Virtual Network Diagnosis as a Service

Wenfei Wu[1], Guohui Wang[2], Aditya Akella[1], and Anees Shaikh[3]

[1]Univerisity of Wisconsin-Madison , [2]Facebook , [3]IBM Research

Abstract

Today's cloud network platforms allow tenants to construct sophisticated virtual network topologies among their VMs on a shared physical network infrastructure. However, these platforms provide little support for tenants to diagnose problems in their virtual networks. Network virtualization hides the underlying infrastructure from tenants as well as prevents deploying existing network diagnosis tools. This paper makes a case for providing virtual network diagnosis as a service in the cloud. We identify a set of technical challenges in providing such a service and propose a Virtual Network Diagnosis (VND) framework. VND exposes abstract configuration and query interfaces for cloud tenants to troubleshoot their 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 overhead by performing local flow capture and on-demand query execution. Our experiments validate VND's functionality and shows its feasibility in terms of quick service response and acceptable overhead; our simulation proves the VND architecture scales to the size of a real data center network.

1 Introduction

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 [6, 10]. 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 [2], VMware/Nicira's NVP [1], and IBM DOVE [17]. 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, and, crucially, each virtual resource may map to multiple physical resources, i.e., a virtual link may map to multiple physical links. 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 this paper, 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

SoCC'13, 1–3 Oct. 2013, Santa Clara, California, USA.
ACM 978-1-4503-2428-1.
http://dx.doi.org/10.1145/2523616.2523621

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 [12], OFRewind [21], Anteater [18] and HSA [13], which expose all the raw network information. This leads to obvious scale issues, but it 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 this paper 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 and also propose the service interface to cloud tenants;

- 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 paper is organized as follows. Section 2 introduces the challenges and necessity of a virtual network diagnosis framework. Section 3 gives our VND design addressing the challenges. Section 4 presents our VND implementation. We evaluate VND feasibility in Section 5 and conclude this paper in Section 6.

2 Background

2.1 Virtual Networks in the Cloud

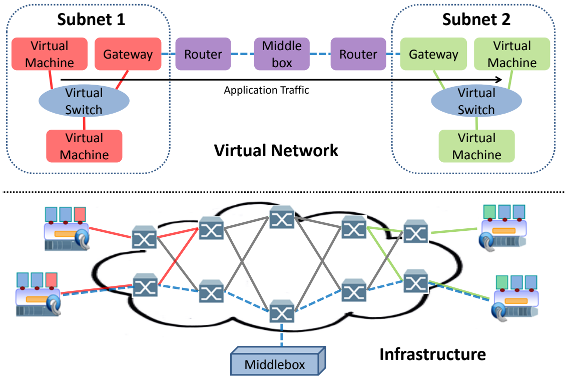


Figure 1: Virtual Overlay Networks

Figure 1 shows an example virtual network for a cloud tenant. In this example, tenant virtual machines are organized into two subnets. The virtual machines in the same IP subnets are in the same broadcast domain and they communicate with external hosts via their subnet gateway; the cloud platform can also provide network services to the virtual networks such as a DHCP server in a subnet, a load balancer or intrusion detection system on a virtual link or a firewall on a gateway. The virtual network is constructed as an overlay network running on the physical network. In a large scale cloud environment, there could be a large number of tenant networks running on the shared physical infrastructure.

The virtual machines run atop hypervisors and connect to in-hypervisor virtual switches (e.g., Open vSwitch). To decouple the virtual network from the physical network, tunnels are set up among all the virtual switches. Several tunneling techniques have been proposed to support efficient encapsulation among virtual switches, such as NVGRE, VxLAN, and STT. All tenant traffic is sent through the tunnels with different tunnel IDs in the encapsulation header used to achieve isolation between tenants.

Routing and other network services are implemented as logical or virtual components. For example, OpenStack supports routing across different networks using a virtual router function which installs a distributed routing table on all of the hypervisors. Middlebox services are implemented by directing traffic through multiple virtual or physical middlebox appliances.

2.2 Challenges of Virtual Network Diagnosis

Once the basic network is set up, configuring various aspects of the network, e.g., firewall rules, routing adjacencies, etc., requires coordinated changes across multiple elements in the tenant's 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 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 [12, 21]. 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.

2.3 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 [12], OFRewind [21], Anteater [18], HSA [13], Veriflow [14] and Frenetic [9] 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 [22] and X-Trace [8] 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 2.2

3 VND Design

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.

3.1 VND Service Operation

Figure 2 illustrates the operation 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 2.1. 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 diagnosis request to the VND control server (Figure 2(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 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 2(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 2(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. 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.

3.2 VND Architecture

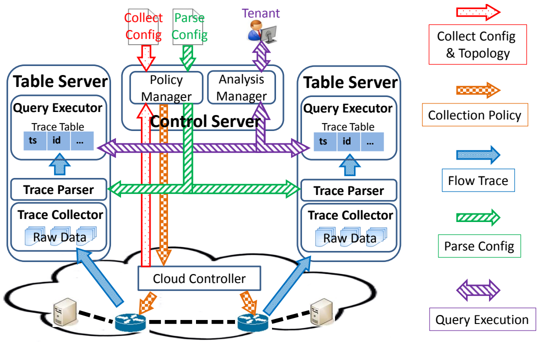
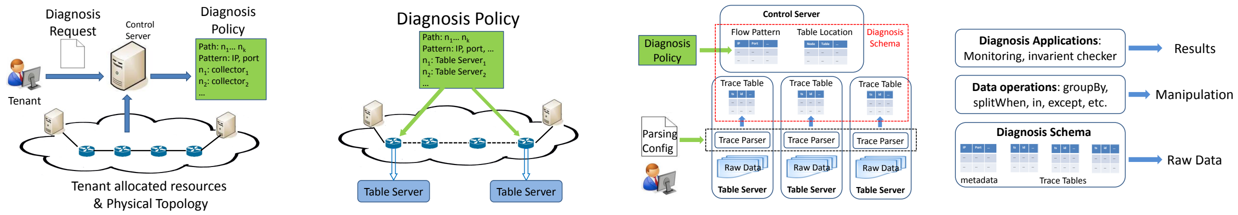


Figure 3: Virtual Network Diagnosis Framework

VND is composed of a **control server** and multiple **table servers** (Figure 3). Table servers collect flow traces from network devices (both physical and virtual), perform initial parsing, and store data into distributed



(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 2: Diagnosis as a Service operation

data tables. The control server allows tenants to specify trace collection and parse configurations, and diagnose their virtual networks using abstract query interfaces. To reduce overhead, trace collection and analysis begin only in reaction to the tenant's diagnosis requests.

3.2.1 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).

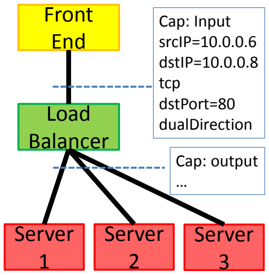
```
1) Virtual Appliance Link : l1
2)     Capture Point node1
3)         Flow Pattern field = value, ...
4)     Capture Point node2
5) ...
6) Virtual Appliance Node : n1
7)     Capture Point input, [output]
8) ...
```

Figure 4: Trace Collection Configuration format

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, it can submit a **trace collection configuration** (Figure 4) 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).

```
1)  Appliance Node : lb
2)    Cap 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) Appliance ...
12)   ...
```

(a) Configuration

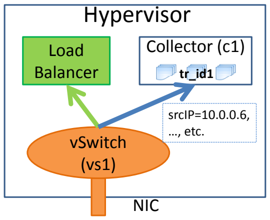


(b) Tenant's view

Figure 5: Trace Collection Example

```
1) Trace ID tr_id1
2) Pattern field = value, ...
3) Physical Cap Point vs1
4) Collector c1 at h1
5) Path vs1, ..., h1, c1
6) Trace ID tr_id2
7) ...
8) Trace ID tr_id3
9) ...
```

(a) Policy



(b) Deployment

Figure 6: Trace Collection Policy

Figure 5 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 5(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 [15] (not shown in the figure). The policy manager then computes a **collection policy** (Figure 6(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 3.4.1. 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 6(b)). We discuss how to avoid interference between diagnostic rules and routing rules in Section 3.4.2.

```
Trace ID tr_id1
Table ID tab_id1
Filter exp
Fields field_list
Table ID tab_id2
...
exp = not exp | exp and exp |
   exp or exp | (exp) | prim,
prim = field ∈ value_set,
field_list = field (as name)
   (,field (as name))*
```

(a) Configuration

```
Trace ID all
Filter: ip.proto = tcp
   or ip.proto = udp
Fields: timestamp as ts,
   ip.src as src_ip,
   ip.dst as dst_ip,
   ip.proto as proto,
   tcp.src as src_port,
   tcp.dst as dst_port,
   udp.src as src_port,
   udp.dst as dst_port
```

(b) Example

Figure 7: Parse Configuration and an Example

Cloud tenants also submit a **parse configuration** in Figure 7(a) to perform initial parsing on the raw flow trace. It has multiple parsing rules, with each rule having filter and field lists that specify the packets of interest and the header fields values to extract, as well as the table columns to store the 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 7(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 3.3.1, we discuss typical diagnosis tasks that can be easily implemented using the query interface.

3.2.2 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 3.3.1, 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.

3.3 Trace Analysis

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.

3.3.1 Diagnostic Interfaces and 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, for example.

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, and 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.

Delay at a hop: Delay time of a packet on a hop indicates the packet processing time at that hop, which indicates 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.

3.3.2 Distributed Query Execution

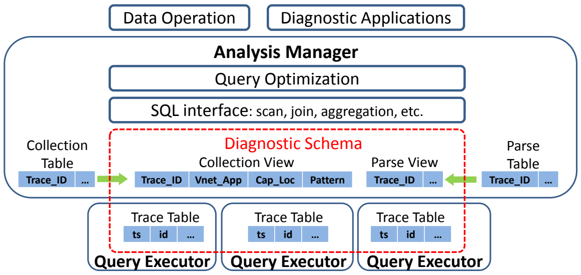


Figure 8: Analysis Framework

The data analysis framework in Figure 8 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

[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.

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 [16]. 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.

3.4 Scalability Optimizations

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

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 is scalable with the cluster and the virtual network size.

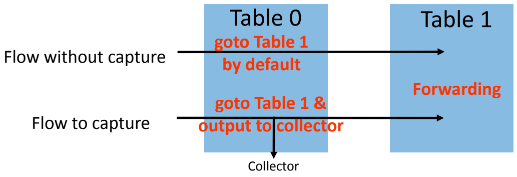


Figure 9: Flow capture with multiple tables

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 [9].

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 the monitoring rule for port 80, and also the overlapping flow space (IP, port 80) of both rules. Otherwise the switch only uses one rule for the overlap part and ignores the other. However, 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 9). 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.

3.5 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 behavior 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 [7] 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 [19]. 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 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 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. These two packets of the same flow should be in the same position in both sequences.

```
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++>
```

4 Implementation

We prototyped the VND on a small layer-2 cluster with 3 HP T5500 workstations and 1 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 3.4.2. 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.

5 Evaluation

We validate VND functions to diagnose virtual network problems, measure its overhead to the existing system and observe its performance as a service. There are two sources of overhead introduced by VND: data collection and query execution. Data collection is performed locally, so we only evaluate its effect on local hypervisor and virtual machines. Query execution is a distributed task, so we evaluate its global impact in terms of extra traffic volume and its performance in terms of response time.

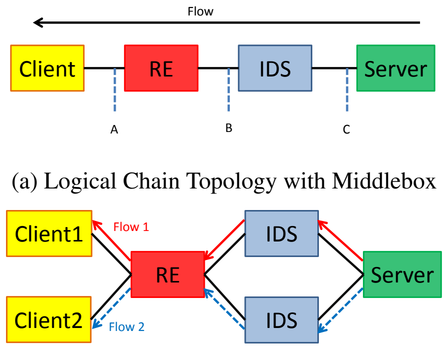
5.1 Functional Validation

The symptoms of virtual network problems are reachability issues and performance issues. VND can cope with both by analyzing the application flow trace. The reachability issue can be easily found by track packets. In this section, we focus on performance issues.

5.1.1 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 [20]. 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 [10]. 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 [11]. 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.



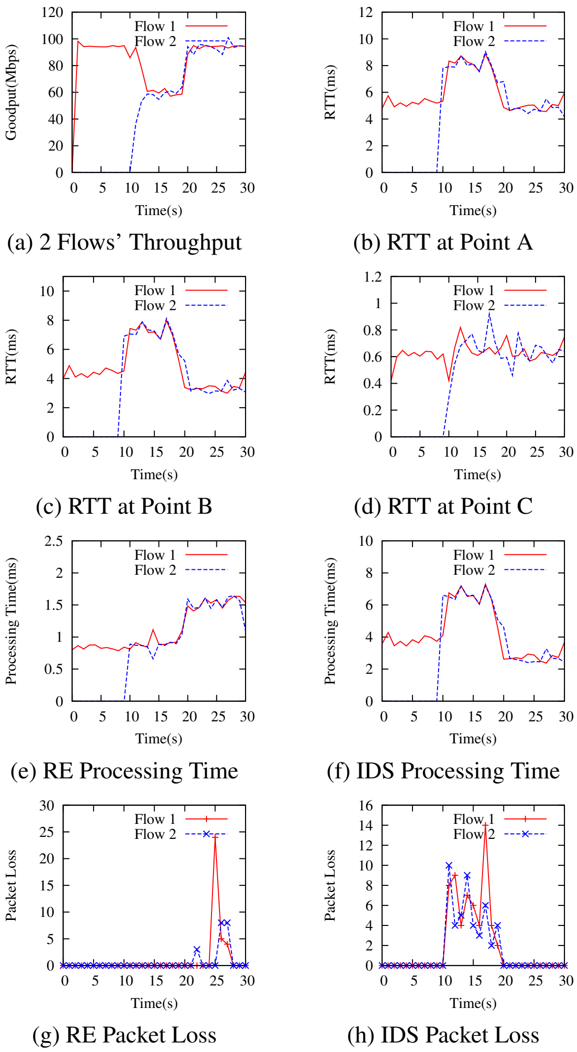
(a) Logical Chain Topology with Middlebox

(b) Actual Topology after Scaling

Figure 10: 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)[4] and an Intrusion Detection System (IDS)[3]. 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 11(a)). To find the bottleneck of the whole chain topology, 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 3.3, and check the RTT at each point. Figure 11(b)(c)(d) shows 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 11(e)(f)). We deploy the packet loss diagnostic application to observe the packet loss at each hop. Figure 11(g)(h) indicates 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 10(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



(a) 2 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 11: Bottleneck Middlebox Locating

does not impact the application performance. The logical chain topology with middleboxes is thus successfully scaled.

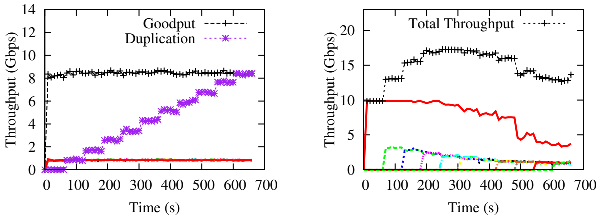
5.1.2 Flow Trajectory

We now test the methods to correlate flows as described in Section 3.5. First, we use packet fingerprints to correlate the input and output packets of middleboxes. We route a flow to traverse an RE and an 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 connection built time of 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 this time sequence. We start 4000 iperf client flows to 10 iperf servers via a load balancer named 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.

5.2 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

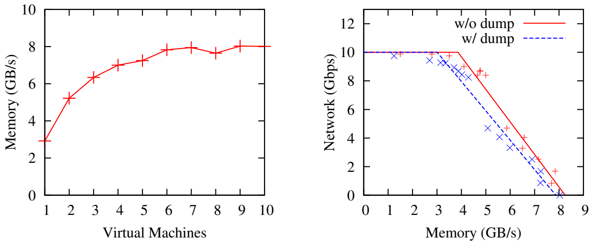
*Curves without a legend are TCP flows

Figure 12: Network Overhead of 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 12(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 12(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.



(a) VM Memory Throughput (b) Network-Memory

Figure 13: Memory Overhead of Trace Collection

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 13(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.

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 13(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

Table 1: 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

$$N + 2.28M = 18.81,$$

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

5.3 Data Query

We use the trace in the bottleneck middlebox detection experiment (Section 5.1.1). We monitor throughput, RTT and packet loss, and observe its overhead and performance in terms of response time. These three diagnostic applications represent different data 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.

Data 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.

5.3.1 Overhead

Storage: At each hop, the total traffic volume is 0.5GB, so the total size of all traces is 1.5GB. After the traces

Table 2: 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: 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

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 1 and 2 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 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.

5.3.2 Performance

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.

5.4 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 data collection from a large number of VMs and support diagnosis requests for a large number of tenants. Since VND co-locates table servers with tenants' VMs and only performs data collection locally, data collection will not be the scalability bottleneck when there is a large number of VMs. The **control server** generates data 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 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 [6]. 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 [6].
- In each physical server, the virtual switch can process up to 18 Gbps network traffic (Section 5.2).
- 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 5.3).

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 10000 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

Table 4: 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

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.

Data 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 [5], 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.

Data Query: In Table 4, 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. Given that link utilization in the typical data centers is normally lower than 30% [5], so most tenants' traffic can be queried at real time. 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.

6 Conclusion

In this paper, 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.

References

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

[2] www.openstack.org.

[3] www.snort.org.

[4] A. Anand, V. Sekar, and A. Akella. Smartre: An architecture for coordinated network-wide redundancy elimination. In *SIGCOMM*, 2009.

[5] T. Benson, A. Akella, and D. Maltz. Network traffic characteristics of data centers in the wild. In *IMC*, 2010.

[6] T. Benson, A. Akella, A. Shaikh, and S. Sahu. Cloudnaas: A cloud networking platform for enterprise applications. In *SOCC*, 2011.

[7] S. K. Fayazbakhsh, V. Sekar, M. Yu, and J. C. Mogul. Flowtags: Enforcing network-wide policies in the presence of dynamic middlebox actions. In *HotSDN*, 2013.

[8] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stocia. X-trace: A pervasive network tracing framework. In *NSDI*, 2007.

[9] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A network programming language. In *ICFP*, 2011.

[10] A. Gember, A. Akella, T. Benson, and R. Grandl. Stratos: Virtual middleboxes as first-class entities. In *UW-Madison, Technical Report*, 2012.

[11] A. Ghodsi, V. Sekar, M. Zaharia, and I. Stoica. Multi-resource fair queueing for packet processing. In *SIGCOMM*, 2012.

[12] N. Handigol, B. Heller, V. Jeyakumar, D. Mazieres, and N. McKeown. Where is the debugger for my software-defined network. In *HotSDN*, 2012.

[13] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *NSDI*, 2012.

[14] A. Khurshid, W. Zhou, M. Caesar, and P. B. Godfrey. Veriflow: Verifying network-wide invariants in real time. In *HotSDN*, 2012.

[15] T. Koponen, M. Casado, M. Gude, J. Stribling, L. Poutevski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: A distributed control platform for large-scale production networks. In *OSDI*, 2010.

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

[17] L. Lewin-Eytan, K. Barabash, R. Cohen, V. Jain, and A. Levin. Designing modular overlay solutions for network virtualization. In *IBM Technical Paper*, 2012.

[18] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. Debugging the data plane with anteater. In *SIGCOMM*, 2011.

[19] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu. Simple-fying middlebox policy enforcement using sdn. In *SIGCOMM*, 2013.

[20] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making middleboxes someone else's problem: Network processing as a cloud service. In *SIGCOMM*, 2012.

[21] A. Wundsam, D. Levin, S. Seetharaman, and A. Feldmann. Ofrewind: Enabling record and replay troubleshooting for networks. In *ATC*, 2011.

[22] M. Yu, A. Greenberg, D. Maltz, J. Rexford, and L. Yuan. Profiling network performance for multitier data center applications. In *NSDI*, 2011.