RINC: Real-Time Inference-based Network Diagnosis in the Cloud

Mojgan Ghasemi
Princeton University
mojgan@cs.princeton.edu

Theophilus Benson
Duke University
tbenson@cs.duke.edu

Jennifer Rexford
Princeton University
jrex@cs.princeton.edu

Abstract

Cloud tenants experience performance problems due to issues within their virtual machines (VMs) or within the cloud infrastructure. To offer good and predictable performance, cloud providers must be able to detect and diagnose performance problems in real time. However, existing cloud diagnosis techniques are either unable to detect problems in the tenant's VMs or are too costly. We argue that rather than collecting all statistics, cloud diagnosis should proceed in phases, with each phase selectively collecting heavier weight measurements. To this end, we introduce a set of novel techniques for inferring the internal state of a VM's midstream network connections which allows us to accurately collect measurements at any point during a connection. Our framework, RINC, runs within the hypervisor, using these techniques to selectively monitor a tenant's connections.

RINC provides a simple query interface to its cloud-wide platform that allows cloud operators to easily write diagnosis applications. We evaluate RINC on a testbed and with a simulator using a combination of real data center traces and synthetic workloads. Our evaluations validate RINC's accuracy and show that, by being selective, RINC is able to scale to a cloud with 100K physical servers or 1Million VMs. Moreover we demonstrate RINC's flexibility and expressibility by implementing five diagnosis applications.

1 Introduction

Diagnosing network performance problems in real time is essential for offering good, predictable service in the cloud. Performance problems can be caused by many different components across the cloud's networking stack (e.g., the sender or receiver virtual machine (VM), the hypervisor, or the network). Therefore, solely collecting network-level statistics, e.g., link utilization, is not sufficient to diagnose performance problems. Unfortunately, the alternative, instrumenting the tenant VMs to collect statistics, can be rather invasive and cumbersome. Instead, in this paper, we argue for an orthogonal approach wherein the hypervisor passively collects lightweight statistics about each tenant's connections. Upon detecting anomalous behavior, the provider can selectively enable heavier-weight monitoring to pin-point the root cause of problems.

1.1 Performance Diagnosis in the Clouds

Public Infrastructure as a Service (IaaS) clouds provide tenants with virtual machines. Unfortunately, the sharing of resources in the cloud often leads to performance problems [34, 31]. Diagnosing and detecting these problems is complicated by the fact that performance anomalies are sometimes caused by tenant-specific problems, such as the configuration of the tenant's operating system or application [14]. Furthermore, server virtualization complicates performance diagnosis because the information needed to accurately diagnosis performance problems resides partially within the tenant VM's networking stack.

Traditional approaches to diagnosing performance problems often collect statistics inside the network, including packet traces [37, 11] or link utilization from network devices. To be effective, these approaches require continuous, fine-grained monitoring from all network devices, leading to significant scalability challenges. In addition, in-network monitoring does not offer visibility into end-to-end metrics (e.g., round-trip times) and application behavior, making it difficult to detect problems within the tenant's VM.

In response to these issues, recent diagnosis techniques collect richer traffic statistics from the end hosts. Unfortunately, these introduce trust and scalability challenges. Some solutions, like Web10G [7], HONE [30] and SNAP [35] address scalability and precision by deploying a kernel-module within the VM. While appropriate in private clouds, these techniques are intrusive in IaaS clouds since they interfere with a tenant's ability to manage his own VM. In addition, they make the cloud provider rely on tenants for updates of the system (e.g., to install patches) and complicate billing by using a tenant's resources for measurement. Other approaches side-step these trust issues by collecting packet traces in the hypervisor. For example, VND [33] stores packet headers in a SQL-database and executes SQL queries to detect problems. However, the database introduces significant scalability challenges due to the large amount of storage, and the approach is limited to

diagnosing problems that can be detected using queries on information stored in packet headers, e.g., RTT and loss rate but not sender or receiver side problems.

1.2 Light-weight, Selective Measurement

Detecting performance problems is much easier than diagnosing them. As such, we believe that measurement should proceed in multiple phases, with each phase narrowing in on the root cause through heavier-weight monitoring and analysis on a smaller subset of the connections. For example, in the first phase, the provider could collect simple statistics (e.g., throughput) to identify troubled connections. Then, in successive steps, the provider could enable heavier-weight measurement of these connections, e.g., tracking the evolution of the TCP state machine and collecting round-trip time (RTT), congestion window (CWND), receiver window (RWND), and maximum segment size (MSS). Finally, this detailed information can be used to pin-point the location of the problem. For example, the provider could determine if an application has data waiting in the TCP send buffer. If the application is back-logged, the provider can collect more detailed statistics for the paths carrying these flows, to detect (possibly intermittent) congestion; if not, the provider could infer that performance is constrained by the sending VM or application.

This simple example highlights the challenges involved in diagnosing and pin-pointing the root cause of performance problems. These challenges arise because performance problems can occur either due to problems in the provider's infrastructure or tenant's VM, thus there is a need to collect state both about the provider's domain, e.g., the network, and the tenant's domain, e.g., TCP state-machine. However, if not done wisely, collecting all statistics from both domains can be cumbersome, thus limiting the adoption of the system.

We argue that a scalable and effective cloud diagnosis system must provide:

- **Connection-level statistics**: Since performance problems may be caused by the provider or the tenant, we must efficiently and accurately infer the internal state of each of the tenant's connection.

- **Cloud-wide measurement and analysis**: Cloud networking infrastructure spans both the network devices and the end hosts, so we must be able to collect and correlate measurements across the entire infrastructure. To avoid low-level manual configuration, we should give cloud operators a high-level query interface to control *when* and *how* to perform more heavy-weight monitoring.

- **Monitoring midstream connections**: Since heavy-weight monitoring could be enabled at any time, our solution must be able to quickly infer the state of a connection without seeing all its packets from the beginning.

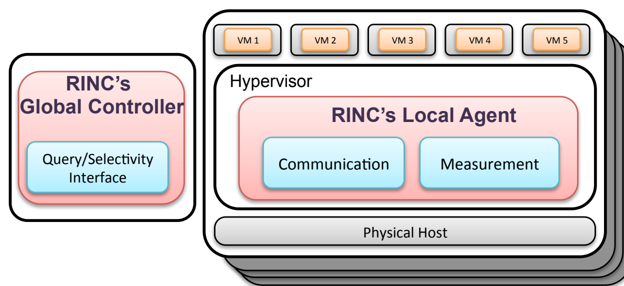


Figure 1: RINC Architecture

To that end, we present RINC (**R**eal-Time **I**nference-based **N**etwork diagnosis in the **C**loud), a framework that enables cloud operators to diagnose problems in real time.

1.3 RINC

RINC's architecture, displayed in Figure 1, consists of a *Local Agent* which runs in the *hypervisor* of each server, and a *Global Coordinator* that aggregates and coordinates measurements across the agents.

Local agent: The *Local Agent* contains two components. First, the *measurement module*, which inspects packets, calculates statistics, and reconstructs each TCP connection's internal state, obviating the need to patch the tenant's VMs. Second, the *communication module* manages communication with the *Global Coordinator*, by translating queries into instructions for enabling data collection and returning the results.

Global coordinator: The *Global Coordinator* aggregates information from the *Local Agents* and, more importantly, hides the details of RINC's distributed implementation. To enable scaling, the *Global Coordinator* provides a flexible interface that provides the cloud operator with a set of mechanisms to minimize the *Local Agent*'s overhead: the first allows operator to directly specify which statistics are important, thus implicitly limiting the set of statistics being collection; and the second automatically determines the minimum set of connections to monitor in order to accurately identify problems with a specific network predicate, e.g., switch, link, hypervisor, or VM.

Roadmap: The next section presents a brief overview of TCP and the challenges of reconstructing the internal state. Section 3 motivates, discusses the challenges of, and presents solutions to monitoring midstream connections. Section 4 presents the distributed architecture of RINC, including the query interface used by operators to control RINC. We present our prototype in Section 5 and evaluation in Section 6. Section 7 discusses related work. Sec-

tion 8 discusses generalizing RINC to arbitrary versions of TCP. We conclude the paper in Section 9.

2 Background on TCP Monitoring

The transport protocol employed by an application significantly impacts its performance. Although there a number of transport protocols, most cloud applications are designed to use TCP, with TCP accounting for 99.91% of the traffic [9]. As such, we focus on developing techniques for inferring TCP connection statistics. In this section, we describe the life of a TCP connection, map this behavior to the TCP state machine, and discuss the challenges in inferring the statistics of a midstream connection, particularly the connection state.

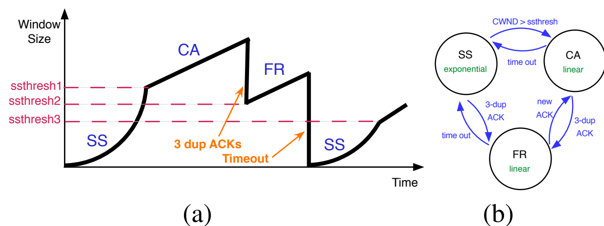
2.1 The Life of a TCP Connection

A TCP connection starts with a three-way handshake where the end-points exchange options to agree on various parameters for the connection, such as the maximum number of bytes a packet can contain (maximum segment size (*MSS*)).

After set-up, the connection begins in the slow-start state (*SS* in Figure 2(a)), where the connection initially sends packets very slowly but tries to learn the available bandwidth by transmitting at an exponentially increasing rate. Upon receiving each packet, the receiver sends back an *ACK* packet. Upon receiving the *ACK*, the sender sends out more packets. TCP remains in slow start until either a loss happens (indicated by receiving three *duplicate-ACKs* or by receiving no *ACKs* before a retransmission timer expires (*RTO*)), or the sender sends a predefined number, *SSThresh*, of bytes (indicated by the red dashed lines in Figure 2(a)).

At this point, the TCP sender assumes it has discovered its fair share of the bandwidth and transitions into the congestion avoidance (*CA*) state. In this state, TCP increases its sending rate linearly (indicated as *CA* in Figure 2(a)). A packet loss can trigger one of two transitions, depending on how the loss was detected. If a coarse-grained loss is inferred from *RTO*, the connection resets the window size to *initial window* (usually one MSS) and returns to the slow-start state. Otherwise, if the loss is detected through three *duplicate-ACKs*, the connection transitions into fast recovery (*FR*), where TCP sets its window size to half the current value and resends lost segments.

The TCP State Machine (summarized in Figure 2(b)) dictates both the conditions for state transitions and TCP's sending behavior in each state. More concretely, the TCP state machine determines when a TCP sender is eligible to transmit more packets. The state variable Congestion window (*CWND*) determines how much the TCP can send without congesting the network; the *CWND* is adjusted every Round Trip Time (*RTT*) for the duration of a connection's lifetime.



(a) (b)

Figure 2: (a) Congestion window, (b) Reno state machine

To this end, TCP randomly picks a packet each *RTT* and calculates its *RTT*. Generally, *CWND* is increased when a new *ACK* is received and it is decreased when a packet is lost. Figure 2(a) displays the evolution of *CWND* over the life time of a sample TCP connection.

2.2 Challenges of Inferring TCP Statistics

Network performance diagnosis can draw upon four main types of statistics about TCP connections:

- **Constants**, such as *MSS*, which are exchanged at the beginning of the connection.
- **Counters**, such as the number of bytes sent, which are updated on every packet.
- **Sampled statistics**, such as *RTT*, which are updated less frequently.
- **State variables**, such as *CWND* or current TCP state, which track the evolution of the connection.

A full list of these statistics are presented in Table 1. RINC must infer these statistics by analyzing the packets seen at the hypervisor, without the benefit of having access to the VM's network stack.

These statistics are particularly useful for developing diagnosis applications. For example, to detect heavy hitters, that is flows that are larger than a certain number of bytes, an operator would instruct RINC to collect *bytes sent (a Counter statistic)* for each connection. Similarly, to diagnose performance problems, such as finding high-latency flows, the operator would configure RINC to collect *RTT (a Sampled statistic)*. There are two main challenges in collecting these statistics, namely:

The **overhead** incurred by RINC for collecting, processing, and storing these statistics. In Figure 3, we present the memory overhead of collecting all statistics and compare this against the overhead of selectively collecting data only for one type of statistic. In Figure 3, we observe that naively collecting all statistics can result in more than a 75% increase in memory compared to selectively collection only the required group of statistics. For example, collecting

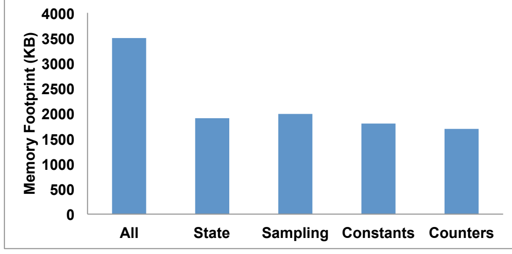


Figure 3: Local Agent memory overhead per module (10K connections per host)

Module	Statistics
Constants	SMSS, RMSS, window scale, Initial sequence number, creation time(SYN) and finish time(FIN)
Counters	packet and byte counters, number of duplicate acks, packet re-transmissions, and timeout counters
State Tracker	TCP state (SS, CA, FR, UNKNOWN), CWND, ssthresh, RWND, Initial Window, Restart Window, and flight size
Sampling	last update time, SRTT, RTTVAR, RTO, sending rate, and throughput

Table 1: RINC's measurement modules, and their major statistics

statistics for heavy-hitters detection (a Counters statistics) requires only half of the memory required for collecting all statistics. Similar trends are observed for processing and networking overheads. (We elaborate on these in Section 6.)

These observations highlight the need for a mechanism that determines *which* statistics to monitor and *when* to collect them. We discuss this mechanism in Section 4.

Detecting **TCP version and configuration parameters** is integral to the process of inferring the state of the connection, as the state machine and the formulae for calculating these variables depend on the version of TCP and TCP's initial configuration parameters (e.g., *SSThresh* and *initial window*). Fortunately, we can determine these parameters by running a set of techniques to finger-print the OS and the TCP version [5, 37]. Alternatively, the cloud provider may have its own internal knowledge about the tenant's VM image, often chosen from a menu of options.

3 Tracking Midstream Flows

Data center connections are mostly long-lived [35] flows, and the number of connections per host can be up to 10K [35] and increasing. Monitoring all the long-lived connections at all the times can be expensive and impractical. To reduce the overhead of collecting and storing their statistics, a cloud provider can selectively collect a small set of statistics and collect more statistics on demand. However, this relies on having an effective way to monitor a *midstream* connection; that is, a connection where we have missed an arbitrary number of initial packets.

Counters and *Sampled* statistics represent cumulative values or long running averages and, as such, cannot be accurately inferred for midstream flows. Fortunately, sampled statistics such as smoothed RTT (SRTT) will converge with the correct values eventually (we will show this in 6.2). In addition, counter statistics such as packet counts demonstrate the relative growth in values since the start of monitoring, which can be helpful in diagnosis of performance problems.

The only way to infer *Constants* is to monitor a TCP connection during the handshake. Lacking this handshake, we can leverage the fact that all connections from a specific VM use the same constants. Thus we can inherit constants from other connections from the same VM.

Finally for *state variables*, unfortunately, the existing techniques [37, 11] for inferring the internal state of a TCP state-machine are only applicable if the TCP connection is monitored from the beginning. For connections that are monitored midstream, inferring the state of the TCP state machine becomes significantly more difficult, for several reasons:

- **Thresholds change:** The thresholds that are generally considered to be constant actually change as the connection evolves over time. For example, Figure 2 (a) shows how *SSThresh* changes with each state transition.
- **Different states can behave identically:** TCP behaves identically in several different states. For example, as shown in Figure 2(a), TCP behaves identically in *CA* and *FR*, linearly increasing its sending rate. Furthermore, a connection in *SS* that does not have enough application data to send could incidentally have a linear sending rate, consistent with the *CA* or *FR* state.
- **Transitions are often triggered by a sequence of events:** For example, if RINC starts monitoring a connection after the first of the 3-*duplicate-ACKs* has been received, it may incorrectly infer the connection is in *CA* state even though the connection transitioned to *FR* state.

Therefore we need to develop novel techniques that allow us to accurately and quickly infer the internal state of a midstream TCP connection. These techniques need to detect new the values for the constants and to disambiguate states with similar behaviors.

Our algorithms for inferring the internal state of midstream connections builds on the insight that once we can

determine the current state of the TCP state-machine, existing techniques can be easily applied. However, as stated before, the challenges in detecting this initial state lie in the ambiguity of the state-machine. To overcome this ambiguity, we leverage three observations: first, certain states and transitions are unambiguous (Section 3.1); second, a certain level of ambiguity is created by the applications behavior, thus inferring the application's effect further decreases ambiguity (Section 3.2); and finally, when we cannot infer the state, it is possible to apply an *active approach* which uses careful probing to force a connection into one of several known states (Section 3.3).

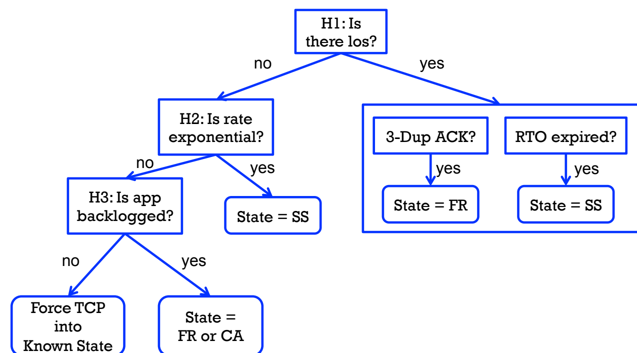


Figure 4: Algorithm for inferring the state of midstream TCP connections

3.1 Unambiguous State Transitions and Behavior

Our first technique, indicated as **H1** and **H2** in Figure 4, leverages the unambiguous behavior, namely of exponential sending in *SS*, and unambiguous state transitions, namely of *RTO* to *FR* and of three *duplicate-ACKs* to *SS*.

The first step, **H1**, is to determine if we have fully observed any events that unambiguously trigger a state transition: we check to see if we've observed three *duplicate-ACKs* or an *RTO* event (using the estimated RTO calculated by the sampling module according to Karn's Algorithm [27], or using the worst case RTO of the cloud connections if no samples exist for the flow) indicating a loss, in which case we know TCP will transition into *FR* and *SS* respectively.

By virtue of starting midstream, our approach has missed certain packets. A subtle implication of this is that we may be unable to detect these transitions because we may have missed one or more of the *duplicate-ACKs*.

Next, **H2**, checks to see if TCP is in one of the unambiguous states: **H2** checks to see if TCP has an exponential sending rate, thus inferring it is in SS. Comparing the relationship between *ACKs* and the data packets to see how many packets the sender transmits after a new *ACK* can do this.

3.2 Application Sending Behavior

Our second algorithm leverages the insight that further ambiguity in the TCP's behavior may be in fact symptomatic of application limitations. More concretely, a TCP connection in *SS* will have a non-exponential sending rate if the application has less data to send than TCP allows it to send. We label such connections as "non-backlogged".

A connection is backlogged if the application has more data to send than TCP will permit. This implies that the packets will be sent once *CWND* permits; when an *ACK* triggers an increase in *CWND*, the TCP connection will immediately send out a new packet. To this end, we define a TCP connection as backlogged if it sends the maximum-sized packet *immediately after* receiving an *ACK*. *Immediately after* is defined as an empirically derived threshold. Thus, if packets are sent within two standard deviations of this threshold, then we infer that the connection is backlogged.

The next part of our algorithm, **H3**, checks to see if a connection with a non-exponential sending state is back-logged or not. If back-logged, then TCP is either in fast recovery or congestion avoidance. However, if the connection is not backlogged then the connection can be in any of TCP's three states.

3.3 Active Approaches: Triggering a Known State Transition

Finally, in dire situations where the state cannot be inferred by TCP's actions or by events, we can **actively** force TCP into a known state. The TCP state machine, presented in Figure 2 (b), shows that regardless of the current state, when TCP experiences three *duplicate-ACKs* or a *RTO* event the state machine transitions into *FR* or *SS* respectively. Thus, we force TCP into either *FR* or *SS*, by having the *Local Agent* simulate either event. To **actively** force TCP into a known state, RINC employs one of the following algorithms:

Forcing TCP into *FR*: the *Local Agent* replays the last *ACK* received 3 times, thus forcing TCP to believe that three *duplicate-ACKs* have been received and that a transition to *FR* is required.

Forcing TCP into *SS*: the *Local Agent* temporarily buffers all *ACKs* for an *RTO* amount of time (using the default value from Karn's Algorithm [27] or the average RTO of the cloud connections); therefore, forcing TCP to think that none of the packets were acknowledged and thus lost. In response to this, TCP transitions into *SS*.

Forcing TCP into either *FR* or *SS* impacts the performance of the connection by forcing it to decrease its *CWND* and thus sending rate, as shown in Figure 2 (a). We argue

that this performance cost is manageable because these *active* approaches are only applied if a connection is already showing signs of performance problem, second, the cloud provider only picks a handful of connections from a set of similarly troubled connections (based on service or host) for active approaches. Furthermore, the cloud provider can define a timer as the maximum time to wait for one of the unambiguous events to happen, after that, if the information is still not available, the cloud provider can use active approaches.

4 Cloud-Wide Platform

Modern cloud hypervisors are designed to be minimal and often restricted in memory and CPU. Unfortunately, existing cloud and data-center diagnosis frameworks [7, 35, 33] either collect all measurements for all flows and thus require extensive CPU and memory, or collect no measurements. However, in reality, diagnosis may require only a subset of the available measurements.

To this end, we have developed a *selectivity* interface that provides cloud operators with the ability to adjust the scope of the measurements being collected by the *Local Agents* based on the properties of the connections (including the TCP five-tuple, and other properties such as the tenant, application, middleboxes and links associated with the connection which we will refer to as its *network predicates*) or the type of statistics. This selectivity interface helps RINC to manage the CPU and memory consumption of the *Local Agents*. Furthermore, to ensure efficiency, RINC employs a *distributed data structure* across the cloud with each *Local Agent* storing a hash table of its connections and their statistics.

This query and selectivity interface is exposed to the cloud operators by the *Global Coordinator* as a set of primitives. Next, we identify the requirements for these primitives and exemplify their flexibility by using them to develop five diagnosis applications.

4.1 RINC's Selective Measurement

RINC has a selectivity and query interface, as represented in Table 2, which is an SQL-like interface allowing the cloud operator to write diagnosis applications. To understand the various dimensions of selectivity that RINC provides control over, we return to the earlier example of trying to diagnose performance problems. As discussed in section1.2, the first step is to detect connections with performance problems. Thus the operator needs to find the throughput of the connections in the cloud and then use that as a criteria. This highlights the need for having **selective collection of statistics**, which allows us to express which statistics need to be collected. This is specified through the keyword *Select* as shown in the selectivity and query interface represented in Table 2. The challenge in specifying selectivity lies in the fact that there are intricate dependencies between certain statistics. For example, to calculate the CWND, other statistic such as TCP state, ssthresh and flight size are needed. To overcome this, RINC allows operators to specify any of the statistics of their interest without worrying about dependencies, and it automatically gathers the necessary statistics. Next phase of the example is focusing on the subset of connections that seem problematic (low throughput). Hence, the operator wishes to focus only on this given set of troubled connections. This highlights the need for having **selective monitoring of connections**. The set of flows to be monitored can be specified explicitly through the keyword *On* along with an expression that specifies the set of flows to monitor. The expression language is consistent with tcpdump [2].

Finally, to help the cloud provider in discovering potential problems in the cloud (such as the hypervisor [34], middleboxes, link failures, or tenant misconfigurations [13]) a fault-localization system is needed to narrow down the scope to a set of candidate predicates. If a hypervisor, middlebox, or link is faulty, most connections passing through it will experience performance problems, so only monitoring one such flow is enough to tell us if the component is likely to be faulty. This highlights the need for a **light-weight fault-localization mechanism** that provides *coverage* over a set of network predicates. (Section 4.1.1)

4.1.1 Selectively Covering Network Predicates

RINC automates the coverage problem explained above by offering an interface (using the keyword *cover*) for the cloud operator to have light-weight coverage on a set of network devices and components. RINC allows the operator to select network predicates of interest, such as links, middleboxes, end-hosts, tenants, or applications. Given this selection, RINC discovers the minimal set of connections that needs to be monitored in order to cover that predicate. To accurately select these connections, the *Global Coordinator* needs to know the forwarding policy and the network topology.

Selecting the minimum set of connections that can cover a predicate is analogous to the minimum set cover problem which is an NP-hard problem. Motivated by recent work on localizing faults [36], we apply a similar greedy heuristic to solve the set-covering problem.

Unfortunately, running this heuristic once may be ineffective as the set of active flows change over time and thus the set of flows needed to diagnose a network predicate may change. To account for this, we allow the cloud operator to specify an *update time* which determines how frequently the heuristics are rerun.

Using this mechanism, cloud operator can use RINC's se-

lectivity interface to find the root cause of problems by first limiting the problem to a set of candidate predicates using *cover*, and then focus more on diagnosis of each potentially faulty item using selectivity mechanisms explained above.

4.2 Example Diagnosis Applications

In addition to the keywords discussed above as part of the selectivity, the interface also supports two other keywords: *Every* and *For*. These keywords will free the cloud provider from having to submit a query again, and will automatically send updates to the same query for as many times as requested with the provided period. If not explicitly specified, the query will only get executed once.

To demonstrate the flexibility and generality of RINC as a diagnosis tool, in this section, we use RINC's *selectivity* and *query interface* to develop five canonical applications. In Table 3, we present the instructions used to implement each of them.

Detecting Long-Lived Connections: The purpose of this application is to find the connections that have lasted longer than a threshold value. The *Local Agent* monitors all TCP connections, calculating their duration and then returning a list of flow keys that have a duration more than the threshold (e.g., 10 seconds). RINC's *Global Coordinator* will aggregate responses and present it to cloud operator.

Detecting Heavy Hitters: The purpose of this application is to find the connections that have sent more than a threshold number of bytes (e.g., 5000). The *Local Agent* monitors all TCP connections and for each, collects the number of bytes sent. Finally, the list of connections matching the criterion is created and sent back to *Global Coordinator*.

Traffic Counter: The purpose of this application is to find out how many distinct source IPs send traffic to a specific destination IP (e.g., 1.2.3.4). First, all the connections to this destination are monitored, then we count the number of distinct source IPs contacting this IP. Note that no information for monitored flows need to be collected and only flow tuples suffice for answering this query.

Detecting Super Spreaders: The goal of this application is to find the set of source IPs who contact more than K (a constant number) distinct destination IPs. To do so, first all connections in the cloud are monitored to get a list of distinct source IPs (round 1). Next, for each of these distinct source IPs, a query is submitted to count the number of distinct destination IPs (round 2). If this number is more than the specified threshold, the source IP is a super spreader and is thus returned.

Root Cause Analysis of Slow Connections: The purpose of this diagnosis application is first, finding troubled connections and then, performing root cause analysis on them. The *Local Agent* monitors all the connections collecting their sending rates to find the subset of troubled connections with a limited sending rate (round 1). For each of these these troubled connections, the application collects more heavy weight statistics such as RTT, CWND and RWND (round 2) to find out if the limiting factor is network, sender, or receiver.

5 Implementation

In Figure 5, we present RINC's architecture. We have implemented an initial prototype of RINC to run in Xen-hypervisor; our *Local Agent* can run along side the virtual-switch in the driver domain.

The *Local Agent* is implemented in approximately 10K lines of C code, using the libpcap [2] interface to intercept packets. We implemented the *active approaches*, using ipables [1], NetFilter, and netfilter_QUEUE APIs [4] for buffering and duplicating packets. The *Local Agent* is implemented in two threads: The first, implementing the *measurement module*; and, the second, implementing the *communication module*.

The *communication module* communicates with the *Global Coordinator* through a persistent TCP connection. The *communication module* interacts with the *measurement module* via a shared memory structure, the *measurement module* has write access to the connection table, hashing the packets and updating their corresponding connection's statistics, while the *communication module* only has read access to the connection table to respond to queries.

The *measurement module* consists of several submodules: for all connections that have been specified for RINC to monitor through the selectivity interface, the *Basic* module maintains the base data-structure, tracking information about the types of statistics being collected for the connection. Fortunately, collecting these statistics incurs little additional overhead as modern hypervisors already perform per-packet operations like packet forwarding. The *Basic* module acts as a manager for the heavier-weight modules.

The heavy-weight modules include those used to collected the statistics described in Table 1: *Counters*, *Constants*, *State Tracker*, and *Sampling*; and a module, *Initializer*, which runs heuristics described in Section 1 for inferring the current state of the TCP state-machine for midstream connections. The *Basic* module activates the *Initializer* when *State Tracker* is requested for a midstream connection.

Global Coordinator is implemented in approximately 500 lines of Python. In our current implementation, the *Global Coordinator* is assumed to have access to the topology. The *Global Coordinator* contains a *Query interface* module which accepts queries in a flexible SQL-like format from the network operator, coordinates the data collection across the cloud by determining which *Local Agents* to dispatch measurement instructions to, and aggregates the re-

Query	:=	Select(Stats) $\star$ On(Filter) $\star$ Where(Criterion) $\star$ Every(Interval) $\star$ For(Times)
Stats	:=	flowKey(srcIP, dstIP, srcPort, dstPort) $\mid$ Stats from Table 1. A flowKey query may be proceeded by *Distinct* and/or *Count*.
Criterion	:=	Stats $\star$ Sign $\star$ Value
Filter	:=	tcpdump expression [2] $\mid$ Cover(Predicate, ID list, Update cover)
Sign	:=	$> \mid < \mid >= \mid <= \mid == \mid ! =$
Predicate	:=	Links, Tenants, Applications, Hosts, Middleboxes
ID list	:=	Integer list of predicate's IDs to be covered
Update cover	:=	Period of checking covered predicates in seconds (Double).
Interval	:=	Update time period in seconds (Double).
Times	:=	Number of times the query will get executed (Integer).

Table 2: RINC's Selectivity and Query Interface

Application	Implementation in RINC
Long-Lived Connections	Select (flowKey) Where (durations > 10) On (tcp)
Heavy Hitter Connections	Select (flowKey) Where (total _bytes_sent >= 5000) On (tcp)
Traffic Counter	Count Distinct Select (srcIP) Where (dstIP==1.2.3.4) On (tcp)
Super Spreaders	**Round 1** list= Distinct Select (srcIP) Where (dstIP==1.2.3.4) On (tcp) **Round 2** for IP in list: 2. Count Distinct Select (dstIP) Where (srcIP == IP)
Root Cause Analysis of Slow Connections	**Round 1** list = Select (flowKey) Where (sending_rate < 10) On (tcp) **Round 2** for flowKey F in list: Select (RTT, CWND, RWND, state, MSS) Where (flowKey==F)

Table 3: Example Applications Using RINC's Interface

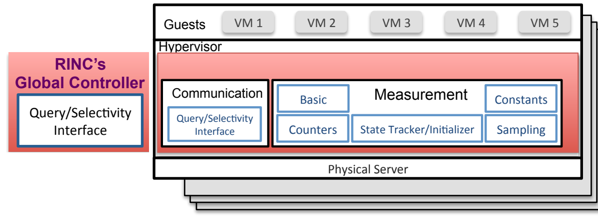


Figure 5: RINC Architecture

sults from these *Local Agents* before returning them to the operator.

6 Evaluation

In this section, we present our evaluation of RINC on two small testbeds, in a simulated data center, and with packet traces from a real data center. In evaluating RINC, we aim to answer the following questions:

- How accurate are RINC's algorithms for inferring the internal state of midstream connections? How long do these algorithms take to converge on the correct state? (Section 6.2)
- Does RINC scale to large clouds? Are the network, memory, and CPU overheads acceptable? (Section 6.3)
- How efficient is RINC at calculating various statistics? (Section 6.4)
- How does RINC compare against related work (VND [33])? (Section 6.5)

6.1 Experimental Setup

Testbed1: We evaluate RINC's accuracy and overheads on a small testbed with four servers interconnected by 1Gig links through a Pronto switch. Each server is configured with 16GBs of memory, twelve 2.0 GHz cores, and runs Ubuntu 13.10 with TCP Reno congestion control algorithm. For verification experiments against Web10G, we patched and recompiled the kernel to run Web10g-2.0.7.

Testbed2: To evaluate RINC's performance, we use a small testbed with three servers interconnected by 1Gig. Each server is configured with 8GBs of memory, eight 2.3 GHz cores, and runs Fedora 14.

Packet Traces: To understand the efficiency of our algorithms, we run RINC against a two-hour data-center trace [12]. The data center contains 500 servers with an over-subscription ratio of 2:1; each server is configured with RHEL 4 and TCP Reno.

Simulation and Simulated Workloads: We developed a simple simulator that models the resource consumption of a *Local Agent* thus allowing us to understand how RINC scales to large data centers. Guided by observations made in recent studies on data center characteristic, our simulator makes the following assumptions:

- Cloud/Data Center Size: We assume the cloud is run atop a physical data center that has 100K servers [17].
- Connections per-VM: We assume each VM has roughly 10K concurrent connections [35].
- Percent of Troubled Connections: Related works [17] have studied performance problems in data centers and observed that 2% and 5% of the connections are troubled.

6.2 Accuracy of RINC

In this section, we validate our inference algorithms against Web10G by comparing the statistics generated for connections that have been monitored from start to finish. Then, we use these as ground truth to evaluate the accuracy of our midstream algorithms by examining how well RINC's initializer converges with the correct values, and how quickly this convergence happens.

6.2.1 Verification Against Full Stream Connections

We used *testbed1* to verify the correctness of RINC's measurements for connections that are monitored from the beginning by comparing the calculated statistics with alternative approaches such as Web10G. We observe identical statistics for constants, counters, and sampling statistics. However, we observed a small error for certain aspects of the complex internal variables. The worst we observed was for CWND values (the estimated values was within the 3% of the correct values).

6.2.2 Accuracy and Efficiency of Midstream Algorithms

To evaluate the functionality of RINC's midstream algorithms, we use real data-center packet traces [12].

For these experiments, we preprocess this trace and eliminate connections with less than four packets and flows where we don't observe the TCP handshake. We only focus on connections with the TCP handshake because this allows us to validate our algorithms by comparing against the ground truth as discussed earlier. After preprocessing we are left with 129,480 connections. We convert each connection into a midstream connection by skipping a random number of packets.

In analyzing our algorithms, we aim to answer the following questions: *how close* does RINC's output get to the

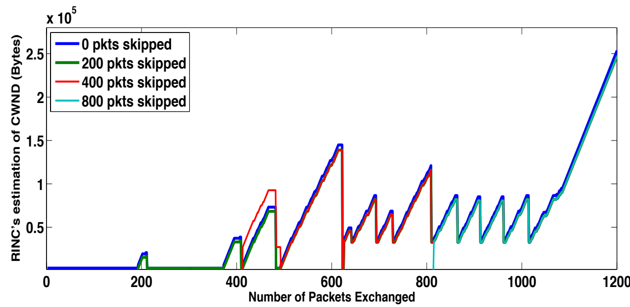


Figure 6: Accuracy of CWND estimation in midstream fashion for different starting points

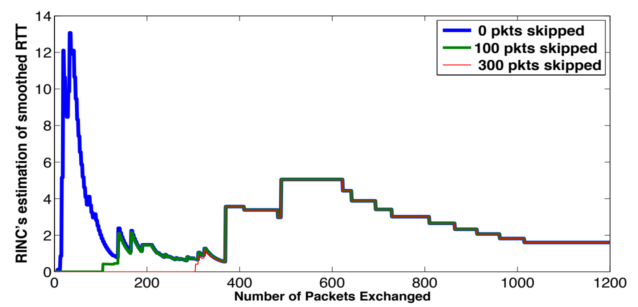


Figure 7: Accuracy of Smoothed RTT estimation in midstream fashion for different starting points

ground truth? (Accuracy), and *How long* does it take for RINC to converge with to the ground truth? (Speed)

Accuracy: To understand the accuracy of RINC's midstream algorithms, we examined different statistics collected for the midstream flows we extracted from the data center trace. In this section, we focus on CWND as we showed, in Section 6.2.1, that CWND has the largest error rate. We also show the accuracy results for RTT since it is an important metric in performance diagnosis in networks.

In Figure 6, we present CWND estimated values for midstream flows created by skipping 200 (green), 400 (red), and 800 (light blue) packets into a flow. The figure also includes the ground-truth, 0 (blue line), the actual CWND as calculated by monitoring the connection from its inception. From Figure 6, we observe three main points: First, (from 200 pkts), once our algorithm converges it never diverges, this occurs because when the initial values are close and both experience the same transitional events on the same state machines, then the result of the two will be similar; Second, (from 400 pkts), while our initial estimations may be largely inaccurate, observing events like a loss allows the *H1* heuristic to correctly infer the real value; and Third, (from 800 pkts), even though initial errors in estimating CWND are large, by observing the sending patterns and using *H2*, RINC's estimation quickly converges to the actual value.

Figure 7 shows RINC's estimation of smoothen RTT values for a midstream connection in two different situations: when the initial 100 packets of a flow are assumed to be lost, and when the initial 300 packets of the connection are assumed to be lost. It can be shown that in both cases even when the initial variance is high, the two values will eventually converge as long as they follow the same algorithm for sampling (here both use [27]).

Speed: Next, we examine the speed with which our algorithms are able to converge. In Figure 8, we present a CDF of the time it takes our algorithms to converge within 10% of the ground truth. Figure 8 shows that for 85% of the midstream connections, RINC converges within 10% of the ground truth in less than 100ms. Upon examination of the

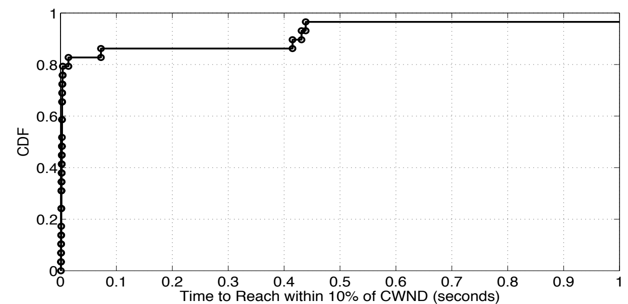


Figure 8: CDF of the time to reach within 10% of the CWND value for midstream connections

connections that take longer than 100ms, we observed that many of these connections do not have any packets in flight for a long duration of time, and are ultimately resolved by the *H1* heuristic.

6.3 Scalability

Next, we evaluate the overhead imposed by the *Local Agent* on the hypervisor, by examining its memory and CPU utilization, and on the network by examining the communications between the *Local Agent* and *Global Coordinator*.

6.3.1 Network Overhead

The network footprint of RINC is a function of the information being reported which varies by application (statistics collected) and by the number of connections being reported on. In this section, we simulate RINC using the simulator and simulated workloads (described in Section 6.1).

In Figure 9, we examine the network utilization for various applications and network sizes. We observe that RINC scales linearly with size and fraction of connections that are reported on. Furthermore, RINC imposes minimal network overhead; RINC utilizes only a small fraction, 0.0006%, of each server's 1Gig uplink.

When querying all statistics for all connections (labelled "Worst case" in Figure 9), we observe that the network over-

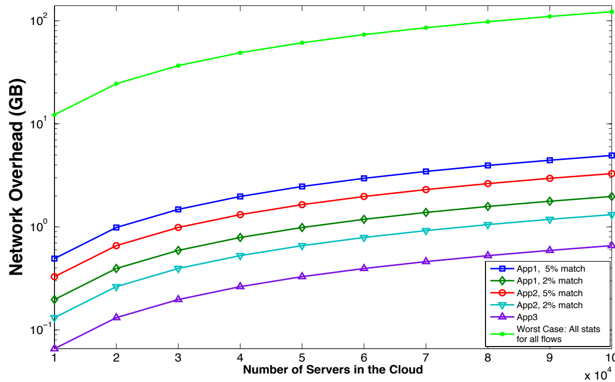


Figure 9: RINC's Network Overhead, simulation parameters: MSS:1460, packet header size:40B, 10K connections per server

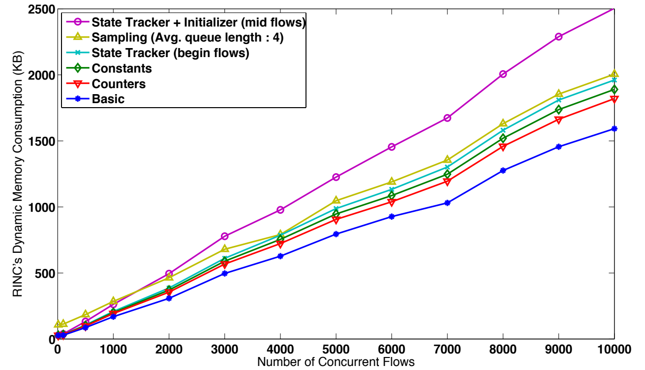


Figure 10: Memory consumption of each measurement module per different number of concurrent flows

head increases by two orders of magnitude. However, RINC still utilizes a small fraction, 0.12%, of each 1Gig server's uplink.

6.3.2 Memory Overhead

RINC's memory overhead consists of two parts: a variable overhead for the sampling module, and a fixed overhead for all other modules. We start by understanding the cause of variability and placing an upper bound on the variability. Given this upper bound, we are then able to calculate the memory footprint of RINC.

Upper-Bounding the Variable Overhead: The sampling module calculates the RTT and RTO for all connections, and this requires the module to maintain multiple packets in a queue for each connection until their corresponding ACKs are received and they can be freed. Thus, the memory overhead of this module depends on the number of un-acknowledged packets a connection has.

To better understand the memory consumption of the sampling module we used iPerf to generate traffic and Linux TC [3] to vary the latency, bandwidth, and loss rates. We observed that 95% of the time all connections in our testbed have less than four sample packets in their queue. Thus to evaluate the memory consumption of this module in our simulator, we assume a sampling queue size of four.[1]

Memory Overhead: Using testbed2, we generate varying number of flows with iPerf (10 to 10K parallel connections), fix the sampling queue length to 4 to emulate the worst-case, and use Valgrind [6] to measure the memory overhead of RINC. The results are shown in Figure10.

From Figure 10 we observe that RINC scales linearly regardless of the statistics being collected. Interestingly we observe three important characteristics of the system. First, our midstream heuristics of inferring internal state, embodied in the initializer, incur a 20% overhead over the normal state-tracking algorithm. This 20% overhead accounts for the additional processing being performed by the initializer when using our heuristics to infer statistics. Second, there is a noticeable memory differences between using *Constants*, *Counters*, *Sampling*, and the *State Tracker* for midstream flows. Thus, we advise that cloud providers develop diagnosis applications in phases, and only turn on the most heavy-weight measurement, *State Tracker*, for the flows that show performance problems. Finally, *Counters* (green) and *Constants* (red) require slightly more memory than the *Basic*; therefore, we advise the cloud providers to use them as the initial phase of diagnosis to find the troubled connection and narrow down the set of connections which need heavier measurements.

6.3.3 CPU Overhead

RINC's CPU overhead is composed of three factors: overhead of capturing packets using the libpcap interface (*pcap*); overhead of having the measurement module process each packet and update the hash table (*connection table update*); and overhead of having the communication module query the hash table and return results of executing the query to the *Global Coordinator* (*query execution*) for as many times as requested.

To evaluate these, we analyze *Local Agent* in our testbed using iPerf to generate load. We vary the number of concurrent flows and their average rates to understand the scaling properties. Due to the physical restrictions of the hosts' NICs, the total bandwidth of all connections on each local agent cannot exceed 1Gbps.

To decompose the overheads of the different components we run three independent tests: first, only involving pcap for (*pcap*), second, involving only the measurement module

[1] It is worth mentioning that while several factors affect the average sampling queue length in a data center (delayed ACKs of the receiver, back-logged versus non back-logged sender, available bandwidth, and queuing delays), this number is still bounded by the flight size, which is bounded by the Delay Bandwidth Product.

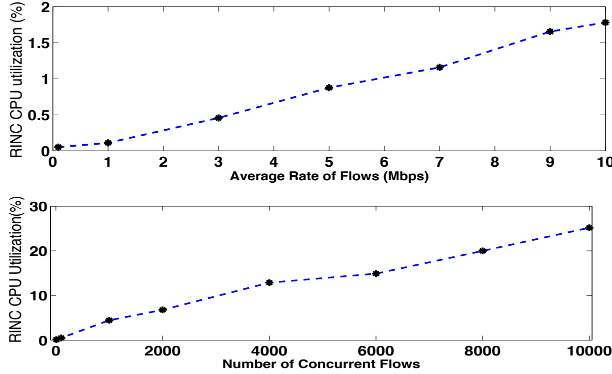


Figure 12: top: CPU rates per average flow rates (100 flows), bottom: CPU rates per number of flows(avg rate: 100kbps)

(*flow table update*), and third, involving both the measurement module and the communication module (*query execution*).

In Figure 11 we can compare these three components and observe that RINC's CPU utilization grows with both the number and the average rate of connections that need to be monitored.

To better understand the dominant factor, we examine each factor independently: number of flows (in Figure 12.a) and connection rates (in Figure 12.b).

We observe that, unlike memory overheads that are strictly a function of number of connections monitored, the CPU overhead is a function of both the number of connections and the sending rate of each connection. This interesting observation highlights a key difference between RINC and related approaches like VND [33], whose memory and CPU overheads are both a function of number of connections and rate. RINC has a better memory overhead because it does not store raw packet headers, rather RINC stores meaningful summarized statistics calculated in real time. We elaborate more in Section 6.5.

6.4 Performance

Next, we evaluate the performance of RINC by examining the *Query Execution Time*, or the time it takes RINC to gather the statistics requested by the diagnosis applications described in Section 4.2. This time is dominated by three factors: the time for the *communication module* to gather statistics, the network latency between the *communication module* module and the *Global Coordinator*, and the time for the *Global Coordinator* to concatenate the results.

Given that the time to concatenate the results is negligible and the network latency is highly dependent on the network characteristics of the cloud, we focus on quantifying and understanding the scaling properties of the *communication module* module. To do this analysis, we run RINC in testbed1 varying the number of iPerf connections generated by each server from 1K to 10K.

As discussed in Section 5, the *communication module* module performs a lookup into the connection table to find the connections that match the query criterion, generates and sends a report of the statistics collected to the *Global Coordinator*. An implication of this, is that performance of the *communication module* grows linearly with the number of connections stored in the flow table, the number of connections that match the query (matching rate), and the number of statistics copied to generate the response to the query. Figure 13, confirms that *communication module* performs linearly as a function of matching rate, number of flows, and type of statistics gathered. This linear relationship is especially evident when 'app1 (10%match)' is compared with 'app2 (10%match)'. Further we observe that in general the *communication module* module response quickly, responding in less than 2.5 msecs when less than 10% of the connections match the application's query.

6.5 Comparison against VND

Finally, we compare RINC against VND [33] and focus on the overhead incurred in the hypervisor and in the network. Recall, VND enables cloud tenants to diagnose problems in their virtual networks by duplicating packets, processing them, and sending this raw data to a central entity and allowing the tenants to query this data.

6.5.1 Overhead

Memory: We observe that while VND has a fixed constant memory overhead, namely '1Gbps network traffic dump costs an extra 59 MB/s of memory' [33], VND requires significantly more memory because it stores raw packet headers at each hop, whereas RINC stores calculated per-flow statistics, allowing RINC to require 3.5MB in the worst case.

Network: RINC outperforms VND in network efficiency by an order of magnitude: RINC uses less than 0.01% of the server uplink while VND uses 0.1%.

CPU: While both RINC and VND capture and process packets headers to generate statistics, VND uses delayed processing; thus, we expect it to have a smaller CPU overhead but at the cost of larger memory and networking overheads.

In summary, our evaluations show that RINC:

- incurs an order of magnitude less overhead than VND [33].
- imposes minimal overhead and scales to large clouds with over 100K servers.

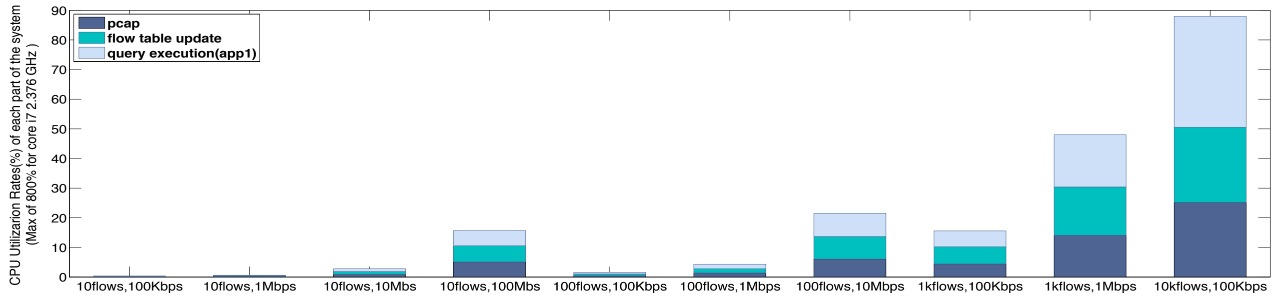


Figure 11: A comparison between CPU utilization rates per different number of concurrent flows and average rates, burst ratio: 1.1, maximum BW: 1.1Gbps, app: long-lived flows detection, running time: 1 minute, query frequency: 500ms

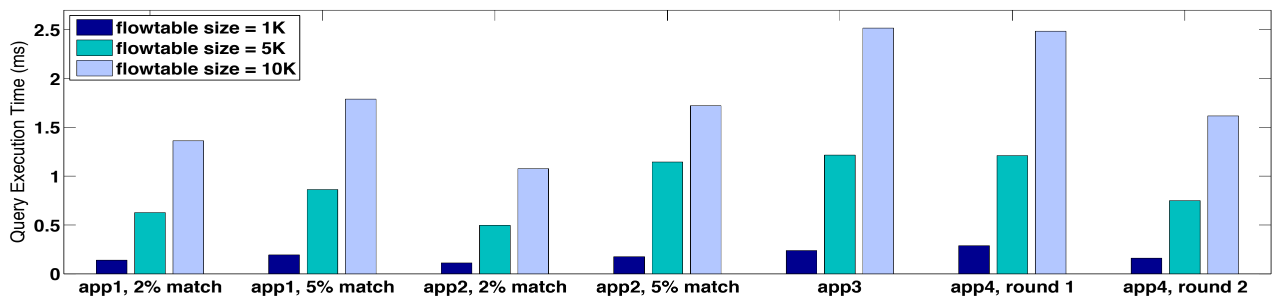


Figure 13: Query Execution Times

- accurately infers the internal state of over 95% of a data center's [12] midstream connections in less than 10 seconds.

7 Related Works

The mostly related work, VND [33] enables cloud tenants to diagnose problems in their virtual networks. Rather than enabling the tenant, RINC tackles a complementary problem of enabling the provider to diagnose problems across the entire cloud stack, including the provider infrastructure and the tenant's VM. Furthermore, by selectively and efficiently collecting statistics, RINC has a much lower overhead than VND.

RINC represents an evolution of a long line of rich work on end-host based techniques [29, 30, 7] as well as network-based [37, 11, 25, 19, 20] for diagnosing TCP connections. Unfortunately, traditional end-host based techniques [29, 30, 7] need to run within the VM and are thus considered too invasive because they require the cloud provider to interfere with the tenant's VM. To overcome this limitation, RINC runs within the hypervisor and infers the internal state of the tenant's VM. Moreover, by inferring the internal state of tenant's connections, RINC is able to diagnose a richer set of problems than network-based systems [37, 11, 25, 19, 20]. Similarly, recent efforts to employ programmability of SDN networks [23, 22, 18] focus on detecting low-level network problems, such as routing problems, thus limiting the set of problems that can be diagnosed. Furthermore, these approaches provide little insight on how individual tenants are impacted.

Orthogonal to detecting and diagnosing problems, are the research efforts on scaling applications within the cloud [26, 28]. Unlike RINC which focuses on understanding anomalous performance, research on scaling applications focuses on predicting and understanding the normal or expected behavior of an application.

Existing approaches to diagnose problems across the network and end-host layers are designed for environments with a single administrative domain, such as, enterprise [21, 10, 32], home-networks [8] and ISP networks, or require cooperation [16, 24] or collaboration [15] between the different administrative domains. Unlike these approaches, RINC enables the cloud provider to diagnose and pinpoint problems across domains without interacting, interfering, or impacting the tenant.

8 Discussion

Adapting To Variants of TCP. The current implementation of RINC assumes that tenants use TCP Reno. However the general principles behind our inference techniques can be applied to other variants of TCP; disambiguating state and forcing TCP into known-states. Most TCP variants are

based of off the TCP-Tahoe's state machine and differ in what actions trigger state transitions, in the behavior in certain states, and in the number of states in the state-machine. Regardless of the differences, similar domain knowledge about the state machine can be used to develop algorithms that allow inference of state.

Enabling Problem Resolution. RINC provides the cloud operators with sufficient information to determine the cause and location of the problem. This information can be used to resolve the problem. For example, upon detecting hypervisor level congestion, an operator may migrate the affected VMs. To automate problem resolution, we envision that applications written with RINC can be integrated with the cloud orchestration frameworks, e.g. OpenStack, allowing it to detect and react to problems in real-time.

9 Conclusion

In this paper, we present a novel real-time diagnosis cloud framework called RINC. RINC overcomes drawbacks inherent in existing approaches by operating in the hypervisor and inferring per-connection statistics, thus eliminating the need to modify tenant VMs. Also by selective collection of statistics it ensures scalability with respect to the number of connections in the cloud. We propose a cloud-wide diagnosis interface that allows the operators specify, on-demand and implicitly, the select statistics to monitor along three dimensions. We identify a set of challenges involved in performing on-demand monitoring of midstream connections and present a set of novel statistics that allow RINC to infer the state on midstream connections. Our evaluations show that RINC imposes minimal overheads, thus allowing it to scale to large clouds, and that RINC is sufficiently expressive to support the development of a wide range of diagnosis application

References

[1] iptables. `http://www.netfilter.org`.

[2] libpcap. `http://www.tcpdump.org`.

[3] Linux traffic control. `http://lartc.org/manpages/tc.txt`.

[4] netfilter library. `http://www.netfilter.org`.

[5] Nmap. `http://www.nmap.org`.

[6] Valgrind. `http://valgrind.org/`.

[7] The web10g project. `http://www.web10g.org`.

[8] B. Aggarwal, R. Bhagwan, T. Das, S. Eswaran, V. N. Padmanabhan, and G. M. Voelker. Netprints: diagnosing home network misconfigurations using shared knowledge. In *NSDI*, 2009.

[9] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, pages 63–74, New York, NY, USA, 2010. ACM.

[10] P. Bahl, R. Chandra, A. G. Greenberg, S. Kandula, D. A. Maltz, and M. Zhang. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *SIGCOMM '07*.

[11] P. Barford and M. Crovella. Critical path analysis of TCP transactions. In *ACM SIGCOMM*, 2000.

[12] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, IMC '10, pages 267–280, New York, NY, USA, 2010. ACM.

[13] T. Benson, S. Sahu, A. Akella, and A. Shaikh. A first look at problems in the cloud. HotCloud '10.

[14] T. Benson, S. Sahu, A. Akella, and A. Shaikh. A first look at problems in the cloud. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 15–15, Berkeley, CA, USA, 2010. USENIX Association.

[15] F. Dinu and T. S. E. Ng. Synergy2cloud: Introducing cross-sharing of application experiences into the cloud management cycle. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*, Hot-ICE'12, pages 6–6, Berkeley, CA, USA, 2012. USENIX Association.

[16] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-trace: A pervasive network tracing framework. In *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation*, NSDI'07, pages 20–20, Berkeley, CA, USA, 2007. USENIX Association.

[17] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. Vl2: A scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, SIGCOMM '09, pages 51–62, New York, NY, USA, 2009. ACM.

[18] N. Handigol, B. Heller, V. Jeyakumar, D. Maziéres, and N. McKeown. Where is the debugger for my software-defined network? In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, pages 55–60, New York, NY, USA, 2012. ACM.

[19] S. Jaiswal, G. Iannaccone, C. Diot, J. Kurose, and D. Towsley. Measurement and classification of out-of-sequence packets in a tier-1 ip backbone. *IEEE/ACM Trans. Netw.*, 15(1):54–66, Feb. 2007.

[20] S. Jaiswal, G. Iannaccone, C. Diot, and D. F. Towsley. Inferring tcp connection characteristics through passive measurements. In *INFOCOM*, 2004.

[21] S. Kandula, R. Mahajan, P. Verkaik, S. Agarwal, J. Padhye, and P. Bahl. Detailed diagnosis in enterprise networks. SIGCOMM '09.

[22] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 9–9, Berkeley, CA, USA, 2012. USENIX Association.

[23] A. Khurshid, W. Zhou, M. Caesar, and P. B. Godfrey. Veriflow: Verifying network-wide invariants in real time. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, pages 49–54, New York, NY, USA, 2012. ACM.

[24] B. Krishnamurthy, H. V. Madhyastha, and O. Spatscheck. Atmen: A triggered network measurement infrastructure. In *Proceedings of the 14th International Conference on World Wide Web*, WWW '05, pages 499–509, New York, NY, USA, 2005. ACM.

[25] A. Lakhina, M. Crovella, and C. Diot. Diagnosing network-wide traffic anomalies. In *Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '04, pages 219–230, New York, NY, USA, 2004. ACM.

[26] H. Nguyen, Z. Shen, X. Gu, S. Subbiah, and J. Wilkes. Agile: Elastic distributed resource scaling for infrastructure-as-a-service. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*, pages 69–82, San Jose, CA, 2013. USENIX.

[27] V. Paxson and M. Allman. Computing TCP's Retransmission Timer. RFC 2988 (Proposed Standard), November.

[28] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes. Cloudscale: Elastic resource scaling for multi-tenant cloud systems. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, SOCC '11, pages 5:1–5:14, New York, NY, USA, 2011. ACM.

[29] P. Sun, M. Yu, M. J. Freedman, and J. Rexford. Identifying performance bottlenecks in cdns through tcp-level monitoring. In *Proceedings of the First ACM SIGCOMM Workshop on Measurements Up the Stack*, W-MUST '11, pages 49–54, New York, NY, USA, 2011. ACM.

[30] P. Sun, M. Yu, M. J. Freedman, J. Rexford, and D. Walker. Hone: Joint host-network traffic management in software-defined networks. *JNSM Special Issue on Management of Software Defined Networks (SDNs)*, 1, Dec 2014.

[31] V. Varadarajan, T. Kooburat, B. Farley, T. Ristenpart, and M. M. Swift. Resource-freeing attacks: Improve your cloud performance (at your neighbor's expense). In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 281–292, New York, NY, USA, 2012. ACM.

[32] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y.-M. Wang. Automatic misconfiguration troubleshooting with peerpressure. In *SOSP*, 2004.

[33] W. Wu, G. Wang, A. Akella, and A. Shaikh. Virtual network diagnosis as a service. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 9:1–9:15, New York, NY, USA, 2013. ACM.

[34] Y. Xu, Z. Musgrave, B. Noble, and M. Bailey. Bobtail: Avoiding long tails in the cloud. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 329–342, Berkeley, CA, USA, 2013. USENIX Association.

[35] M. Yu, A. Greenberg, D. Maltz, J. Rexford, L. Yuan, S. Kandula, and C. Kim. Profiling network performance for multi-tier data center applications. In *NSDI*, 2011.

[36] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown. Automatic test packet generation. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '12, pages 241–252, New York, NY, USA, 2012. ACM.

[37] Y. Zhang, L. Breslau, V. Paxson, and S. Shenker. On the characteristics and origins of Internet flow rates. In *ACM SIGCOMM*, 2002.