Profiling Network Performance for Multi-Tier Data Center Applications

Minlan Yu[*] *Albert Greenberg*[†] *Dave Maltz*[†] *Jennifer Rexford*[*] *Lihua Yuan*[†]
Srikanth Kandula[†] *Changhoon Kim*[†]
[*] *Princeton University* [†] *Microsoft*

Abstract

Network performance problems are notoriously tricky to diagnose, and this is magnified when applications are often split into multiple tiers of application components spread across thousands of servers in a data center. Problems often arise in the communication between the tiers, where either the application or the network (or both!) could be to blame. In this paper, we present SNAP, a scalable network-application profiler that guides developers in identifying and fixing performance problems. SNAP passively collects TCP statistics and socket-call logs with low computation and storage overhead, and correlates across shared resources (e.g., host, link, switch) and connections to pinpoint the location of the problem (e.g., send buffer mismanagement, TCP/application conflicts, application-generated microbursts, or network congestion). Our one-week deployment of SNAP in a production data center (with over 8,000 servers and over 700 application components) has already helped developers uncover 15 major performance problems in application software, the network stack on the server, and the underlying network.

1 Introduction

Modern data-center applications, running over networks with unusually high bandwidth and low latency, should have great communication performance. Yet, these applications often experience low throughput and high delay *between* the front-end user-facing servers and the back-end servers that perform database, storage, and indexing operations. Troubleshooting network performance problems is hard. Existing solutions—like detailed application-level logs or fine-grain packet monitoring—are too expensive to run continuously, and still offer too little insight into where performance problems lie. Instead, we argue that data centers should perform continuous, lightweight profiling of the end-host network stack, coupled with algorithms for classifying and correlating performance problems.

1.1 Troubleshooting Network Performance

The nature of the data-center environment makes detecting and locating performance problems particularly challenging. Applications typically consist of tens to hundreds of application components, arranged in multiple tiers of front-ends and back-ends, and spread across hundreds to tens of thousands of servers. Application developers are continually updating their code to add features or fix bugs, so application components evolve independently and are updated while the application remains in operation. Human factors also enter into play: most developers do not have a deep understanding of how their design decisions interact with TCP or the network. There is a constant influx of new developers for whom the intricacies of Nagle's algorithm, delayed acknowledgments, and silly window syndrome remains a mystery.[1]

As a result, new network performance problems happen all the time. Compared to equipment failures that are relatively easy to detect, performance problems are tricky because they happen sporadically and many different components could be responsible. The developers sometimes blame "the network" for problems they cannot diagnose; in turn, the network operators blame the developers if the network shows no signs of equipment failures or persistent congestion. Often, they are both right, and the network stack or some subtle interaction between components is actually responsible [2, 3]. For example, an application sending small messages can trigger Nagle's algorithm in the TCP stack, causing transmission delays leading to terrible application throughput.

In the production data center we study, the process of actually detecting and locating even a single network per-

[1] Some applications (like memcached [1]) use UDP, and re-implement reliability, error detection, and flow control; however, these mechanisms can also introduce performance problems.

formance problem typically requires tens to hundreds of hours of the developers' time. They collect detailed application logs (too heavy-weight to run continuously), deploy dedicated packet sniffers (too expensive to run ubiquitously), or sample the data (too coarse-grained to catch performance problems). They then pore over these logs and traces using a combination of manual inspection and custom-crafted analysis tools to attempt to track down the issue. Often the investigation fails or runs out of time, and some performance problems persist for months before they are finally caught and corrected.

1.2 Lightweight, Continuous Profiling

In this paper, we argue that the data centers should *continuously* profile network performance, and analyze the data in real time to help pinpoint the source of the problems. Given the complexity of data-center applications, we cannot hope to fully automate the detection, diagnosis, and repair of network performance problems. Instead, our goal is dramatically reducing the demand for developer time by automatically identifying performance problems and narrowing them down to specific times and places (e.g., send buffer, delayed ACK, or network congestion). Any viable solution must be

- **Lightweight:** Running everywhere, all the time, requires a solution that is very lightweight (in terms of CPU, storage, and network overhead), so as not to degrade application performance.
- **Generic:** Given the constantly changing nature of the applications, our solution must detect problems without depending on detailed knowledge of the application or its log formats.
- **Precise:** To provide meaningful insights, the solution must pinpoint the component causing network performance problems, and tease apart interactions between the application and the network.

Finally, the system should help two very different kinds of users: (i) a *developer* who needs to detect, diagnose, and fix performance problems in his particular application and (ii) a *data-center operator* who needs to understand performance problems with the underlying platform so that he can tune the network stack, change server placement, or upgrade network equipment. In this paper, we present *SNAP (Scalable Network-Application Profiler)*, a tool that enables application developers and data-center operators to detect and diagnose these performance problems. SNAP represents an "existence proof" that a tool meeting our three requirements can be built, deployed in a production data center, and provide valuable information to both kinds of users.

SNAP capitalizes on the unique properties of modern data centers:

- SNAP has *full knowledge* of the network topology, the network-stack configuration, and the mapping of applications to servers. This allows SNAP to use correlation to identify applications with frequent problems, as well as congested resources (e.g., hosts or links) that affect multiple applications.
- SNAP can instrument the *network stack* to observe the evolution of TCP connections directly, rather than trying to infer TCP behavior from packet traces. In addition, SNAP can collect finer-grain information, compared to conventional SNMP statistics, without resorting to packet monitoring.

In addition, once the developers fix a problem (or the operator tunes the underlying platform), we can verify that the change truly did improve network performance.

1.3 SNAP Research Contributions

SNAP passively collects TCP statistics and socket-level logs in real time, classifies and correlates the data to pinpoint performance problems. The profiler quickly identifies the right location (end host, link, or switch), the right layer (application, network stack, or network), at the right time. Our major contributions of the paper are:

Efficient, systematic profiling of network-application interactions: SNAP provides a simple, efficient way to detect performance problems through real-time analysis of passively-collected measurements of the network stack. We provide a systematic way to identify the component (e.g., sender application, send buffer, network, or receiver) responsible for the performance problem. SNAP also correlates across connections that belong to the same application, or share underlying resources, to provide more insight into the sources of problems.

Performance characterization of a production data center: We deployed SNAP in a data center with over 8,000 servers, and over 700 application components (including map-reduce, storage, database, and search services). We characterize the sources of performance problems, which helps data-center operators improve the underlying platform and better tune the network.

Case studies of performance bugs detected by SNAP: SNAP pinpointed 15 significant and unexpected problems in application software, the network stack, and the interaction between the two. SNAP saved the developers significant effort in locating and fixing these problems, leading to large performance improvements.

Section 2 presents the design and development of SNAP. Section 3 describes our data-center environment

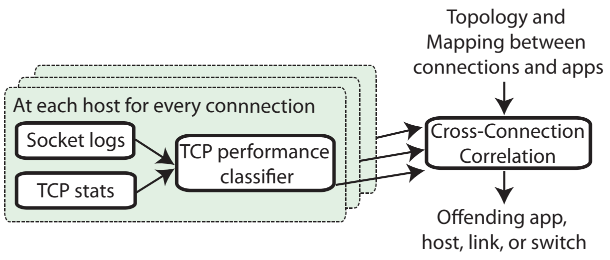
Topology and
Mapping between
connections and apps
At each host for every connnection
Socket logs
TCP stats
TCP performance classifier
Cross-Connection Correlation
Offending app, host, link, or switch

Figure 1: SNAP socket-level monitoring and analysis

and how SNAP was deployed. Section 4 validates SNAP against both labeled data (i.e., known performance problems) and detailed packet traces. Then, we present an evaluation of our one-week deployment of SNAP from the viewpoint of both the data-center operator (Section 5) and the application developer (Section 6). Section 7 shows how to reduce the overhead of SNAP through dynamic tuning of the polling rate. Section 8 discusses related work and Section 9 concludes the paper.

2 Design of the SNAP Profiler

In this section, we describe how SNAP pinpoints performance problems. Figure 1 shows the main components of our system. First, we collect *TCP-connection statistics*, augmented by *socket-level logs* of application read and write operations, in real time with low overhead. Second, we run a *TCP classifier* that identifies and categorizes periods of bad performance for each socket, and logs the diagnosis and a time sequence of the collected data. Finally, based on the logs, we have a *centralized correlator* that correlates across connections that share a common resource or belong to the same application to pinpoint the performance problems.

2.1 Socket-Level Monitoring of TCP

Data centers host a wide variety of applications that may use different communication methods and design patterns, so our techniques must be quite general in order to work across the space. The following three goals guided the design of our system, and led us *away* from using the SNMP statistics, packet traces, or application logs.

(i) Fine-grained profiling: The data should be fine-grained enough to indicate performance problems for individual applications on a small timescale (e.g, tens of milliseconds or seconds). Switches typically only capture link loads at a one-minute timescale, which is far too coarse-grained to detect many performance problems. For example, the TCP incast problem [3], caused by micro bursts of traffic at the timescale of tens of milliseconds, is not even visible in SNMP data.

Statistic	Definition
CurAppWQueue	# of bytes in the send buffer
MaxAppWQueue	Max # of bytes in send buffer over the entire socket lifetime
#FastRetrans	Total # of fast retransmissions
#Timeout	Total # of timeouts
#SampleRTT	Total # of RTT samples
#SumRTT	Sum of RTTs that TCP samples
RwinLimitTime	Cumulated time an application is receiver window limited
CwinLimitTime	Cumulated time an application is congestion window limited
SentBytes	Cumulated # of bytes the socket has sent over the entire lifetime
Cwin	Current congestion window
Rwin	Announced receiver window

Table 1: Key TCP-level statistics for each socket [5]

(ii) Low overhead: Data centers can be huge, with hundreds of thousands of hosts and tens of thousands sockets on each host. Yet, the data collection must not degrade application performance. Packet traces are too expensive to capture in real time, to process at line speed, or to store on disk. In addition, capturing packet traces on high-speed links (e.g., 1-10 Gbps in data centers) often leads to measurement errors including drops, additions, and resequencing of packets [4]. Thus it is impossible to capture packet trace everywhere, all the time to catch new performance problems.

(iii) Generic across applications: Individual applications often generate detailed logs, but these logs differ from one application to another. Instead, we focus on measurements that do not require application support so our tool can work across a variety of applications.

Through our work on SNAP, we found that the following two kinds of per-socket information can be collected cheaply enough to be used in analysis of large-scale data center applications, while still providing enough insight to diagnose where the performance problem lie (whether they are from the application software, from network issues, or from the interaction between the two).

TCP-level statistics: RFC 4898 [5] defines a mechanism for exposing the internal counters and variables of a TCP state-machine that is implemented in both Linux [6] and Windows [7]. We select and collect the statistics shown in Table 1 based on our diagnosis experience[2], which together expose the data-transfer performance of a socket. There are two types of statistics: (1) instantaneous snapshots (e.g., *Cwin*) that show the current value

[2]There are a few other variables in the TCP stack such as the time TCP spends in SlowStart stage, which are also useful but we did not mention in the paper due to space limit.

Locations	Problems	App/Net	Detection method
Sender app	Sender app limited	App	Not any other problems
Send buffer	Send buffer limited	App and Net	*CurAppWQueue* ≈ *MaxAppWQueue*
Network	Fast retransmission	Net	*diff(#FastRetrans) > 0*
	Timeout	Net	*diff(#Timeout) > 0)*
Receiver	Delayed ACK	App and Net	*diff(SumRTT) > diff(SampleRTT)*MaxQueuingDelay*
	Receiver window limited	App and Net	*diff(#RwinLimitTime) > 0*

Table 2: Classes of network performance for a socket

of a variable in the TCP stack; and (2) cumulative counters (e.g., *#FastRetrans*) that count the number of events (e.g., the number of fast retransmissions) that happened over the lifetime of the socket. *#SampleRTT* and *SumRTT* are the cumulative values of the number of packets TCP sampled and the sum of the RTTs for these sampled packets. To calculate the retransmission timeout (RTO), TCP randomly samples one packet in each congestion window, and measures the time from the transmission of a packet to the time TCP receives the ACK for the packet as the RTT for this packet.

These statistics are updated by the TCP stack as individual packets are sent and received, making it too expensive to log every change of these values. Instead, we periodically poll these statistics. For the cumulative counters, we calculate the difference between two polls (e.g., *diff(#FastRetrans)*). For snapshot values, we sample with a Poisson interval. According to the PASTA property (Poisson Arrivals See Time Averages), the samples are a representative view of the state of the system.

Socket-call logs: Event-tracing systems in Windows [8] and Linux [9] record the time and number of bytes (*ReadBytes* and *WriteBytes*) whenever the socket makes a read/write call. Socket-call logs show the applications' data-transfer behavior, such as how many connections they initiated, how long they maintain each connection, and how much data they read/write (as opposed to the data that TCP actually transfers, i.e., *SentBytes*). These logs supplement the TCP statistics with application behavior to help developers diagnose problems. The socket-level logs are collected in an event-driven fashion, providing fine-grained information with low overhead. In comparison, the TCP statistics introduce a trade-off between accuracy and the polling overhead. For example, if SNAP polls TCP statistics once per second, a short burst of packet losses is hard to distinguish from a modest loss rate throughout the interval.

In summary, SNAP collects two types of data in the following formats: (i) timestamp, 4-tuples (source and destination address/port), *ReadBytes*, and *WriteBytes*; and (ii) timestamp, 4-tuples, TCP-level logs (Table 1). SNAP uses TCP-level logs to classify the performance problems and pinpoint the location of the problem, and then provides both the relevant TCP-level and socket-level logs for the affected connections for that period of time. Developers can use these logs to quickly find the root cause of performance problems.

2.2 Classifying Single-Socket Performance

Although it is difficult to determine the root cause of performance problems, we can pinpoint the component that is limiting performance. We classify performance problems in terms of the stages of data delivery, as summarized in the two columns of Table 2[3]:

1. Application generates the data: The sender application may not generate the data fast enough, either by design or because of bottlenecks elsewhere (e.g., CPU, memory, or disk). For example, the sender may write a small amount of data, triggering Nagle's algorithm [10] which combines small writes together into larger packets for better network utilization, at the expense of delay.

2. Data are copied from the application buffer to the send buffer: Even when the network is not congested, a small send buffer can limit throughput by stalling application writes. The send buffer must keep data until acknowledgments arrive from the receiver, limiting the buffer space available for writing new data.

3. TCP sends the data to the network: A congested network may drop packets, leading to lower throughput or higher delay. The sender can detect packet loss by receiving three duplicate ACKs, leading to a fast retransmission. When packet losses do not trigger a triple duplicate ACK, the sender must wait for a retransmission timeout (RTO) to detect loss and retransmit the data.

4. Receiver receives the data and sends an acknowledgment: The receiver may not read data, or acknowledge their arrival, quickly enough. The receiver window can limit the throughput if the receiver is not reading the data quickly enough (e.g., caused by a CPU starvation), allowing data to fill the receive buffer. A receiver delays sending acknowledgments in the hope of piggybacking the ACK on data in the reverse direction. The receiver acknowledges every other packet and waits up to 200 ms before sending an ACK.

[3]The table only summarizes major performance problems and can be extended to cover other problems such as out-of-order packets.

The TCP statistics provide direct visibility into certain performance problems like packet loss and receiver-window limits, where cumulative counts (e.g., *#Timeout*, *#FastRetrans*, and *RwinLimitTime*) indicate whether the problem occurred at any time during the polling interval. Detecting other problems relies on an instantaneous snapshot, such as comparing the current backlog of the send buffer (*CurAppWQueue*) to its maximum size (*MaxAppWQueue*); polling with a Poisson distribution allows SNAP to accurately estimate the fraction of time a connection is send-buffer limited. Pinpointing other latency problems requires some notion of expected delays. For example, the RTT should not be larger than the propagation delay plus the maximum queuing delay (*MaxQueuingDelay*) (whose value is measured in advance by operators), unless a problem like delayed ACK occurs. SNAP incorporates knowledge of the network configuration to identify these parameters.

SNAP detects send-buffer, network, and receiver problems using the rules listed in the last column of Table 2, where multiple problems may take place for the same socket during the same time interval. If any of these problems are detected, SNAP logs the diagnosis and all the variables in Table 1—as well as *WriteBytes* from the socket-call data—to provide the developers with detailed information to track down the problem. In the absence of any of the previous problems, we classify the connection as sender-application limited during the time interval, and log only the socket-call data to track application behavior. Being sender-application limited should be the most common scenario for a connection.

2.3 Correlation Across TCP Connections

Although SNAP can detect performance problems on individual connections in isolation, combining information across multiple connections helps pinpoint the location of the problem. As such, a central controller analyzes the results of the TCP performance classifier, as shown earlier in Figure 1. The central controller can associate each connection with a particular application and with shared resources like a host, links, and switches.

Pinpointing resource constraints (by correlating connections that share a host, link, or switch): Topology and routing data allow SNAP to identify which connections share resources such as a host, link, top-of-rack switch, or aggregator switch. SNAP checks if a performance problem (as identified by the algorithm in Table 2) occurs on many connections traversing the same resource at the same time. For example, packet losses (i.e., *diff(#FastRetrans)* > 0 or *diff(#Timeout)* > 0) on multiple connections traversing the same link would indicate a congested link. This would detect congestion occurring on a much smaller timescale than SNMP could measure. As another example, send-buffer problems for many connections on the same host could indicate that the machine has insufficient memory or a low default configuration of the send-buffer size.

Pinpoint application problem (by correlating across connections in the same application): SNAP also receives a mapping of each socket (as identified by the four-tuple) to an application. SNAP checks if a performance problem occurs on many connections from the same application, across different machines and different times. If so, the application software may not interact well with the underlying TCP layer. With SNAP, we have found several application programs that have severe performance problems and are currently working with developers to address them, as discussed in Section 6.

The two kinds of correlation analysis are similar, except for (i) sets of connections to compare S (i.e., connections sharing a resource vs. belonging to the same service) and (ii) the timescale for the comparison — correlation interval T (i.e., transient resource constraining events taking a few minutes or hours vs. permanent service code problems that lasts for days).

We use a simple linear correlation heuristic that works well in our setting Given a set of connections S and a correlation interval T, the SNAP correlation algorithm outputs whether these connections have correlated performance problems, and provides a time sequence of SNAP logs for operators and developers to diagnose.

We construct a performance vector $\overrightarrow{P_T(c,t)} = (time_k(p_1,c), ..., time_k(p_5,c))_{k=1..\lceil T/t \rceil}$, where t is an aggregation time interval in T and $time_k(p_i)(i = 1..5)$ denotes the total time that connection c is having problem p_i during time period $[(k-1)t, kt]$.[4] We pick c_1 and c_2 in S, calculate the Pearson correlation coefficient, and check if the average across all pairs of connections (*Average Correlation Coefficient ACC*) is larger than a threshold α:

$$ACC = \underset{c_1,c_2 \in S, c_1 \neq c_2}{avg} (cor(\overrightarrow{P_T(c_1,t)}, \overrightarrow{P_T(c_2,t)}) > \alpha,$$

where

$$cor(\overrightarrow{x}, \overrightarrow{y}) = \frac{\sum_i (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_i (x_i - \bar{x})^2 (y_i - \bar{y})^2}}.$$

If the correlation coefficient is high, SNAP reports that the connections in S have a common problem. To extend this correlation for different classes of problems (e.g., one connection's delayed ACK problem triggers

[4] $p_i(i = 1..5)$ are the problems of send buffer limited, fast retransmission, timeout, delayed ACK and receiver window limited respectively. We do not include sender application limited because its time could be determined given the times of the first five problems.

Characteristic	Value
#Hosts	8K
#Applications	700
Operating systems	Win 2003,2008R2
Default send buffer	8 KB
Maximum segment size (MSS)	1460 Bytes
Minimum retrans. timeout	300 ms
Delayed ACK timeout	200 ms
Nagle's algorithm	mostly off
Slow start restart	off
Receiver window autotuning	off

Table 3: Characteristics in the production data center.

the sender application limited problem on another connection), we can extend our solution to use other inference techniques [11, 12] or principal component analysis (PCA) [13].

In practice, we must choose t carefully. With a large value of t, SNAP only compares the coarse-grained performance between connections; for example, if $t = T$, we only check if two connections have the same performance problem with the same percentage of time. With a small t, SNAP can detect fine-grained performance problems (e.g., two connections experiencing packet loss at almost the same time), but are susceptible to clock differences of the two machines and any differences in the polling rates for the two connections. The aggregation interval t should be large enough to mask the differences between the clocks and cannot be smaller than the least common multiple of the polling intervals of the connections.

3 Production Data Center Deployment

We deployed SNAP in a production data center. This section describes the characteristics of the data center and the configuration of SNAP, to set the stage for the following sections.

3.1 Data Center Environment

The data center consists of 8K hosts and runs 700 application components, with the configuration summarized in Table 3. The hosts run either Windows Server 2008 R2 or Windows Server 2003. The default send buffer size is 8K, and the maximum segment size is 1460 Bytes. The minimum retransmission timeout for packet loss is set to 300 ms, and the delayed-acknowledgment timeout is 200 ms. These values in Windows OS are configured for Internet traffic with long RTT.

While the OS enables Nagle's algorithm (which combines small writes into larger packets) by default, most delay-sensitive applications disable Nagle's algorithm using the *NO_DELAY* socket option.

Most applications in the data center use persistent connections to avoid establishing new TCP connections whenever they have data to transmit. Slow-start restart is disabled to reduce the delay arising when applications transfer a large amount of data after an idle period over a persistent connection.

Receiver-window autotuning—a feature in Windows Server 2008 that allows TCP to dynamically tune the receiver window size to maximize throughput—is disabled to avoid bugs in the TCP stack (e.g., [14]). Windows Server 2003 does not support this feature.

3.2 SNAP Configuration

We ran SNAP continuously for a week in August 2010. The polling interval for TCP statistics follows the Poisson distribution with an average inter-arrival time of 500 ms. We collected the socket-call logs for all the connections from and to the servers running SNAP. Over the week, we collected less than 1 GB on each host per day and the total is just terabytes of logs for the week. This is a very small amount of data compared to packet traces which take more than 180 GB per host per day at a 1 Gbps link, even if we just keep packet header information.

To identify the connections sharing the same switch, link, and application, we collect the information about the topology, routing, and the mapping between sockets and applications in the data center. We collect topology and routing information from the data center configuration files. To identify the mapping between the sockets and applications, we first run a script at each machine to identify the process that created each socket. We then map the processes to the application based on the configuration file for the application deployment.

To correlate performance problems across connections using the correlation algorithm we proposed in Section 2.3, we chose two seconds as the aggregation interval t to summarize the time on each performance problems to mask time difference between machines. To pinpoint transient resource constraints which usually last for minutes or hours, we chose one hour as the correlation interval T. To pinpoint problems from application code which usually last for days, we chose 24 hours as the correlation interval T. We chose the correlation threshold $\alpha = 0.4$.[5]

[5]It is hard to determine the threshold α in practice. Operators can choose the top n shared resources/application code to investigate their performance problems.

4 SNAP Validation

To validate the design of SNAP in Section 2 and evaluate whether SNAP can pinpoint the performance problems at the right place and time, we take two approaches: First, we inject a few known problems in our production data center and check if SNAP correctly catches these problems; Second, to validate the decision methods that use inference to determine the performance class in Table 2 rather than observing from TCP statistics directly, we compare SNAP results against packet traces.

4.1 Validation by Injecting Known Bugs

To validate SNAP, we injected a few known data-center networking problems and verified if SNAP correctly classifies those problems for each connection. Next, running our correlation algorithm on the SNAP logs of these labeled problems together with the other logs from the data center, SNAP correctly pinpointed *all* the labeled problems. For brevity, we first discuss two representative problems in detail and then show how SNAP pinpoints problematic host for each of them.

Problems in receive-window autotuning: We first injected a receiver-window autotuning problem: This problem happens when a Windows Server 2008 R2 machine initiates a TCP connection to a Windows Server 2003 machine with a SYN packet that requests the receiver window autotuning feature. But due to a bug in the TCP stack of the Windows Server 2003[6], the 2003 server does not parse the request for the receiver window autotuning feature correctly, and returns the SYN ACK packet with a wrong format. As a result, the 2008 server tuned its receiver window to four Bytes, leading to low throughput and long delay.

To inject this problem, we picked ten hosts running Windows 2008 in the data center and turn on their receiver window autotuning feature. Each of the ten hosts initiated TCP connections to a HTTP server running Windows 2003 to fetch 20 files of 5KB each from a host running Windows 2003.[7] It took the Windows 2003 server more than 5 seconds to transfer each 5KB file. SNAP correctly reported that all these connections are receiver window limited all the time, and SNAP logs showed that the announced receiver window size (*RWin*) is 4 Byte.

TCP incast: TCP incast [3] is a common performance problem in data centers. It happens when an *aggregator* distributes a request to a group of *workers*, and after processing the requests, these workers send the responses back at almost the same time. These responses together overflow the switch on the path and experience significant packet losses.

[6]This bug is later fixed with a patch, but some machines do not have the latest patch.

[7]We ran ten hosts to the same 2003 server to validate if the SNAP can correlate these connections and pinpoint the server.

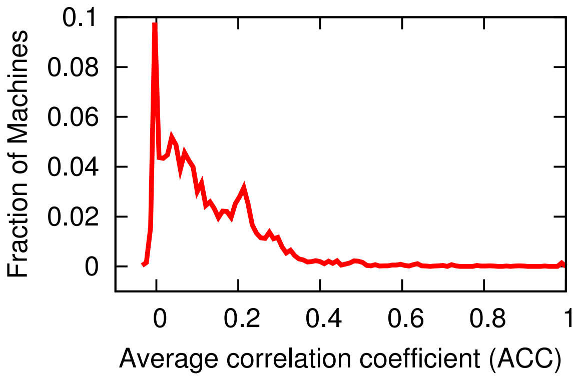

Figure 2: PDF of #Machines with different average correlation coefficient.

We wrote an application that generates a TCP incast traffic pattern. To limit the effect of our experiment to the other applications in the production data center, we picked 36 hosts under the same top-of-rack switch (TOR), used one host as the *aggregator* to send requests to the remaining 35 hosts which serve as *workers*. These workers respond with 100KB data immediately after they receive the requests. After receiving all the responses, the aggregator sends another request to the workers. The aggregator sends 20 requests in total.

SNAP correctly identified that seven of the 35 connections have experienced a significant amount of packet loss causing retransmission timeouts. This is verified from our application logs which show that it takes much longer time to get the response through the seven connections than the rest of the connections.[8]

Correlation to pinpoint resource constraints for the two problems: We mixed the SNAP logs of the receiver window autotuning problem and TCP incast with the logs of an hour period collected at all other machines in the data center. Then we ran SNAP correlation algorithm across the connections sharing the same machine.

SNAP correctly identified the Windows Server 2003 servers that have receiver-window limited problems across 5-10 connections with an average correlation coefficient (ACC) of 0.8. SNAP also correctly identified the aggregator machine because the ACC across all the connections that traverse the TOR is 0.45. Both are above

[8]In this experiment, SNAP can only tell that the connections have correlated timeouts. If the same problem happens for different aggregators running the same application code, we can tell that the application code causes the timeouts. If SNAP reports all the connections have simultaneous small writes (identified from socket call logs) and correlated timeouts, we can infer that the application code has incast problems.

the threshold $\alpha = 0.4$, which is chosen based on the discussion in Section 3.

Our correlation algorithm clearly distinguished the two injected problems with the performance of connections on the other machines in the data center. Figure 2 presents the probability density function (PDF) of the number of machines with different values of ACC. Only 2.7% of the machines have an ACC larger than 0.4. In addition to the two injected problems, the other machines with ACC > 0.4 may also indicate some problems that happen during our experiment, but we have not verified these problems yet.

4.2 Validation Against Packet Traces

We also need to validate the performance-classification algorithm defined in Table 2. The detection methods for the performance class of fast retransmissions, timeouts, receiver window limited is always accurate because these statistics are directly observed phenomena (e.g., #Timeouts) from the TCP stack. The accuracy of identifying send buffer problems is closely related to the probability of detecting the moments when the send buffer is full in the Poisson sampling, which is well studied in [15].

There is a tradeoff between the overhead and accuracy of identifying delayed ACK. The accuracy of identifying the delayed ACK and small writes classes is closely related to the estimation of the RTT. However, we cannot get per-packet RTT from the TCP stack because it is a significant overhead to log data for each packet. Instead, we get the sum of estimated RTTs (*SumRTT*) and the number of sampled packets (*SampleRTT*) from the TCP stack.

We evaluate the accuracy of identifying delayed ACK in SNAP by comparing SNAP's results with the packet trace. We picked two real-world applications from the production data center for which SNAP detects delayed ACK problems: One connection serves as an aggregator distributing requests for a Web application that has the delayed ACK problems for 100% of the packets[9]. Another belongs to a configuration-file distribution service for various jobs running in the data center, which has 75% of the packets on average experiencing delayed ACK. While running SNAP with various polling rates, we captured packet traces simultaneously. We then compared the results of SNAP with the number of delayed-ACK incidents we identify from packet traces.

To estimate the number of packets that experience delayed ACK, SNAP should find a distribution of RTTs for the sampled packets that sum up to *SumRTT*. Those

[9]This application distributes requests whose size is smaller than MSS (i.e., one packet), and waits more than the delayed ACK timeout 200 ms before sending out another request. So the receiver has to keep each packet for 200 ms before sending the ACK to the sender.

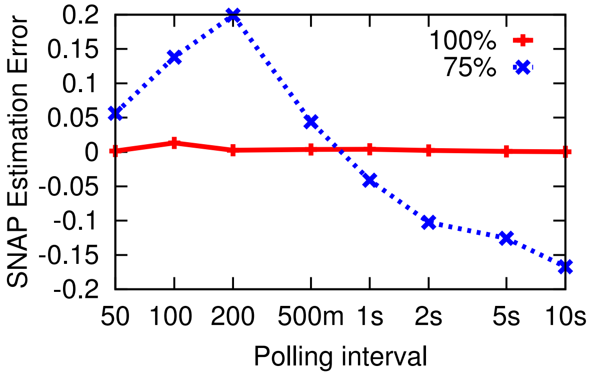

Figure 3: SNAP estimation error of identifying delayed ACK problems.

packets that experience delayed ACK have a RTT around *DelayedACKTimeout*. The rest of the packets all experience the maximum queuing delay. Therefore, we use the equation: *(diff(#SumRTT)* $-$ *diff(#SampleRTT)* * *MaxQueuingDelay)/DelayedACKTimeout* to count the number of packets experiencing delayed ACK. We use *MaxQueuingDelay* = 10 ms and *DelayedACKTimeout* = 180 ms. The delayed timeout is set as 200 ms in TCP stack, but TCP timer is only accurate at 10 ms level and thus the real *DelayedACKTimeout* varies around 200 ms. So we use 180 ms to be conservative on the delayed ACK estimation.

Figure 3 shows the estimation error of SNAP's results which is defined by $(d_t - d_s)/d_t$, where d_s is the percentage of packets that experience delayed ACK reported by SNAP and d_t is the actual percentage of delayed ACK we get from the packet trace. For the application that always has delayed ACK, SNAP's estimation error is 0.006 on average. For the application that has 75% of packets experiencing delayed ACK, the estimation error is within 0.2 for the polling intervals that range from 500 ms to 10 sec.

Figure 3 shows that the estimation error drops from positive (underestimation) to negative (overestimation) with the increase of the polling interval. When the polling interval is smaller than 200 ms, there is at most one packet experiencing delayed ACK in one polling interval. If a few packets take less than *MaxQueuingDelay* to transfer, we would overestimate the part of *SumRTT* that is contributed by these packets, and thus the rest of RTT is less than *DelayedACKTimeout*. When the polling interval is large, there are more packets experiencing delayed ACK in the same time interval. Since we have use 180 ms instead of 200 ms to detect delayed ACK, we would underestimate those packets that take longer than 180 ms delayed ACK. Nine such packets would contribute enough RTT for SNAP to assume one more delayed ACK.

5 Profiling Data Center Performance

We deployed SNAP in the production data center to characterize different classes of performance problems, and provided information to the data-center operators about problems with the network stack, network congestion or the interference between services. We first characterize the frequency of each performance problem in the data center, and then discuss the key performance problems in our data center—packet loss and the TCP send buffer.

5.1 Frequency of Performance Problems

Table 4 characterizes the frequency of the network performance problems (defined in Table 2) in our data center. Not surprisingly, the overall network performance of the data center is good. For example, only 0.82% of all the connections were receiver limited during their lifetimes. However, there are two key problems that the operators should address:

Operators should focus on the small fraction of applications suffering from significant performance problems. Several connections/applications have severe performance problems. For example, about 0.11% of the connections are receiver-window limited essentially all the time. Even though 0.11% sounds like a small number, when 8K machines are each running many connections, there is almost always some connection or application experiencing bad performance. These performance problems at the "tail" of the distribution also constrain the total load operators are willing to put in the data center. Operators should look at the SNAP logs of these connections and work with the developers to improve the performance of these connections so that they can safely "ramp up" the utilization of the data center.

Operators should disable delayed ACK, or significantly reduce Delayed ACK timeout: About two-thirds of the connections experienced delayed ACK problems. Nearly 2% of the connections suffer from delayed-ACKs for more than 99.9% of the time. We manually explore the delay-sensitive services, and count the percentage of connections that have delayed ACK. Unfortunately, about 136 delay-sensitive applications have experienced delayed ACKs. Packets that have delayed ACK would experience an unnecessary increase of latency by 200 ms, which is three orders of magnitude larger than the propagation delay in the data center and well exceeds the latency bounds for these applications. Since delayed ACK causes many problems for data-center applications, the operators are considering disabling delayed ACK or significantly reducing the delayed ACK timeout. The problems of delayed ACK for data center applications are also observed in [16].

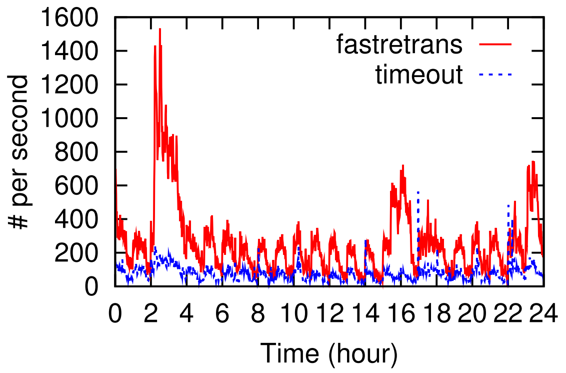

Figure 4: # of fast retransmissions and timeouts over time.

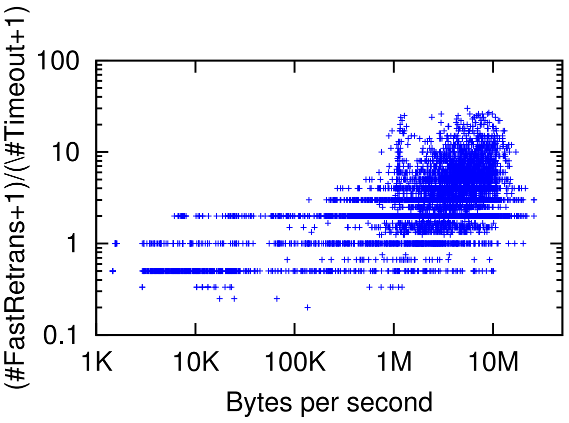

Figure 5: Comparing #FastRetrans and #Timeouts of flows with different throughput.

5.2 Packet Loss

Operators should schedule backup jobs more carefully to avoid triggering network congestion: Figure 4 shows the number of fast retransmissions and timeouts per second over time. The percentage of retransmitted bytes increases between 2 am and 4 am. This is because most backup applications with large bulk transfers are initiated in this time period.

Operators should reduce the number and effect of packet loss (especially timeouts) for low-rate flows: SNAP data shows that about 99.8% of the connections have low throughput ($<$ 1 MB/sec). Although these low-rate flows do not consume much bandwidth and are usually not the cause of network congestion, they are significantly affected by network congestion. Figure 5 is a scatter plot that shows the ratio of of fast retransmissions to timeouts vs. the connection sending rate. Each point in the graph represents one polling interval of one connection. Low-rate flows usually experience more timeouts than fast retransmission because they do not have multiple packets in flight to trigger triple duplicate ACKs. Timeouts, in turn, limit the throughput of these flows. In contrast, high-rate flows experience

Performance limitation	**% of conn. with prob. for >X% of time** >0	$>25\%$	$>50\%$	$>75\%$	$>99.9\%$	**#Apps with prob. for >X% of time** $>5\%$	$>50\%$
Sender app limited	97.91%	96.62%	89.61%	59.21%	32.61%	561	557
Send buffer limited	0.45%	0.06%	0.02%	0.01%	0.01%	1	1
Congestion	1.90%	0.46%	0.22%	0.17%	0.15%	30	6
Receiver window limited	0.82%	0.36%	0.21%	0.15%	0.11%	22	8
Delayed ACK	65.71%	33.20%	10.10%	3.21%	1.82%	154	144
(belong to delay sensitive apps)	63.52%	32.82%	9.71%	3.01%	1.61%	136	129

Table 4: Percentage of connections and number of applications that have different TCP performance limitations.

more fast retransmission than timeouts and can quickly recover from packet losses achieving higher throughput ($>$ 1 MB/sec).

5.3 Send Buffer and Receiver Window

Operators should allow the TCP stack to automatically tune the send buffer and receiver window sizes, and consider the following two factors:

More send buffer problems on machines with more connections: SNAP reports correlated send buffer problems on hosts with more than 200 connections. This is because the larger the send buffer for each connection, the more memory is required for the machine. As a result, the developers of different applications on the same machine are cautious it setting the size of the send buffer; most use the default size of 8K, which is far less than the delay-bandwidth product in the data center and thus is more likely to become the performance bottleneck.

Mismatch between send buffer and receiver window size: SNAP logs the announced receiver window size when the connection is receiver limited. From the log we see that 0.1% of the total time when the senders indicate that their connections are bottlenecked by the receiver window, the receiver actually announced a 64 KB window. This is because the send buffer is larger than the announced receiver size, so the sender is still bottlenecked by the receiver window.

To fix the send-buffer problems in the short term, SNAP could help developers to decide what send buffer size they should set in an online fashion. SNAP logs the congestion window size (*CWin*), the amount of data the application expect to send (*WriteBytes*), and the announced receiver window size (*RWin*) for all the connections. Developers can use this information to size the send buffer based on the total resources (e.g., set the send buffer size to $Cwin_{thisconn} * TotalSendBufferMemory / \sum CWin$). They can also evaluate the effect of their change using SNAP. In the long term, operators should have the TCP stack automatically tune both the send-buffer and receiver-window sizes for all the connections (e.g., [6]).

6 Performance Problems Caught by SNAP

In this section, we show a few examples of performance problems caught by SNAP. In each example, we first show how the performance problem is exposed by SNAP's analysis of socket and TCP logs into performance classifications and then correlation across connections. Next, we explain how SNAP's reports help guide developers to identify quickly the root causes. Finally, we discuss the developer's fix or proposed fix to these problems. For most examples, we spent a few hours or days to discuss with developers to understand how their programs work and to discover how their programs cause the problems SNAP detects. It then took several days or weeks to iterate with developers and operators to find out the possible alternative ways to achieve their programing goals.

6.1 Sending Pattern/Packet Loss Issues

Spreading application writes over multiple connections lowers throughput: When correlating performance problems across connections from the same application, SNAP found one application whose connections always experienced more timeouts (*diff(#Timeout)*) than fast retransmission (*diff(#FastRetrans)*) especially when the *WriteBytes* is small. For example, SNAP reported repeated periods where one connection transferred an average of five requests per second with a size of 2 KB - 20 KB, while experiencing approximately ten timeouts but no fast retransmissions.

The developers were expecting to obtain far more than five requests per second from their system, and when this report showing small writes and timeouts was shown to them the cause became clear. The application sends requests to a server and waits for responses. Since some requests take longer to process than others and developers wanted to avoid having to implement request IDs while still avoiding head-of-line blocking, they open two connections to the server and place new requests on whichever connection is unused.

However, spreading the application writes over two connections meant that often there were not enough outstanding data on a connection to cause three duplicate

ACKs and trigger fast retransmission when a packet was lost. Instead, TCP fell back to its slower timeout mechanism.

To fix the problem, the application could send all requests over a single connection, give requests a unique ID, and use pools of worker threads at each end.[10] This would improve the chances there is enough data in flight to trigger fast retransmission when packet loss occurs.

Congestion window failing to prevent sudden bursts: SNAP discovered that some connections belonging to an application frequently experience packet loss (*#FastRetrans* and *#Timeout* are both high, and correlate strongly to the application and across time). SNAP's logs expose a time sequence of socket write logs (*WriteBytes*) and TCP statistics (*Cwin*) showing that before/during the intervals where packet loss occurs, there is a single large socket write call after an idle period. TCP immediately sends out the data in one large chunk of packets because the congestion window is large, but it experiences packet losses. For example, one application makes a socket call with *WriteBytes* $>$ *100 MB* after an idle period of 3 seconds, the *Cwin* is 64 KB, and the traffic burst leads to a bunch of packet losses.

The developers told us they use a persistent connection to avoid three-way handshake for each data transfer. Since "slow start restart" is disabled, the congestion window size does not age out and remains constant until there is a packet loss. As a result, the congestion window no longer indicates the carrying capacity of the network, and losses are likely when the application suddenly sends a congestion window worth of data.

Interestingly, the developers are opposed to enabling slow start restart, and they intentionally manipulate the congestion window in an attempt to reduce latency. For example, if they send 64 KB data, and the congestion window is small (e.g., 1 MSS), they need at multiple round-trip times to finish the data transfer. But if they keep the congestion window large, they can transfer the data with one RTT. In order to have a large congestion window, they first make a few small writes when they set up the persistent connection.

To reduce both the network congestion and delay, we need better scheduling of traffic across applications, allowing delay-sensitive applications to send traffic bursts when there is no network congestion, but pacing the traffic if the network is highly used. The feedback mechanism proposed in DCTCP [17] could be applied here.

Delayed ACK slows recovery after a retransmission timeout: SNAP found that one application frequently had two problems (timeout and delayed ACK) at almost the same time. As shown in Figure 6, when the fourth packet of the transferred data is lost, the TCP sender waits for a retransmission timeout (because there are not enough following packets to trigger triple-duplicate ACKs). However, the congestion window drops to one after the retransmission. As a result, TCP can only send a *single* packet, and the receiver waits for a delayed ACK timeout before acknowledging the packet. Meanwhile, the sender cannot increase its sending window until it receives the ACK from the receiver. To avoid this, developers are discussing the possibility of dropping the congestion window down to two packets when a retransmission timeout occurs. Disabling delayed ACK is another option.

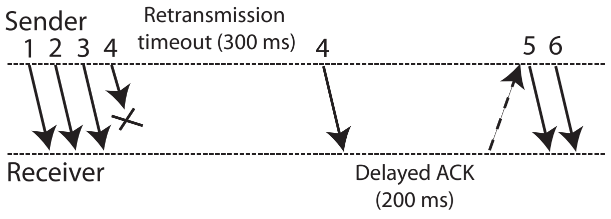

Figure 6: Delayed ACK after a retransmission timeout.

6.2 Buffer management and Delayed ACK

Some developers do not manage the application buffer and the socket send buffer appropriately, leading to bad interactions between buffer management and delayed ACK.

Delayed ACK caused by setting send buffer to zero: SNAP reports show that some applications have delayed ACK problems most of the time and these applications had set their send socket buffer length to 0. Investigation found that these applications set the size of the socket send buffer to zero in the expectation that it will decrease latency because data is not copied to a kernel socket buffer, but sent directly from the user space buffer. However, when send buffer is zero, the socket layer locks the application buffer until the data is ACK'd by the receiver so that the socket can retransmit the data in case a packet is lost. As a result, additional socket writes are blocked until the previous one has finished.

Whenever an application writes data that results in an odd number of packets being sent, the last packet is not ACK'd until the delayed ACK timer expires. This effectively blocks the sending application for 200 ms and can reduce application throughput to 5 writes per second. One team attempted to improve application performance by shrinking the size of their messages, but ended up creating an odd number of packets and triggering this issue — destroying the application's performance instead of helping it. After the developers increased the send buffer

[10]Note that the application should use a single connection because its requests are relatively small. For those applications that have a large amount of data to transfer for each request, they still have to use two connections to avoid head of line blocking during the network transfer.

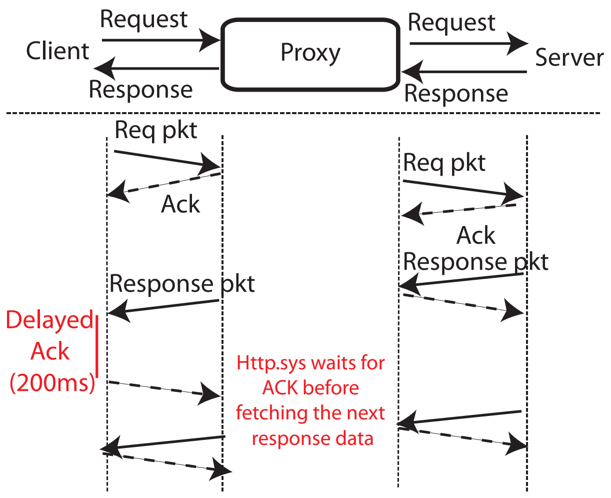

Figure 7: Performance problem in pipeline communication.

size, throughput returned to normal.

Delayed ACK affecting throughput: SNAP reports showed that an application was writing small amounts of data to the socket (*WriteBytes*) and its connections experienced both delayed ACK and sender application limited issues. For example, during 30 minutes, the application wrote 10K records at only five records per second and with a the record size of 20–100 Bytes.

The developers explained theirs is a logging application where the client uploads records to a server, and should be able generate far more than five records per second. Looking into the code with the developers, we found three key problems in the design: (i) *Blocking write:* to simplify the programming, the client does blocking writes and the server does blocking reads. (ii) *Small receive buffer:* The server calls recv() in a loop with a 200 bytes buffer in hopes that exactly one record is read in each receive call. (iii) *Send buffer is set to zero:* Since the application is delay-sensitive, the developer set send buffer size to zero. The application records are 20–100 Bytes — much less than the MSS of 1460 Bytes. Additionally, Nagle's algorithm forces the socket to wait for an ACK before it can send another packet (record).[11] As a result, the *single* packet containing each record always experience delayed ACK, leading to a throughput of only five records per second. To address this problem while still avoiding the buffer copying in memory, developers changed the sender code to write a group of requests each time. Throughput improved to 10K requests/sec after the change—a factor of 5000 improvement.

Delayed ACK affecting performance for pipelined applications: By correlating connections to the same machine, SNAP found two connections with performance problems that co-occur repeatedly: SNAP classified one connection as having a significant delayed ACK problem and the other as having sender application problems.

Developers told us that these two connections belong to the same application and form a pipeline pattern (Figure 7). There is a proxy that sits between the clients and servers and serves as a load balancer. The proxy passes requests from the client to the server, fetches a sequence of responses from the server, and passes them to the client. SNAP finds such a strong correlation between the delayed ACK problem and the receiver limited problem because both stem from the passing of the messages through the proxy.

After looking at the code, developers figured out that the proxy uses a single thread and a single buffer for both the client and the server. The proxy waits for the ACK of every transfer (one packet in each transfer most of the time) before fetching a new response data from the server.[12] When the developers changed the proxy to use two different threads with one fetching responses from the server and another sending responses to the client and a response queue between the two threads, the 99% tail of the request processing time drops from 200 ms to 10 ms.

6.3 Other Problems

SNAP has also detected other problems such as switch port failure (significant correlated packet losses across multiple connections sharing the same switch port), receiver window negotiation problems as reported in [14] (connections are always receiver window limited while receiver window size stays small), receiver not reading the data fast enough (receiver window limited), and poor latency caused by Nagle algorithm (sender application limited with small *WriteBytes*

7 Reducing SNAP CPU Overhead

To run in real time on all the hosts in the data center, SNAP must keep the CPU overhead and data volume low. The volume of data is small because (i) SNAP logs socket calls and TCP statistics instead of other high-overhead data such as packet traces and (ii) SNAP only logs the TCP statistics when there is a performance problem. To reduce CPU overhead, SNAP allows the operators to set the target percentage of CPU usage on each host. SNAP stays within a given CPU budget by dynamically tuning the polling rate for different connections.

[11] A similar performance problem caused by interactions between delayed ACK and Nagle is discussed in [10].

[12] The proxy is using the HTTP.sys library without setting the HTTP_SEND_RESPONSE_FLAG_BUFFER_DATA flag [18], which waits for the ACK from the client before sending a "send complete" signal to the application. By waiting for the ACK, HTTP.sys can make sure the application send buffer is not overwritten until the data is successfully transferred.

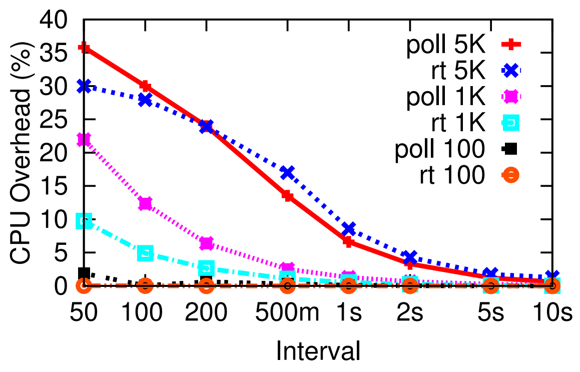

Figure 8: The CPU overhead of polling TCP statistics (*poll*) and reading TCP table (*rt*) with different number of connections (10, 100, 1K, 5K) and different intervals (from 50 ms to 10 sec).

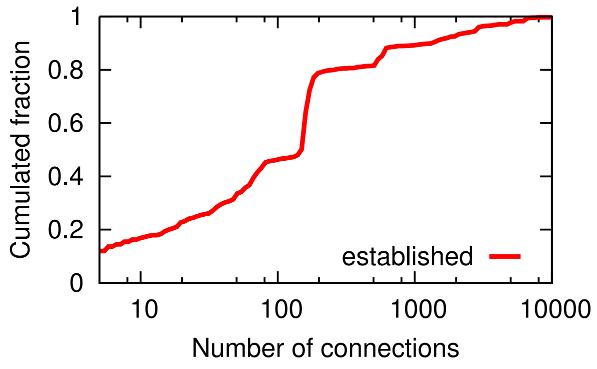

Figure 9: Number of connections per machine.

CPU Overhead of Profiling Since SNAP collects logs for all the connections at the host, the overhead of SNAP consists of three parts: logging socket calls, reading the TCP table, and polling TCP statistics.

Logging socket calls: In our data center, the cost of turning on the event tracing for socket logging is a median of 1.6% of CPU capacity [19].

Polling CPU statistics and reading TCP table: The CPU overhead of polling TCP statistics and reading the TCP table depends on the polling frequency and the number of connections on the machine. Figure 8 plots the CPU overhead on a 2.5 GHz Intel Xeon machine. If we poll TCP statistics for 1K connections at 500 millisecond interval, the CPU overhead is less than 5%. The CPU overhead of reading the TCP table is similar.

The CPU overhead is closely related to the number of connections on each machine. Figure 9 takes a snapshot of the distribution of the number of established connections per machine. There are at most 10K established sockets and a median of 150. This means operators can configure the interval of reading TCP table in most machines to be 500 millisecond or one second to keep the CPU overhead lower than 5%.[13] Since most of the connections in our data center are long-lived connections (e.g., persistent HTTP connections), we can read the TCP table at a lower frequency compared to TCP statistics polling. For the machines with many connections, we need to carefully adjust the polling rate of TCP statistics for each connection to achieve a tradeoff between diagnosis accuracy and the CPU overhead.

[13]We read TCP tables at 500 millisecond interval in our data collection.

Dynamic Polling Rate Tuning To achieve the best tradeoff between CPU overhead and accuracy, operators can first configure l_{CPU} (u_{CPU}) to be the lower (upper) bound of the CPU percentage used by SNAP. We then propose an algorithm to dynamically tune the polling rate for different connections to keep CPU overhead between the two bounds. The basic idea of the algorithm is to have high polling rate for those connections that are having performance issues and have low polling rate for the others.

We start by polling all the connections on one host at the same rate. If the current CPU overhead is below l_{CPU}, we pick a connection that has the most performance problems in the past $T_{history}$ time, and increase its polling rate for more detailed data. Similarly if the current CPU overhead is above u_{CPU}, we pick a connection that has the least performance problems in the past $T_{history}$ time, and decrease its polling rate for more detailed data. Note that a lower polling rate introduces lower diagnosis accuracy. We can still catch those performance problems with the cumulative counters, but may miss some problems that rely on snapshots to detect.

8 Related Work

Previous work in diagnosing performance problems focuses on either the application layer or the network layer. SNAP addresses the interactions between them that cause particularly insidious performance issues.

In the application layer, prior work has taken several approaches: instrumenting application code [20, 21, 22] to find the causal path of problems, inferring the abnormal behaviors from history logs [11, 12], or identifying fingerprints of performance problems [23]. In contrast, SNAP focuses on profiling the interactions between applications and the network and diagnosing *network* performance problems, especially ones that arise from those interactions.

In the network layer, operators use network monitoring tools (e.g., switch counters) and active probing tools (ping, traceroute) to pinpoint network problems such as switch failures or congestion. To diagnose network performance problems, capture and analysis of packet traces remains the gold-standard. T-RAT [24] uses packet traces to diagnosis *throughput* bottlenecks in

Internet traffic. Tcpanaly [4] uses packet traces to diagnose TCP stack problems. Others [25, 26] also infer the TCP performance and its problems from packet traces. In contrast, SNAP focuses on the multi-tier applications in data centers where it has access to the network stack, enabling us to create *simple* algorithms based on counters *far cheaper to collect than packet traces* to expose the network performance problems of the applications.

9 Conclusion

SNAP combines socket-call logs of the application's desired data-transfer behaviors with TCP statistics from the network stack that highlight the delivery of data. SNAP leverages the knowledge of topology, routing, and application deployment in the data center to correlate performance problems among connections, to pinpoint the congested resource or problematic software component.

Our experiences in the design, development, and deployment of SNAP demonstrate that it is practical to build a lightweight, generic profiling tool that runs continuously in the entire data center. Such a profiling tool can help both operators and developers in diagnosing network performance problems.

With applications in data centers getting more complex and more distributed, the challenges of diagnosing the performance problems between the applications and the network will only grow in importance in the years ahead. For future work, we hope to further automate the diagnosis process to save developers' efforts by exploring the appropriate variables to monitor in the stack, studying the dependencies between the variables SNAP collects, and combining SNAP reports with automatic analysis of application software.

Acknowledgments

We thank our shepherd Jason Flinn, the anonymous reviewers, Rob Harrison, Eric Keller, and Vytautas Valancius for their comments on earlier versions of this paper. We also thank Kevin Damour, Chuanxiong Guo, Randy Kern, Varugis Kurien, Saby Mahajan, Jitendra Padhye, Murari Sridharan, Ming Zhang for inspiring discussions on this paper.

References

[1] http://memcached.org.

[2] B. Krishnamurthy and J. Rexford, "HTTP/TCP Interaction," in *Web Protocols and Practice: HTTP/1.1, Networking Protocols, Caching, and Traffic Measurement*, Addison-Wesley, 2001.

[3] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. Andersen, G. Ganger, G. Gibson, and B. Mueller, "Safe and effective fine-grained TCP retransmissions for datacenter communication," in *ACM SIGCOMM*, 2009.

[4] V. Paxson, "Automated packet trace analysis of TCP implementations," in *ACM SIGCOMM*, 1997.

[5] http://www.ietf.org/rfc/rfc4898.txt.

[6] www.web100.org.

[7] http://msdn.microsoft.com/en-us/library/bb427395%28VS.85%29.aspx.

[8] http://msdn.microsoft.com/en-us/library/bb968803%28VS.85%29.aspx.

[9] http://datatracker.ietf.org/wg/syslog/charter/.

[10] "TCP performance problems caused by interaction between Nagle's algorithm and delayed ACK." www.stuartcheshire.org/papers/NagleDelayedAck.

[11] S. Kandula, R. Mahajan, P. Verkaik, S. Agarwal, J. Padhye, and V. Bahl, "Detailed diagnosis in computer networks," in *ACM SIGCOMM*, 2009.

[12] P. Bahl, R. Chandra, A. Greenberg, S. Kandula, D. A. Maltz, and M. Zhang, "Towards highly reliable enterprise network services via inference of multi-level dependencies," in *ACM SIGCOMM*, 2007.

[13] I. Jolliffe, *Principal Component Analysis*. Springer-Verlag, 1986.

[14] support.microsoft.com/kb/983528.

[15] C. Sarndal, B. Swensson, and J. Wretman, *Model Assisted Survey Sampling*. Springer-Verlag, 1992.

[16] A. Diwan and R. L. Sites, "Clock alignment for large distributed services," *Unpublished report*, 2011.

[17] M. Alizadeh, A. Greenberg, D. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center TCP (DCTCP)," in *ACM SIGCOMM*, 2010.

[18] http://blogs.msdn.com/b/wndp/archive/2006/08/15/http-sys-buffering.aspx.

[19] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The nature of datacenter traffic: Measurements and analysis," in *Proc. Internet Measurement Conference*, 2009.

[20] M. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer, "Path-based failure and evolution management," in *NSDI*, 2004.

[21] P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat, "Pip: Detecting the unexpected in distributed systems," in *NSDI*, 2006.

[22] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica, "X-Trace: A pervasive network tracing framework," in *NSDI*, 2007.

[23] P. Bodik, M. Goldszmidt, A. Fox, D. B. Woodard, and H. Andersen, "Fingerprinting the datacenter: Automated classification of performance crises," in *EuroSys*, 2010.

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

[25] Y.-C. Cheng, J. Bellardo, P. Benko, A. C. Snoeren, G. M. Voelker, and S. Savage, "Jigsaw: Solving the puzzle of enterprise 802.11 analysis," in *ACM SIGCOMM*, 2006.

[26] M. Tariq, A. Zeitoun, V. Valancius, N. Feamster, and M. Ammar, "Answering what-if deployment and configuration questions with WISE," in *ACM SIGCOMM*, 2008.