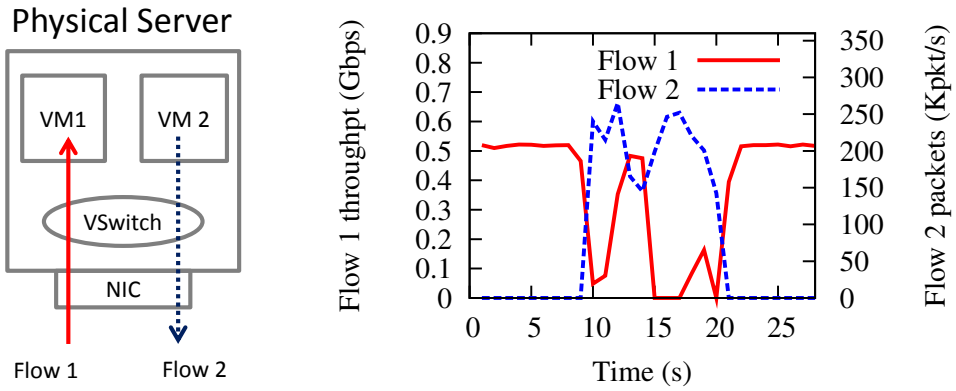
During the interval 10-20 seconds, we flood a large amount of packets into the physical machine (these are received by the non middlebox VMs). As such, the virtualization stack cannot clear the pNIC DMA buffer quickly enough, leading to lower TCP throughput for the middlebox VMs. What we noticed from PerfSight was that packets were dropped in the pNIC, which is as we expected (Table 4.1).

During 30-40 seconds, the tenant VMs (non middlebox) flood a large amount of outgoing packets impacting middlebox traffic throughput. The flooded packets quickly fill up the CPU backlog first, leading to drops there for middlebox traffic, which is again as expected (Table 4.1).

During 50-60 and 70-80 seconds, we make tenant VMs perform CPU intensive and memory access intensive workloads. As a result, the middlebox VMs do not acquire enough resources to process packets. This causes their packets to be accumulated and dropped at the TUN's socket buffer (which is the last buffer before entering VMs). When VMs are contending for resources, all VM's performance are impacted, so all VMs are dropping packets.

While the above showed various forms of contention and interference, we now illustrate a single VM becoming a bottleneck. During 90-100 seconds, we start a CPU intensive workload inside one middlebox VM. From PerfSight, we note that only that VM drops packets at its associated TUN.

4.6.1.3 Diagnosis Accuracy



(a) 2 VMs (b) VM Traffic

Figure 4.10: An example of CPU backlog queue contention detection

We now establish PerfSight's accuracy. We first conduct two experiments for detecting contention.

Detecting contention in virtualization stack (case 1). In Figure 4.4 we can observe that the pCPU backlog queue is exercised by multiple datapaths, which leads to the risk of it becoming a location where significant contention arises.

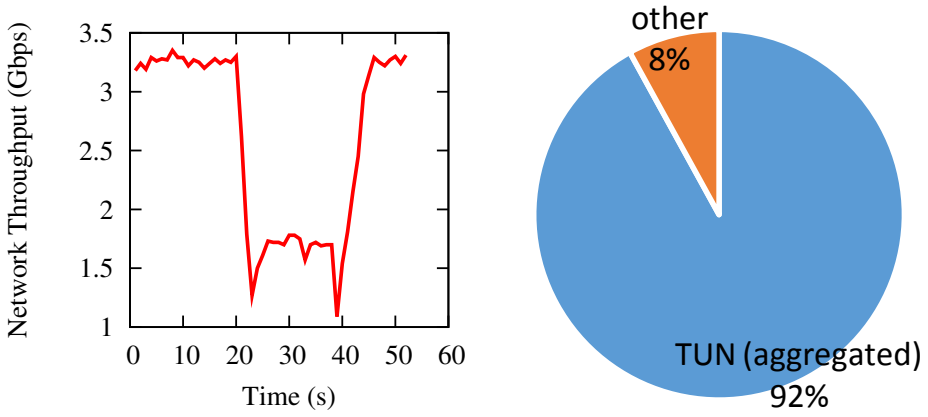
We set up two VMs – VM1 and VM2 – in a physical machine, and then make VM1 receive network traffic with a rate limit of 500Mbps. Roughly 10s into the measurement, we make VM2 send small packets as fast as it can. At this time, VM2's throughput decreases and oscillates; see Figure 4.10.

To diagnose this contention problem, PerfSight first checks if the VMs are overwhelming the NIC. VM2's peak sending rate (obtained using the GetThroughput() routine) is 250K

packets per second (80Mbps); thus, the sum of the sending and receiving rates is well below the NIC capacity (1Gbps). PerfSight then checks packet drop counters at various elements in the virtualization stack. We found that the enqueue element in the virtualization stack (Figure 4.4) saw significant drops, and because outgoing bandwidth is not the problem (Table 4.1), the resource under contention ought to be the pCPU backlog queues (Table 4.1).

In a virtualized setup, both the incoming and outgoing packets are put into pCPU backlog queues first, and then forwarded during backlog processing. However, each CPU core's backlog queue length is limited to 300 packets. In our example, VM2 overwhelms the pCPU backlogs by flooding a large amount of small packets, and only a handful of VM1's packets can ever be enqueued into the backlog queue. This manifests as many more packets of VM1 arriving into the enqueue element and far fewer making it out.

Without PerfSight helping us identify where exactly packets are getting dropped and tying that back to the resource under contention (Table 4.1), it is difficult to determine the reason for VM1 and VM2 interfering with each other's performance.



(a) Network throughput (b) Packet drop

Figure 4.11: An example of memory bandwidth contention detection

Detecting contention in virtualization stack (case 2). Along the lines of the example Section 4.1, we simulated a machine with significant oversubscription in the virtualization

stack. We start by running on a machine some number of VMs performing network transfers initially; their total network throughput is about 3.25Gbps as shown in Figure 4.11(a). At time 20s, another set of VMs start to access memory intensively. The total network throughput decreases to 1.7Gbps, because the two sets of VMs are contending for the memory bandwidth. We assume that the tenant running the former set of VMs wishes to diagnose this problem.

We observe that the physical machine is dropping packets at the network-intensive VMs' TUNs (Figure 4.11(b)). Thus, we infer that the machine's memory or outgoing bandwidth are overloaded (based on Table 4.1); at this time, we cannot distinguish which specific resource is experiencing contention. In response, wearing the operator's hat, we migrate some of the network-intensive VMs, and the network throughput recovers to the original 3.2Gbps.

Combating propagation. We validate our approach for identifying poorly-performing middleboxes in the face of propagation of problems where root cause cannot be obviously identified. The scenario we use is a multi-chain setting shown Figure 4.12(a): we deploy a load balancer (Balance [1]) and two content filter proxies (CherryProxy [2]) between clients and HTTP servers. We make the two content filter proxies output logs to a shared file system (NFS). All VMs' vNIC capacity are set to be 100Mbps.

In this virtual network, we simulate different cases and perform diagnosis; we find that our application always determines the root cause accurately. We conducted a variety of experiments where different points in this multi-chain are problematic and the issues can propagate arbitrarily through the chains, but for simplicity we focus on those exercising the VMs shown within the box shown by the dashed red line. For each experiment (Figures 4.12(b), 4.12(c), and 4.12(d)) we show the performance metrics we derived for each middlebox (e.g., b/t_{in}, b/t_{out}), and the corresponding state we inferred for the middlebox.

In our first experiment, we make the client perform HTTP POSTs as fast as possible (with the idea of creating a bottleneck at the server within the dashed red box); we observe the state of middleboxes and determine that the load balancer and the content filter are WriteBlocked and the NFS server is Readblocked (Figure 4.12(b)). From this, our algorithm infers that server 1 is overloaded, identifying the true bottleneck.

In a second run, we have the client make HTTP POST requests at a slow rate. We monitor middlebox states and determine that other middleboxes are ReadBlocked, leading our algorithm to conclude that the client is Underloaded, which is indeed the case (Figure 4.12(c)).

As a final experiment, we inject an internal error (memory leak) into the NFS server,[1] causing it to become overloaded. The problem propagates through the chain middleboxes within the dashed red box, causing the load balancer and the content filter to be WriteBlocked. Using our algorithm, we exclude the ReadBlocked and WriteBlocked middleboxes and correctly determine that the NFS server is the bottleneck (Figure 4.12(d)).

4.6.1.4 PerfSight for Network Management

We now consider a richer setting where we illustrate how we envision an operator using PerfSight. We show how an operator can detect and respond to contention and bottlenecks. We then describe propagation.

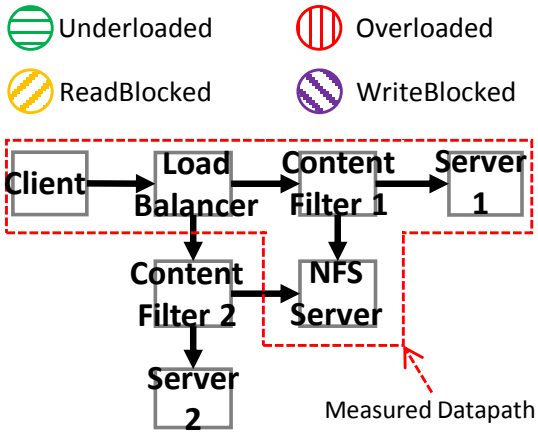
We use a set up with two tenants each with their own virtual network. Each network has a server, a load balancer proxy and a client with the client sending traffic to the server (Figure 4.14). We assume that the operator places the two load balancers in the same physical machine. We measure both tenants' throughput and show it in Figure 4.13.

Initially (0-10s), tenant 1 sends traffic at 180Mbps, while tenant 2 intends to send two flows at 360Mbps in total. However, tenant 2's load balancer can only process 200Mbps traffic, so tenant 2's total throughput is constrained by its load balancer.

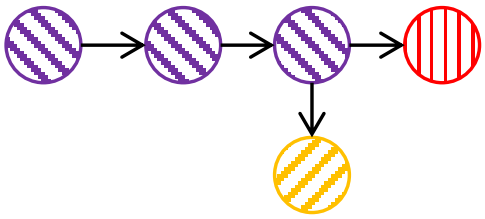
At this point, using PerfSight the operator identifies that the TUN of load balancer 2 is dropping packets and it is in an Overloaded state. Thus, the operator has identified tenant 2's bottleneck.

Between 10 and 20s, the operator introduces a management task that happens to be memory access intensive into the physical machine. The operator finds that both tenants' load balancer VMs are impacted, they're dropping packets at their TUNs, and they're in ReadBlocked state. The operator identifies memory bandwidth over-subscription and responds by

[1] CentOS bug track 7267.

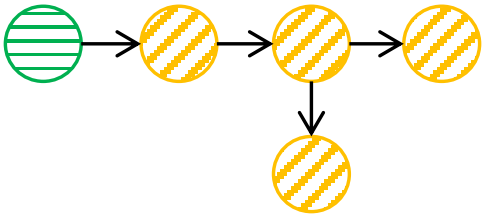


(a) Legend and topology



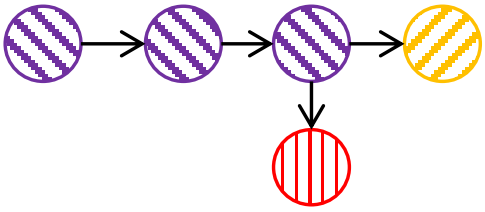
	LB	CF	NFS	Server
b/t_i	323	362	3.2	316
b/t_o	32	36	N/A	N/A

(b) Overloaded server



	LB	CF	NFS	Server
b/t_i	16.2	15.6	1.7	17
b/t_o	248	160	N/A	N/A

(c) Underloaded client



	LB	CF	NFS	Server
b/t_i	138	154	126	10
b/t_o	35	11.5	N/A	N/A

(d) Problematic NFS

Figure 4.12: Root cause detection in the face of propagation (Unit: Mbps, b: bytes, t_i: t_{input}, t_o: t_{output})

migrating the memory intensive task elsewhere. Thus, we see that the throughput immediately reverts to the original value (20-30s).

However, this still does not address the bottleneck that tenant 2 is facing at its load balancer. After determining that tenant 2 is bottlenecked at its load balancer, the operator scales it out and reroutes half of tenant 2's traffic to the new instance. The total throughput increases to 360Mbps and no packets are dropped at the load balancers.

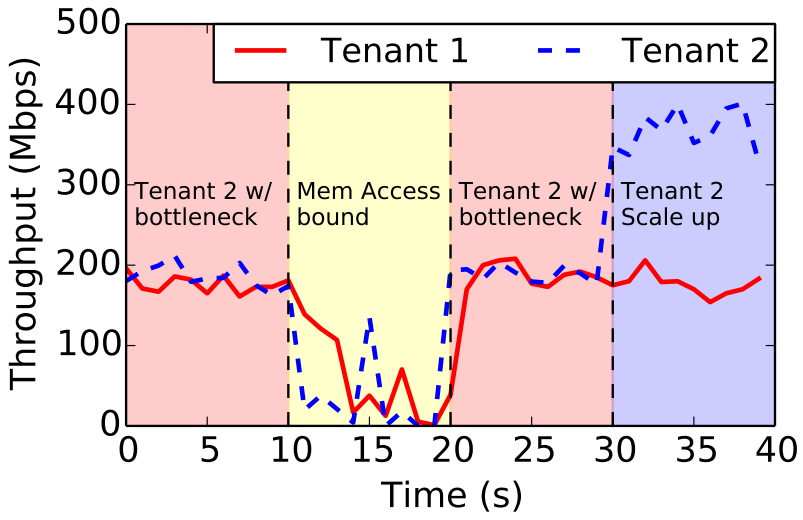
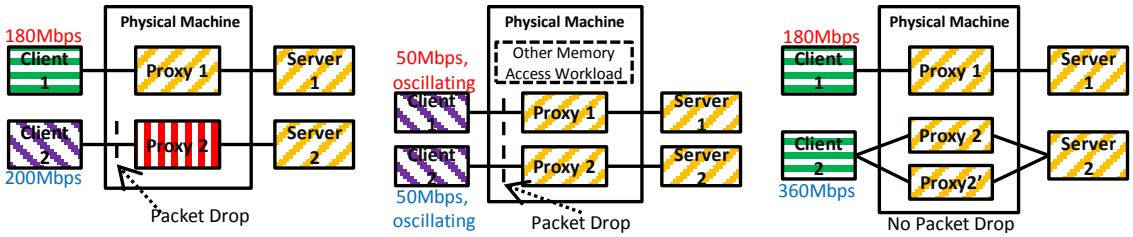


Figure 4.13: Throughput in multi-tenant experiment

4.6.2 Performance, Overhead and Scalability

Response time in statistics gathering. The agent on each physical machine is the pivot of data collection and delivery. We measure how quickly the agent can exchange data with other components. As is shown in Figure 4.15, fetching statistics from network devices (e.g., TUN, pNIC) costs about 2ms, and all other components' statistics collection can be completed in 500us. This time granularity is fine enough to closely monitor the network states.

Overhead of intrumentation. Among the counters we implemented, packet counts/bytes are simple/low overhead because each time they are incremented by a value; time counters are a bit more complex because they need to get time twice, and accumulate the difference. We first measure the time spent to update a counter. We find that simple counters consume 3ns per update, while a timer counter consumes 0.29us per update in our testbed. Even with the maximum throughput of 10Gbps and 1500 bytes MTU, each packet takes 1.2us to traverse an element. Thus, the simple counters impose 10^{-3} smaller overhead, which is negligible.



(a) Bottleneck (b) Memory access bound (c) Scale up

Figure 4.14: Using PerfSight to manage a multi-tenant cloud

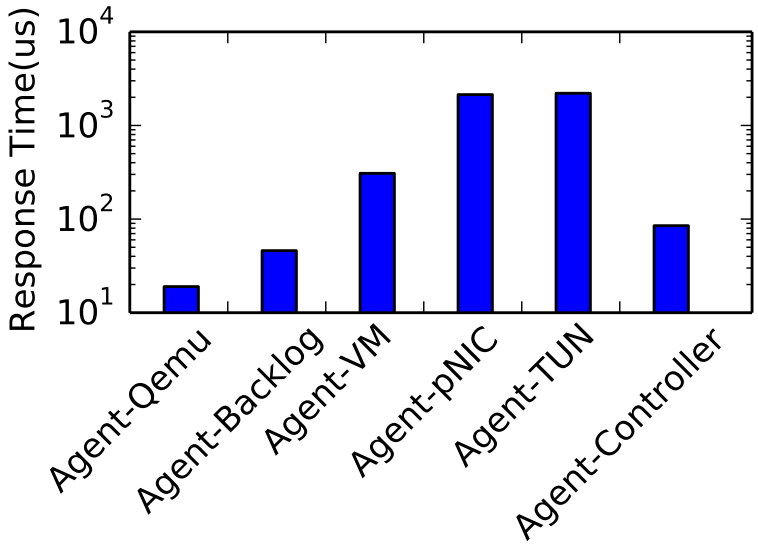


Figure 4.15: Response time between the agent and other components

However, the time counters do have an impact on performance. If an element is overloaded and CPU bound, time counters would cost extra CPU cycles, which degrades the element's performance. To delve into this, we build a virtual network with an HTTP server, an HTTP client and a proxy. The client uploads data to the server via the proxy. In this setting, if we limit the client's sending rate (on its vNIC), the proxy will be ReadBlocked; if not, TCP would saturate the virtual link and cause the proxy to become Overloaded. We compare the throughput between the cases where the proxy is/not instrumented with time counters. The result is shown in Table 4.2. We can conclude that the impact of the time

Table 4.2: Throughput with/without time counters

1: Blocked, without counters, 2: Blocked, with counters

3: Overloaded, without counter, 4: Overloaded, with counters.

Each experiment is repeated for 100 times.

Experiment	1	2	3	4
Mean μ (Mbps)	42.02	41.79	499	490.2
Variance σ^2	4.57	4.92	1554	1281

counters is very small (under 2% in terms of throughput; we also found the average latency impact to be under 1.5% (not shown for brevity)).

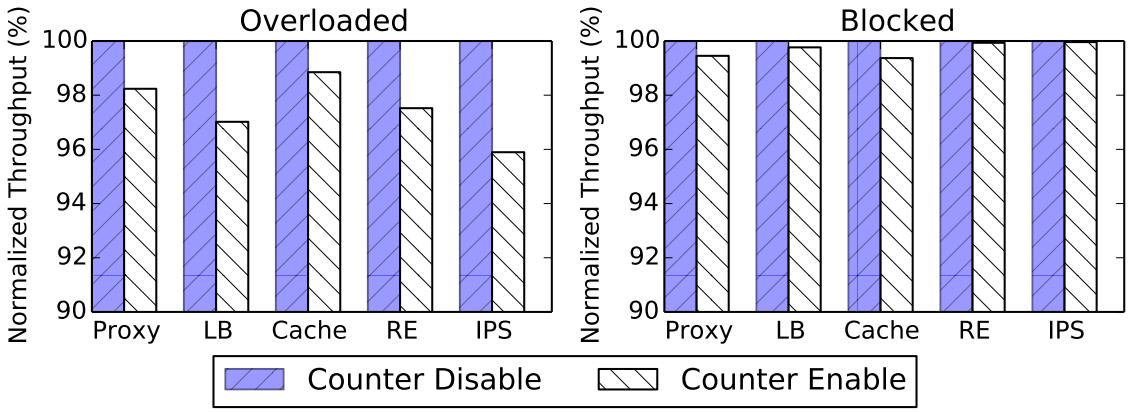


Figure 4.16: Time counter overhead within middleboxes

We repeat the similar experiments on different kinds of middleboxes [23, 1, 16], and show the results in Figure 4.16. In all kinds of middleboxes, the impact is less than 5%.

Overhead of statistics gathering. Another source of overhead is from polling these counters. The overhead is shown in Figure 4.17, and it still very small. Even if we poll them every 100ms (which is generally sufficient for the kind of diagnostics we wish to run), the CPU utilization is less than 0.5%, which is negligible.

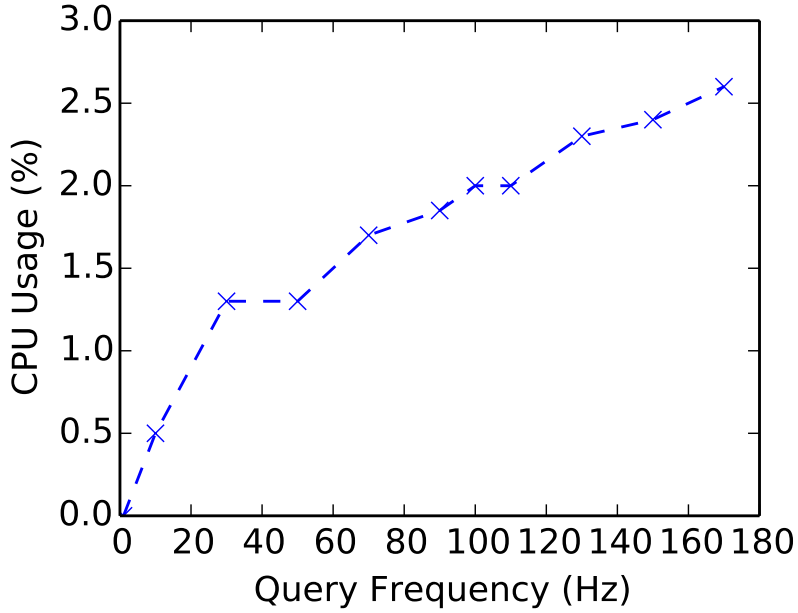


Figure 4.17: Query frequency and CPU usage

Scalability. Our experimental set up is indeed small, but we believe that our system's response time and overhead will stay small even at larger set ups due to the following reasons: (1) The statistics gathering is distributed in each element, so it's not likely to become a bottleneck. (2) The diagnostic applications' complexity is O(n) where n is the number of involved elements. (3) Cloud operators can aggregate tenants' tickets to diagnose if they have elements overlapping with each other.

4.7 Limitations and Opportunities

When we designed PerfSight, we intended to solve performance problems. Cloud network data planes may have various other problems, e.g., availability and isolation issues. Thus, PerfSight is not guaranteed to be complete to solve all problems in data planes. We list several issues that cannot be solved (completely) by PerfSight below.

- **Availability issues.** There are several factors that may cause data plane to be unavailable. For example, misconfigurations from the control plane or logical errors in the

data plane may both cause all traversing packets to be dropped. In this case, using PerfSight, the operator could find packet drop locations, but could not distinguish the root cause.

- **False negative.** A piece of middlebox software has its own logic. Thus it may drop or delay packets according to its own logic. In that case, PerfSight may falsely find the middlebox to be problematic. In this case, dropping packets is no longer the symptom of performance problems.

- **Latency problem.** By default, a TCP connection tries to saturate the link capacity on its path. If an element has large latency, it will process packets slowly, and becomes a bottleneck on the datapath (with the symptom of dropping packets). However, if the application controls the sending rate so that TCP cannot saturate the link capacity, the element will cause latency of the connection without dropping packets. In this case, PerfSight cannot discover the problem.

PerfSight focuses on performance problems, and, as such, it complements existing solutions. Network operators can use other solutions to check availability issues (e.g., Anteater, HSA, software verification). PerfSight can also look deeper into the element logic to distinguish packet drops caused by internal logic and bottlenecks. PerfSight can be extended to add statistics such as processing time to detect latency issues; the extension requires tracing packets and records the timestamps before and after processing.

4.8 Summary

With the advent of NFV, data planes are becoming increasingly complex, involving a variety of software packet processing elements. In this chapter, we argued that these new "software data planes" are susceptible to subtle performance problems that don't occur (or are infrequent) in traditional hardware-based data planes. We argued that diagnosing these problems is difficult because no existing tool or system provides the right level of information to tease apart various potential causes of the observed degradation. To this end, we present a

system, PerfSight, a ground-up approach for extracting comprehensive low-level information regarding packet processing performance of the various elements in the data plane and for conducting rich analysis on the information gathered. Through careful experiments, we show that our framework can result in accurate detection of the root causes of performance problems in software data planes, and it imposes very little overhead.

Chapter 5

Related Work

In this chapter, we discuss various efforts in the area of network diagnosis. We first discuss the technologies used for network diagnosis. We then discuss network diagnostic solutions built based on these supporting technologies. Finally, we discuss network diagnostic algorithms that can be used in network diagnostic solutions.

5.1 Supporting Technologies

On end hosts, we have various utilities such as ping, traceroute and tcpdump [22, 21]. Ping and traceroute check reachability in the network, and tcpdump captures packet traces and prints packet contents. RFC 4898 [12] proposes to record extended performance statistics for TCP, including TCP states and various counters (e.g., bytes, retransmissions).

On switches (both virtual and physical), the industry has standards such as sFlow, NetFlow and Switched Port Analyzer (SPAN) [20, 13, 14]. sFlow samples packets and delivers the samples to a server. NetFlow enables a switch to collect statistics about packets (e.g., IP address, port and protocol) and deliver the statistics to a server. SPAN is port mirroring; it enables a switch to duplicate traffic of one port to another. In the current OpenFlow standard [58], each switch rule has its own statistics (e.g., packet count, drops), which helps the network operator to monitor the routing status.

SNMP [17] is widely used in network management systems to monitor network-attached devices. The protocol defines the format of the management data between network devices and the SNMP manager.

All these technologies provide detailed information in the network. These tools do not point out problems in the network directly. They rely on the network operator's experience to judge the problem based on the information. These technologies usually form the foundation of network diagnostic solutions (as described in the next section and our VND and PerfSight).

5.2 Network Diagnostic Solutions

5.2.1 Solutions for Network Stacks

SNAP [66] is designed based on RFC 4898. SNAP collects TCP statistics and socket-call logs. By observing abnormal behaviors of a TCP flow (e.g., delayed ACK), the operator can narrow down the scope of a fault; by correlating multiple TCP flows on their faulty time, the operator can find common application misconfigurations.

In X-Trace [33], the authors proposed inserting metadata into packets of application tasks. The metadata is preserved as the packets traverse different network layers. By observing whether the metadata arrives at each layer successfully, the operator knows whether a certain layer is functioning correctly.

In NeST [57], the authors proposed instrumenting the network stack. States are defined for each component in the network stack based on whether they are processing or waiting for messages. NeST uses a dependency graph of components and the components' states to infer the stalled component.

Compared with these solutions, our PerfSight has two differences. First, PerfSight is designed for cloud networks instead of traditional networks. The cloud networks have more components and the components are much more heterogeneous (i.e., blocking/nonblocking middleboxes). Second, X-Trace and NeST target the availability issue. They can check whether a layer/component is processing and delivering packets. PerfSight targets performance problems, whose symptoms are not as obvious as availability problems.

5.2.2 Solutions for Traditional Networks

HSA and Anteater [46, 56] model switch forwarding behaviors and packet headers. The model can describe the way in which packets traverse a network device. The authors proposed network invariants, such as reachability and loop-free, and used the model to check these invariants.

Libra [67] similarly checks network invariants. It targets networks on a large scale. In Libra, the authors proposed a map-reduce algorithm to check the network invariants.

Compared with our approaches, VND uses packet traces to perform diagnosis directly, and PerfSight targets a different scope (i.e., performance problems).

5.2.3 Solutions for Software-Defined Networks

OFRewind [65] introduced a proxy between the SDN controller and the switches. The proxy records the control messages in the SDN network, and provides the replay functionality. By replaying the control messages, the network problem can reappear, enabling the operator to debug the system.

NetSight [41] is a network-wide wireshark. NetSight collects all packet histories in the network (optimized by compression), and uses regular expressions as interfaces to filter packets of interest. The authors designed four applications based on the interfaces. The applications check network invariants or profile various network components (e.g., a path or a device).

Frenetic [34] provides an abstraction layer to operate OpenFlow switches based on the OpenFlow syntax. This layer enables the operator to specify packets of interest and perform operations such as group and split.

VeriFlow and NetPlumber [47, 45] are two solutions use to monitor network invariants lively. They add a proxy between the SDN controller and the switches in the network. The proxy intercepts the control messages. Both solutions model the switch forwarding behaviors and use their models to check network invariants such as reachability, loop-free and consistency.

NICE [29] aims to find bugs in SDN controller applications. It leverages symbolic execution to perform model checking. During the symbolic execution, the states in the controller are used to check the network invariants.

In all these solutions, VeriFlow, NetPlumber and NICE solve problems in the network control plane, which is a different scope compared with our works. OFRewind, NetSight and Frenetic have similarities with our VND approach. Our VND design considers the cloud environment, and its interfaces retain the abstractions (i.e., a tenant only has views of its own virtual network). VND's SQL interface provides more operations than NetSight and OFRewind, therefore, it simplifies tenants' operations. Compared with Frenetic, VND's view comprises is the entire network instead of a single switch, so the tenant can debug the entire virtual network comprehensively.

5.3 Network Diagnostic Algorithms

Another set of research literature discusses network diagnostic algorithms. In these works, the network is usually viewed as a topology composed of nodes and links. These works aims to use misbehavior symptoms to locate faulty nodes or links. These algorithms can be integrated with VND as diagnostic applications, but they do not discuss the operational feasibility in virtual networks. PerfSight instruments deeply into the software data plane at a finer granularity. These diagnostics algorithms are as follows.

In netDiagnoser and Max-Coverage [31, 51], the end-to-end path reachability is monitored. The links that have the most intersections with the problematic path are suspected to be the root cause links. In SCORE [50], the links in a topology are put into groups if they share the same risk of failure (e.g., links may share a fiber). SCORE observes the ill-performing links and uses a greedy algorithm to determine the most suspectable group(s). Shrink [43] uses a similar model as SCORE, but it uses a Bayesian model to infer the root cause group. In [48], the authors proposed using a codebook for troubleshooting. The two-dimension codebook has multiple symptoms as rows and a set of known problems as columns. Each problem is marked with one or several symptoms. New problems are

matched in the codebook to find the problem with the most similar symptoms. In NetMedic and Sherlock [44, 24], the network topology is viewed as a Bayesian network, and the faulty probability is computed by historical statistics. In the case of a network problem, the root case is determined by the probability and the Bayesian network.

Chapter 6

Conclusion

In this thesis, we have demonstrated new network problems in the cloud environment and our approaches to solve them. We have shown the feasibility and performance of our approach. Here, we highlight the main contributions of our works, and then close this thesis with a discussion of options for future work.

6.1 Contributions

VND. We proposed a virtual network diagnostic service from the cloud provider to its cloud tenants. We identified a set of technical challenges in providing such a service and propose a Virtual Network Diagnosis (VND) service framework. VND exposes abstract configuration and query interfaces for cloud tenants to troubleshoot their own virtual networks. It controls software switches to collect flow traces, distributes traces storage, and executes distributed queries for different tenants for network diagnosis. It reduces the data collection and processing overheads by performing local flow capture and on-demand query execution. Our experiments validated the functionality of VND approach and showed its feasibility in terms of quick service response and acceptable overhead; our simulation proved the VND architecture cloud be scaled to the size of a real data center network.

PerfSight. The new "software data planes" in the cloud infrastructure are susceptible to at least three new classes of performance problems. To diagnose such problems, we designed, implemented and evaluated PerfSight, a ground-up system that works by extracting comprehensive low-level information regarding packet processing and I/O performance of

the various elements in the software data plane. PerfSight then analyzes the information gathered in various dimensions (e.g., across all VMs on a machine, or all VMs deployed by a tenant). By looking across aggregates, we showed that it becomes possible to detect and diagnose key performance problems. Our experimental results showed that our framework can result in accurate detection of the root causes of key performance problems in software data planes, and it imposes very little overhead.

6.2 Future Work

Our study has not covered all aspects of the cloud infrastructure. There are still other network troubleshooting issues in the cloud infrastructure, which we consider as research areas for future work. In addition, our study of the network problems also points out a way to improve the network performance.

Control Plane. The cloud control plane translates and deploys management policies (e.g., virtual networks) into low-level devices. There are several layers in this process: the management policy, the logic view, the physical view, and the device states. Diagnostic tools for traditional networks usually target the physical-view layer and the device states; they have proposed network-wide invariants such as loop-free, reachability in these two layers. I argue that in the public cloud, more invariants need to be guaranteed, such as isolation and fault tolerance. I intend to model this layer-by-layer translation and the invariants. In addition, I believe we need to use the actual data plane behaviors to verify the network invariants instead of the states in the control plane (as is done in current solutions); to implement this, we can make use of existing techniques such as sFlow, NetFlow or VND to capture the network traffic and to verify whether the data plane behavior violates the network invariants.

Software Data Plane Optimization. An observation of our data plane diagnostic work is that processing overheads is imposed in the software data plane, causing unsatisfactory performance (e.g., low throughput). The overhead is usually caused by uncoordinated design of data plane components (these components are usually designed for generic usage). In a

specific environment (e.g., multi-tenant cloud), the software data plane can be further optimized. For example, two VMs in the same physical server can exchange network traffic with zero memory copy; VM outgoing traffic can bypass the NAPI routine to the NIC directly. I intend to further optimize the the software data plane in the context of multi-tenant clouds.

I would like to discover whether the software data plane can be software-defined. The optimization examples mentioned above actually design new "short-cut" datapaths for network traffic to accelerate processing speed. It is possible to control the datapath in the software data plane, so that different flows go through different datapaths and achieve different benefits. For example, trusted VMs can exchanges packets directly without going through virtual switches, while untrusted VMs should not for security reasons; delay-sensitive middlebox traffic can output directly to the physical NIC, while others are put into the CPU backlog queue for bulk processing. This requires a new design and implementation of the software data plane and the comprehensive evaluation of different practices.

LIST OF REFERENCES

[1] Balance: the open source load-balancer and tcp proxy. http://www.inlab.de/balance.html.

[2] Cherryproxy: a filtering http proxy extensible in python. http://www.decalage.info/python/cherryproxy.

[3] Cisco global cloud index: Forecast and methodology, 20132018. Technical report.

[4] http://nicira.com/en/network-virtualization-platform.

[5] http://rightscale.com.

[6] https://bugzilla.kernel.org/buglist.cgi?quicksearch=kvm.

[7] https://en.wikipedia.org/wiki/amazon_web_services#history.

[8] https://en.wikipedia.org/wiki/microsoft_azure#significant_outages.

[9] http://sourceforge.net/p/balance/bugs/.

[10] http://sourceforge.net/p/unfs3/bugs/.

[11] https://patchwork.ozlabs.org/project/openvswitch/list/.

[12] https://www.ietf.org/rfc/rfc4898.txt.

[13] http://tools.ietf.org/html/rfc3954.html.

[14] http://www.cisco.com/c/en/us/tech/lan-switching/switched-port-analyzer-span/index.html.

[15] Path mtu discovery. http://tools.ietf.org/html/rfc1191.

[16] Snort: Open source network intrusion prevention. http://www.snort.org.

[17] www.net-snmp.org.

[18] www.openstack.org.

[19] www.openvswitch.org.

[20] www.sflow.org.

[21] www.tcpdump.org.

[22] www.traceroute.org.

[23] Ashok Anand, Vyas Sekar, and Aditya Akella. Smartre: an architecture for coordinated network-wide redundancy elimination. In ACM SIGCOMM Computer Communication Review, volume 39, pages 87–98. ACM, 2009.

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

[25] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In USENIX Annual Technical Conference, FREENIX Track, pages 41–46, 2005.

[26] Theophilus Benson, Aditya Akella, and David A Maltz. Network traffic characteristics of data centers in the wild. In Proceedings of the 10th ACM SIGCOMM conference on Internet measurement, pages 267–280. ACM, 2010.

[27] Theophilus Benson, Aditya Akella, Anees Shaikh, and Sambit Sahu. Cloudnaas: a cloud networking platform for enterprise applications. In Proceedings of the 2nd ACM Symposium on Cloud Computing, page 8. ACM, 2011.

[28] Tian Bu, Nick Duffield, Francesco Lo Presti, and Don Towsley. Network tomography on general topologies. In Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '02, pages 21–30, New York, NY, USA, 2002. ACM.

[29] Marco Canini, Daniele Venzano, Peter Peresini, Dejan Kostic, Jennifer Rexford, et al. A nice way to test openflow applications. In NSDI, volume 12, pages 127–140, 2012.

[30] Margaret Chiosi, D Clarke, P Willis, A Reid, J Feger, M Bugenhagen, W Khan, M Fargano, C Cui, H Deng, et al. Network functions virtualisation–introductory white paper. In SDN and OpenFlow World Congress, 2012.

[31] Amogh Dhamdhere, Renata Teixeira, Constantine Dovrolis, and Christophe Diot. Netdiagnoser: Troubleshooting network unreachabilities using end-to-end probes and routing data. In Proceedings of the 2007 ACM CoNEXT conference, page 18. ACM, 2007.

[32] Seyed Kaveh Fayazbakhsh, Vyas Sekar, Minlan Yu, and Jeffrey C Mogul. Flowtags: Enforcing network-wide policies in the presence of dynamic middlebox actions. In Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking, pages 19–24. ACM, 2013.

[33] Rodrigo Fonseca, George Porter, Randy H Katz, Scott Shenker, and Ion Stoica. X-trace: A pervasive network tracing framework. In Proceedings of the 4th USENIX conference on Networked systems design & implementation, pages 20–20. USENIX Association, 2007.

[34] Nate Foster, Rob Harrison, Michael J Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A network programming language. In ACM SIGPLAN Notices, volume 46, pages 279–291. ACM, 2011.

[35] Maurice Gagnaire, Felipe Diaz, Camille Coti, Christophe Cerin, Kazuhiko Shiozaki, Yingjie Xu, Pierre Delort, Jean-Paul Smets, Jonathan Le Lous, Stephen Lubiarz, et al. Downtime statistics of current cloud solutions. International Working Group on Cloud Computing Resiliency, Tech. Rep., June, pages 176–189, 2012.

[36] Aaron Gember, Robert Grandl, Ashok Anand, Theophilus Benson, and Aditya Akella. Stratos: Virtual middleboxes as first-class entities. UW-Madison TR1771, 2012.

[37] Aaron Gember-Jacobson, Wenfei Wu, Xiujun Li, Aditya Akella, and Ratul Mahajan. Management plane analytics. In Proceedings of the 2015 ACM Conference on Internet Measurement Conference, pages 395–408. ACM, 2015.

[38] Ali Ghodsi, Vyas Sekar, Matei Zaharia, and Ion Stoica. Multi-resource fair queueing for packet processing. ACM SIGCOMM Computer Communication Review, 42(4):1–12, 2012.

[39] Ashvin Goel, Charles Krasic, Kang Li, and Jonathan Walpole. Supporting low latency tcp-based media streams. In Quality of Service, 2002. Tenth IEEE International Workshop on, pages 193–203. IEEE, 2002.

[40] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. I know what your packet did last hop: Using packet histories to troubleshoot networks. In Proc. NSDI, 2014.

[41] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. I know what your packet did last hop: Using packet histories to troubleshoot networks. In Proc. NSDI, 2014.

[42] Nikhil Ashok Handigol. Using packet histories to troubleshoot networks. PhD thesis, Stanford University, 2013.

[43] Srikanth Kandula, Dina Katabi, and Jean-Philippe Vasseur. Shrink: A tool for failure diagnosis in ip networks. In Proceedings of the 2005 ACM SIGCOMM workshop on Mining network data, pages 173–178. ACM, 2005.

[44] Srikanth Kandula, Ratul Mahajan, Patrick Verkaik, Sharad Agarwal, Jitendra Padhye, and Paramvir Bahl. Detailed diagnosis in enterprise networks. ACM SIGCOMM Computer Communication Review, 39(4):243–254, 2009.

[45] Peyman Kazemian, Michael Chan, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. Real time network policy checking using header space analysis. In NSDI, pages 99–111, 2013.

[46] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In NSDI, pages 113–126, 2012.

[47] Ahmed Khurshid, Wenxuan Zhou, Matthew Caesar, and P Godfrey. Veriflow: verifying network-wide invariants in real time. ACM SIGCOMM Computer Communication Review, 42(4):467–472, 2012.

[48] Shmuel Kliger, Shaula Yemini, Yechiam Yemini, David Ohsie, and Salvatore J Stolfo. A coding approach to event correlation. Integrated Network Management, 95:266–277, 1995.

[49] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M Frans Kaashoek. The click modular router. ACM Transactions on Computer Systems (TOCS), 2000.

[50] Ramana Rao Kompella, Jennifer Yates, Albert Greenberg, and Alex C Snoeren. Ip fault localization via risk modeling. In Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2, pages 57–70. USENIX Association, 2005.

[51] Ramana Rao Kompella, Jennifer Yates, Albert Greenberg, and Alex C Snoeren. Detection and localization of network black holes. In INFOCOM 2007. 26th IEEE International Conference on Computer Communications. IEEE, pages 2180–2188. IEEE, 2007.

[52] Teemu Koponen, Keith Amidon, Peter Balland, Martín Casado, Anupam Chanda, Bryan Fulton, Igor Ganichev, Jesse Gross, Natasha Gude, Paul Ingram, et al. Network virtualization in multi-tenant datacenters. In NSDI, 2014.

[53] Teemu Koponen, Martin Casado, Matasha Gude, Jeremy Stribling, Leon Poutevski, 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, 2010.

[54] Donald Kossmann. The state of the art in distributed query processing. ACM Computing surveys, 32(4):422–469, December 2000.

[55] Liane Lewin-Eytan, Katherine Barabash, Rami Cohen, Vinit Jain, and Anna Levin. Designing modular overlay solutions for network virtualization. In IBM Technical Paper, 2012.

[56] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P Godfrey, and Samuel Talmadge King. Debugging the data plane with anteater. ACM SIGCOMM Computer Communication Review, 41(4):290–301, 2011.

[57] Justin N Mccann. Automating performance diagnosis in networked systems. 2012.

[58] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. ACM SIGCOMM Computer Communication Review, 38(2):69–74, 2008.

[59] Rahul Potharaju and Navendu Jain. Demystifying the dark side of the middle: A field study of middlebox failures in datacenters. In IMC, 2013.

[60] Zafar Ayyub Qazi, Cheng-Chun Tu, Luis Chiang, Rui Miao, Vyas Sekar, and Minlan Yu. Simple-fying middlebox policy enforcement using sdn. In ACM SIGCOMM Computer Communication Review, volume 43, pages 27–38. ACM, 2013.

[61] Constantinos Dovrolis Parameswaran Ramanathan and David Moore. Packet dispersion techniques and capacity estimation.

[62] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. Making middleboxes someone else's problem: network processing as a cloud service. ACM SIGCOMM Computer Communication Review, 42(4):13–24, 2012.

[63] Linhai Song and Shan Lu. Statistical debugging for real-world performance problems. In Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, pages 561–578. ACM, 2014.

[64] Wenfei Wu, Guohui Wang, Aditya Akella, and Anees Shaikh. Virtual network diagnosis as a service. In Proceedings of the 4th annual Symposium on Cloud Computing, page 9. ACM, 2013.

[65] Andreas Wundsam, Dan Levin, Srini Seetharaman, Anja Feldmann, et al. Ofrewind: Enabling record and replay troubleshooting for networks. In USENIX Annual Technical Conference, 2011.

[66] Minlan Yu, Albert G Greenberg, David A Maltz, Jennifer Rexford, Lihua Yuan, Srikanth Kandula, and Changhoon Kim. Profiling network performance for multi-tier data center applications. In NSDI, 2011.

[67] Hongyi Zeng, Shidong Zhang, Fei Ye, Vimalkumar Jeyakumar, Mickey Ju, Junda Liu, Nick McKeown, and Amin Vahdat. Libra: Divide and conquer to verify forwarding tables in huge networks. In 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14). Seattle, WA: USENIX Association, pages 87–99, 2014.