Packet-Level Telemetry in Large Datacenter Networks

Yibo Zhu[1,2] Nanxi Kang[1,3] Jiaxin Cao[1] Albert Greenberg[1] Guohan Lu[1] Ratul Mahajan[1]
Dave Maltz[1] Lihua Yuan[1] Ming Zhang[1] Ben Y. Zhao[2] Haitao Zheng[2]

[1]Microsoft [2]U. C. Santa Barbara [3]Princeton University

ABSTRACT

Debugging faults in complex networks often requires capturing and analyzing traffic at the packet level. In this task, datacenter networks (DCNs) present unique challenges with their scale, traffic volume, and diversity of faults. To troubleshoot faults in a timely manner, DCN administrators must *a)* identify affected packets inside large volume of traffic; *b)* track them across multiple network components; *c)* analyze traffic traces for fault patterns; and *d)* test or confirm potential causes. To our knowledge, no tool today can achieve both the specificity and scale required for this task.

We present Everflow, a packet-level network telemetry system for large DCNs. Everflow traces specific packets by implementing a powerful packet filter on top of "match and mirror" functionality of commodity switches. It shuffles captured packets to multiple analysis servers using load balancers built on switch ASICs, and it sends "guided probes" to test or confirm potential faults. We present experiments that demonstrate Everflow's scalability, and share experiences of troubleshooting network faults gathered from running it for over 6 months in Microsoft's DCNs.

CCS Concepts

•Networks $\rightarrow$ Network management;

Keywords

Datacenter network, failure detection, probe

1. INTRODUCTION

From online commerce to smartphone apps, datacenter networks (DCNs) are essential to large online services. DCNs typically operate at high utilization levels, and even small performance degradations or fault-induced downtime can lead to millions of lost revenue. These high stakes argue for a proactive model of DCN management, where infrastructure observes, analyzes, and corrects faults in near-real time.

SIGCOMM '15, August 17–21, 2015, London, United Kingdom
© 2015 ACM. ISBN 978-1-4503-3542-3/15/08…$15.00
DOI: http://dx.doi.org/10.1145/2785956.2787483

Understanding and debugging faults in DCNs is challenging because faults come in all shapes and sizes. For example:

1. Some packets may experience abnormally high delay between servers A and B, but it may not be clear which of the many links is responsible.
2. Packets destined for a specific set of servers may be dropped, even when the packet-drop counters at switches exhibit no abnormality.
3. TCP connections to a VIP (Virtual IP) may encounter intermittent timeouts, and traceroute probes to debug the issue may be blocked by the load balancers.
4. Load may not be balanced among a group of ECMP (Equal Cost Multi Path) links; the network administrators do not know if this issue is due to flow size differences or a deeper problem (e.g., a poor hash function).

The diagnosis of such DCN faults requires examining network behavior at the granularity of packets. Problems such as faulty interfaces or switch software bugs can produce seemingly random failures that impact specific groups of packets based on any combination of characteristics, such as route taken in the network, packet headers, or timing. As a result of their subtle effects, these failures are often difficult to debug through analyses that rely on flow-level information [7], aggregate counters [5, 36], or sampled traffic [29]. We refer to tracing, collection and analysis of packet-level traffic behavior as *packet-level network telemetry*.

Building a responsive packet-level telemetry system for large DCNs is challenging for at least three reasons. First, today's DCNs carry unprecedented levels of network traffic. A large DCN usually has over 100,000 servers, each with a 10 to 40 Gbps network connection. At high utilization levels, aggregate traffic can easily exceed 100 Tbps. Analyzing even tiny subsets of this data is intractable for today's commodity switches, and moving traces to commodity servers for analysis would congest or even disrupt the network.

Second, DCN faults often occur over multiple hops or switches, and effective diagnosis requires intelligently tracing small subsets of packets over the network, as well as the ability to search the packet traces based on sophisticated query patterns, *e.g.,* protocol headers, sources and destinations, or even devices along the path. This task is not only akin to searching in the proverbial haystack for needles, but for specific needles of arbitrary size, shape and color. Existing systems that rely on packet-level analysis [14, 31] indiscriminately trace packets; they cannot scale to the size of large DCNs or search for packets based on complex patterns.

Finally, passive tracing alone, which captures an instantaneous snapshot of the network, has limited effectiveness. The traces observed in the snapshot may not be enough to identify whether the problem is transient or persistent, and they may not provide enough information to localize the problem. For example, when we see a packet trace stops before reaching its final destination, we may not be able to tell if this is due to a random drop or a blackhole.

We present Everflow, a network telemetry system that provides scalable and flexible access to packet-level information in large DCNs. To consistently trace individual packets across the network, Everflow uses "match and mirror." Commodity switches can apply actions on packets that match on flexible patterns over packet headers or payloads; we use this functionality with mirroring packets to our analysis servers as the action. By installing a small number of well-chosen match-and-mirror rules means we can reap the benefits of packet-level tracing while cutting down the overhead by several orders of magnitude. To quickly search for patterns in large volumes of packet traces, we build a scalable trace collection and analytics pipeline. We leverage switch-based load balancing [12] to distribute tracing and processing across multiple servers while preserving flow-level locality for efficient analysis. Finally, Everflow supports *guided probes*—packets that are specially crafted and injected into the network to follow preset paths. Guided probes help validate the performance and behavior of individual devices or links.

Everflow has been deployed since August 2014 in part of Microsoft's DCN infrastructure, including a cluster of 37 switches and a larger cluster of 440 switches. Both clusters carry a wide range of application traffic. We have also deployed Everflow selectively on-demand to other production clusters to help debug tricky network faults. We capture our deployment experiences by describing several representative debugging cases in §7, including latency problems, packet drops, routing loops, ECMP load imbalance, and protocol-specific issues around RDMA (Remote Direct Memory Access). In each case, we describe the observed symptoms, steps taken, and how a solution was found using Everflow.

We also perform detailed microbenchmarks to quantify the performance of Everflow along key aspects, including packet analysis rate, bandwidth and storage overhead, and overall system scalability. Our evaluations consistently show that all components of Everflow impose low overheads, and scale well to large DCNs with 100 Tbps of traffic.

In developing Everflow, we sought to address a real need for scalable, packet-level telemetry which can debug faults that are hard to tackle via conventional techniques. We prioritized usability and simplicity over functionality or features, *e.g.,* Everflow works on commodity switches and requires no specialized hardware. Our experiences with Everflow demonstrate that it scales well to large DCNs, and provides significant benefits to network administrators.

2. PACKET-LEVEL NETWORK TELEMETRY

Operational DCNs comprise a wide range of hardware and software components, including multiple types of switches, load balancers, and servers dedicated for processing and storage. Faults can and do arise from any single or combination of components, which makes debugging quite challenging in practice. In this section, we describe a number of sample faults that commonly occur in large DCNs, to illustrate why conventional tools are insufficient and why packet-level network telemetry is useful.

Silent packet drops: drops not reported by the culprit switch (*e.g.,* discard counters are zero). This situation may occur due to software bugs or faulty hardware on the switch. Although such drops can be detected on end hosts (*e.g.,* by monitoring TCP retransmissions), it is quite tricky to localize the culprit switch using conventional tools because of the large number of switches in DCNs. With consistent tracing of specific packets across switches, we can immediately locate the last hop switch where the victim packets appear as well as their expected next hop switch(es). We then send guided probes to each next hop switch to confirm the culprit.

Silent blackhole: the type of routing blackhole that does not show up in forwarding tables. Therefore it cannot be detected by tools that examine forwarding table entries [22, 23]. Silent blackhole can happen due to corrupted entries in TCAM tables. Packet-level network tracing will allow us to detect and localize a silent blackhole similar to how we deal with silent packet drops.

Inflated end-to-end latency: high latency for a flow. It is yet another problem that is easy to detect by end hosts, but can be difficult to debug using conventional tools. With packet-level network traces across switches, this problem becomes trivial, since we can obtain hop-by-hop latencies between the two end points.

Loops from buggy middlebox routing: routing problems caused by middleboxes instead of switches (or routers). This may happen when a middlebox incorrectly modifies a packet and its forwarding behavior (see §7.3). Such a problem cannot be found by examining the switch routing or forwarding tables because the tables are all correct. Given the network trace of a packet, we can easily identify such a problem as the trace will violate basic invariants such as loop-freedom.

Load imbalance: the problem of (5-tuple) flows being forwarded unevenly by a group of ECMP links. The naive detection method of comparing load counters at ECMP links can have false positives, since link load differences may actually be caused by differences in flow size (to be expected). Even when load imbalance is confirmed, the counters are too coarse-grained to answer key questions for debugging, such as "is the imbalance caused by flows that match specific patterns?" A packet-level network telemetry system allows us to count the number of specific 5-tuple pattern flows mapped to each link and thus offers a more reliable and direct method of detection and debugging.

Protocol bugs: bugs in the implementation of network protocols such as BGP, PFC (Priority-based Flow Control) and RDMA (Remote Direct Memory Access). When a protocol misbehaves, performance and reliability of the network suffers. Troubleshooting protocol bugs is tricky, because many protocols are implemented by third-party (switch and NIC) vendors and cannot be easily instrumented. Tracing the packets of these protocols offers a reliable yet independent way to identify protocol misbehaviors. Here network telemetry is particularly suitable because host-based tracing may be too expensive (*e.g.,* for RDMA) or unavailable (*e.g.,* for PFC and BGP).

3. OVERVIEW OF EVERFLOW

This section outlines the challenges in designing a scalable packet-level network telemetry system and introduces the key ideas in Everflow to address these challenges.

3.1 Design challenges

Tracing and analysis scalability. The first challenge facing packet-level network telemetry for large DCNs is scalability of trace collection. As mentioned earlier, the traffic in a large DCN (with 100,000+ servers) can easily go beyond 100 Tbps. Tracing packets at this scale will consume a tremendous amount of network and server resources. Consider a typical scenario where the average packet size is 1,000 bytes, each mirrored packet is truncated to 64 bytes (the minimum Ethernet frame size), and the network diameter is 5 hops (in a typical 3-tier Clos topology). If we simply trace every packet at every hop, as proposed in Packet History [14], the tracing traffic will be $\frac{64B}{1000B} \times 5(hops) \times 100(Tbps) = 32(Tbps)$.[1] Such a high rate of tracing traffic may cause congestion and packet drops, especially when network utilization is already high.

A similar same scalability challenge applies to trace analysis. Because commodity switches have limited storage and CPU power, traced packets must be sent to servers for analysis. Even assuming a server can process tracing traffic at line rate (10 Gbps), we will still need $\frac{32(Tbps)}{10(Gbps)} = 3,200$ servers for analysis which is prohibitively expensive.

Done naively, the situation can be worse. For most types of analysis (see §4.1 for details) require the per-hop trace of the same packets to be sent to the same analysis server. This stickiness can be achieved by having switches send traced packets to a set of reshufflers which then reshuffle the packets according to the hash of the packet header. Adding these reshuffling servers will double the total number of servers.

Limitations of passive tracing. In practice, passive tracing alone may not provide enough information for network troubleshooting. Fig 1(a) shows a scenario where a packet p is last seen at switch S_1 but not at the next hop switch S_2. Although this situation indicates that p is dropped at S_2, one cannot determine whether the problem is transient (which could be ignored) or persistent (which requires attention). One possible solution is to correlate multiple traces with dropped packet at S_2. However, there may not be enough traces available for such correlation because: i) the victim flows may have very few packets; ii) only a small fraction of packets are traced due to sampling. Even if we could know it is a persistent drop event, we will still have difficulty in figuring out whether this is a random drop (*e.g.,* due to a faulty cable) or a blackhole that only drops the packets of certain 5-tuples.

Fig 2(a) shows another scenario where a packet p traverses switches S_1 and S_2 and the two traced packets are sent to an analyzer A. To avoid the clock synchronization problem on S_1 and S_2, we try to calculate the latency of link (S_1, S_2) using the timestamps of the two traced packets on A. However, this cannot be done because the path $S_1 \to A$ can be quite different from the path $S_2 \to A$.

3.2 Key ideas

Everflow addresses these challenges with four key ideas. The first three tackle the scalability challenge while the last one overcomes the limitations of passive tracing.

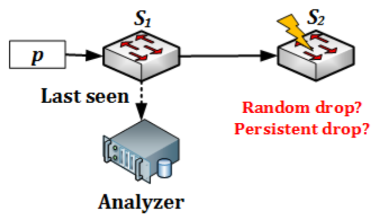
Match and mirror on switch. Commodity DCN switches can match based on pre-defined rules and then execute certain actions (*e.g.,* mirror and encapsulate), which do not change original packet forwarding behavior. Everflow leverages this capability to reduce tracing overhead. Specifically, we design three types of matching rules to handle common DCN faults (§2). This rule set is by no means exhaustive and can be expanded to other types of faults.

First, we design matching rules to capture every flow in DCNs. The most straightforward rule is to randomly *trace* 1 out of n packets,[2] which is heavily biased towards large flows. In a DCN where the flow size distribution is highly skewed, this approach will miss many small flows. Yet these small flows are often associated with customer-facing, interactive services with strict performance requirements. To cover small flows, we configure a new set of rules that match based on the TCP SYN, FIN and RST fields in packets. Since DCN traffic is typically dominated by TCP [21], these rules allow us to trace every TCP flow in DCNs.

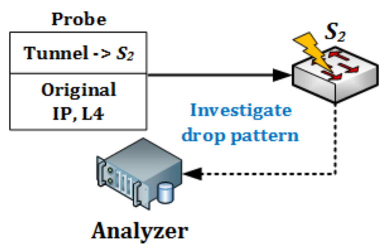
Second, we configure additional matching rules to enable flexible tracing. Basic TCP matching may not catch every type of fault, *e.g.,* the packet drops in the middle of a TCP flow. In fact, the exact set of traced packets that are needed depends on the nature of the fault. For instance, we may want to trace the packets of a particular application, with a specific port number, or between certain pairs of servers. To support such flexible tracing, we allow the packets to be marked by a special "debug" bit in the header. The marking criteria can be defined in any manner, as long as the total tracing overhead stays below a threshold. In the switches, we install a rule to trace any packet with the "debug" bit set, which is similar to how a software developer sets the

[1]Packet History [14] proposed techniques to reduce the bandwidth overhead. But these techniques require heavy modifications to both switching ASIC and the host networking stack and are thus hard to deploy in current networks.

[2]This differs from the random sampling in sFlow [29] – Everflow ensures the same set of packets will be sampled and traced across all switches while in sFlow different switches may sample a different set of packets. That is one of the reasons why we cannot use sFlow for tracing in our system.

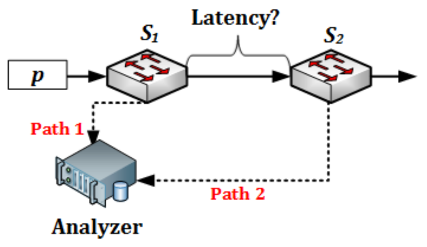


(a) Passive tracing is insufficient

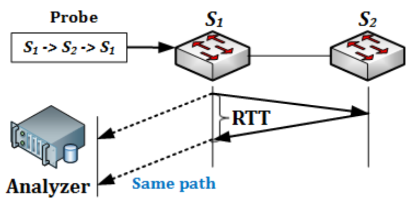


(b) Guided probing confirms the problem

Figure 1: Debugging packet drops

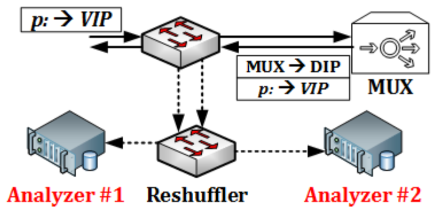


(a) Extracting link latency from passive tracing is hard

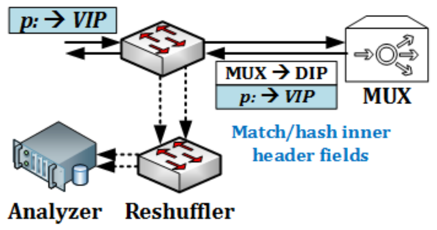


(b) Guided probing can accurately measure link latency

Figure 2: Measuring link latency



(a) Encapsulation breaks the packet trace



(b) Obtaining the complete packet trace by matching on inner header

Figure 3: Handling packet encapsulation

debug flag during compilation to enable fine-grained tracing in their code. As a result, Everflow can efficiently trace any subset of regular data packets.

Finally, our tracing coverage goes beyond data packets. A DCN has a small amount of traffic associated with network protocols such as BGP, PFC, and RDMA. We call it *protocol traffic* to distinguish from regular data traffic. Although the absolute volume of the protocol traffic is small, it is critical for the overall DCN health and performance. Thus, Everflow has rules to trace all the protocol traffic.

Scalable trace analyzers. Although matching rules limit tracing overhead, the total amount of tracing traffic can still be huge because of the sheer scale of the DCN. To reduce analysis overhead, we observe that at any time only a tiny fraction of the traced packets (<0.01%) will exhibit abnormal behaviors (*e.g.,* loops or drops). This observation motivates us to treat abnormal packet traces differently from normal traces, *e.g.,* keep detailed, per-hop state for the former and aggregate, per-device state for the latter. As we explain in §4.1, this differential treatment helps reduce analysis overhead by three orders of magnitude. If needed, we can selectively recover the lost information in the normal traces via guided probes (described below).

Switch-based reshuffler. As mentioned in §3.1, we need a low-cost way to reshuffle a large volume of tracing traffic. We leverage work on turning a commodity switch into a hardware Mux (HMux) for load balancing [12]. The idea is to first define a VIP (virtual IP) on an HMux, which will map to a set of DIPs (direct IPs) where each DIP corresponds to an analysis server. We then configure all switches to send traced packets to the VIP. When a traced packet p reaches the HMux, the HMux redirects p to a DIP based on the hash of p's 5-tuple. This ensures that the traced packets of the same 5-tuple will be redirected to the same DIP.

An HMux can reshuffle traffic using the full forwarding capacity of a switch (typically > 1 Tbps). This is at least 100 times faster than a server with a 10 Gbps NIC, and dramatically cuts down the cost of reshuffling. We can further increase reshuffling capacity by configuring multiple HMuxes with the same VIP [12].

We need to pay special attention to encapsulated packets. (For now, let us ignore traced packets that are also encapsulated. We explain how to handle them in §6.1.) Packet encapsulation is often used in DCNs for load balancing [28] and network virtualization [24]. Fig 3(a) shows an example of how a SLB (Software Load Balancer) Mux may break the trace analysis. The original destination IP of a packet p is a VIP (Virtual IP). A Mux will encapsulate p with a new DIP (Direct IP) as the destination. Because of this, the traced packets of the original p and the encapsulated p are sent to two different analyzers, and are processed separately. To address this problem, we install a rule that matches on the inner header of an encapsulated packet, and configure HMuxes to hash based on inner header fields. This allows us to trace p's complete path from source to DIP (see Fig 3(b)).

Guided Probing. As mentioned earlier, a packet drop could happen due to multiple reasons. Sometimes, passive tracing alone may be insufficient to disambiguate between these possibilities. This ambiguity leads to the following question: what if we could arbitrarily replay a packet trace? More specifically, what if we could inject any desired packet into any desired switch and trace the behavior of the injected packet (by setting its debug bit)? We call this *guided probing* and will describe it in more detail in §6.2.

One immediate use of guided probing is to recover the lost tracing information due to trace sampling or aggregation. To recover the trace of any packet p, we simply need to inject p into the first hop switch (with its debug bit set).

Further, guided probing is useful in overcoming the limitations of passive tracing. In the example of Fig 1(b), we can inject multiple copies of p into switch S_2 to see whether p is dropped persistently or not. In addition, we can craft probe packets with different patterns (*e.g.,* 5-tuples) to see if the drops are random or specific to certain 5-tuples. Such probing cannot debug transient faults because they may disappear before probing is initiated. We consider this limitation acceptable because persistent faults usually have a more severe impact than transient ones.

We extend guided probing such that it can not only inject a packet into a desired switch but also cause a packet to traverse a desired sequence of hops (similar to source routing). This extension allows us to measure the roundtrip latency of any link in the network, as illustrated in Fig 2(b). A probe packet p is instructed to traverse $S_1 \rightarrow S_2 \rightarrow S_1$. Because p traverses S_1 twice, S_1 will generate two traced packets of p at time t_1 and t_2 separately, and $t_2 - t_1$ equals to the roundtrip latency of link (S_1, S_2).

Since many commodity switches today do not provide the timestamping function, we cannot obtain t_1 and t_2 directly. However, observer that the two traced packets of p are close in time (*e.g.,* within 1 ms) and take exactly the same path from S_1 to the analyzer. Thus we can use the difference in their arrival time at the analyzer to approximate $t_2 - t_1$.

4. TRACE COLLECTION AND ANALYSIS

We now present the trace collection and analysis pipeline of Everflow. As shown in Fig 4, it consists of four key components: the *controller*, *analyzer*, *storage* and *reshuffler*. On top of that, there are a variety of *applications* that use the packet-level information provided by Everflow to debug network faults. The controller coordinates the other components and interacts with the applications. During initialization, it configures the rules on switches. Packets that match these rules will be mirrored to the reshufflers and the directed to the analyzers which output the analysis results into storage. The controller also provides APIs which allow Everflow applications to query analysis results, customize counters on the analyzers, inject guided probes, and mark the debug bit on hosts. We describe the analyzer and controller in this section and the reshuffler and storage in §6.

4.1 Analyzers

The analyzers are distributed set of servers, each of which processes a portion of tracing traffic. The reshufflers will balance the loads among the analyzers and ensure that the traced packets of the same 5-tuple are sent to the same analyzer (§3.2). Each analyzer keeps two type of states: packet trace and counter.

Packet trace. The analyzer keeps a table of packet traces where each trace is a chain of the mirrored instances of the same original packet. A trace is uniquely identified by the 5-tuple and the IPID of the original packet. It has one copy of the full packet content and a set of per-hop information,

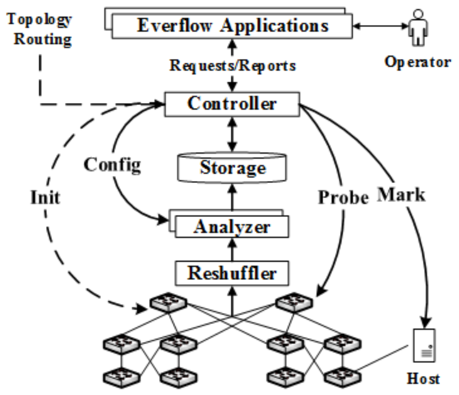


Figure 4: Everflow architecture

including the IP address of the switch where the packet is mirrored, timestamp, TTL, source MAC address (to identify the previous hop), and DSCP/ECN. A trace is considered complete when no new packet arrives for 1 second (which is much smaller than the end-to-end latency inside a DCN).

For each complete packet trace, the analyzer checks for two types of problems: loop and drop. A loop exhibits as the same device appearing multiple times in the trace. A drop is detected when the last hop of the trace is different from the expected last hop(s) which can be computed using the DCN topology and routing policy. For example, the expected last hop of a packet destined to an internal IP address of the DCN is the ToR switch directly connected to the IP address. The expected last hops of a packet destined to an external IP address are the border switches of the DCN.

To correctly handle packet encapsulation due to SLB (see Fig 3(a)), we merge the traces of an original packet p_o and an encapsulated packet p_e if the 5-tuple and IPID of p_e's inner IP header match those of p_o's IP header.

Despite of the use of match and mirror, the amount of packet traces can still be huge. To further reduce the storage overhead, each analyzer will only write to storage the traces that exhibit abnormal behaviors (*e.g.,* loop or drop), have the debug bit set (*e.g.,* guided probes), or correspond to the protocol traffic (*e.g.,* PFC and BGP). For the other traces (which represent a vast majority of all traces), each analyzer will aggregate them into the types of counters listed below and then periodically (*e.g.,* once every 10 seconds) write these counters into the storage. In the end, the controller combines the counters from individual analyzers into the final ones.

Link load counters. For each link, the analyzer will compute the aggregate load (*e.g.,* number of packets, bytes and flows) from the packet traces. Besides that, it may also compute more fine-grained load counters, *e.g.,* the load generated by certain prefixes or by intra-DC traffic. These fine-grained load counters can be dynamically added or removed by Everflow applications via the controller.

Latency counters. The analyzer will compute the latency of each link from the traces of guided probes (see §6.2). For any packet trace that traverses a SLB Mux, it will also compute the latency of the Mux. This process is shown in

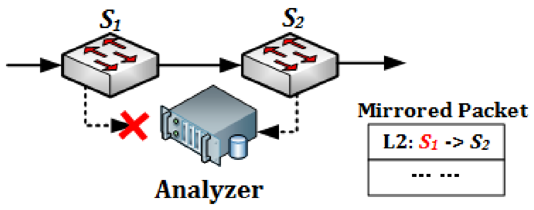


Figure 5: Detecting the drop of mirrored packets

Fig 3(b) where a Mux server is connected to a ToR switch that mirrors both the original packet p_o and the encapsulated packet p_e. Because the mirrored instances of p_o and p_e will take the same path from the ToR switch to the same analyzer (as explained in §3.2), we can estimate the Mux latency using the arrival time difference between them at the analyzer. The estimated Mux latency includes the round trip latency of the link between the ToR switch and the Mux, which is negligibly small compared to the Mux latency. To save space, the analyzer will quantize individual latency samples into predefined bins in a latency histogram.

Mirrored packet drop counters. A mirrored packet may be dropped before reaching the analyzer. We can often infer such a drop from the packet trace. Fig 5 shows that a packet p traverses switches S_1 and S_2. However, p's trace contains only S_2 but not S_1, clearly indicating that p's mirrored packet from S_1 is dropped. In our current deployment, we found the drop rate is low (around 0.001%).

Sometimes mirrored packets may be dropped due to congestion near a particular reschuffler or analyzer. To further increase the reliability of our trace collection pipeline, we deploy multiple reshufflers and analyzers in different parts of the DCN and shift mirrored traffic away from any congested reshuffler and/or analyzer that exhibits a high mirrored packet drop rate.

4.2 Controller APIs

Everflow applications interact with the controller via several APIs to debug various network faults. With these APIs, the applications can query packet traces, install fine-grained load counters, trigger guided probes, and selectively trace traffic by marking the debug bit.

GetTrace(Filter, Condition, StartTime, EndTime) is used to get the packet traces between $StartTime$ and $EndTime$. The $Filter$ parameter specifies the types of traced packets to be filtered and is similar to the widely-used display filter in Wireshark. It allows filtering based on the Ethernet, IP, TCP, or UDP headers of the original packets as well as the outer IP header of the mirrored packets (which contains the IP address of the switch that sends the mirrored packets as shown in Fig 6). For example, the $Filter$ "ip.proto == 6 && ip.dst == 192.16.0.0/16 && switch == 10.10.0.10" matches all the TCP packets going to 192.16.0.0/16 and mirrored by switch 10.10.0.10. The $Condition$ parameter specifies the properties of the traces that cannot be extracted from the packet headers. For example, it allows filtering based on whether the traces contain a drop or a loop, or have a SLB Mux latency larger than 1 ms (if the traces traverse a Mux).

GetCounter(Name, StartTime, EndTime) is used to retrieve the counter values between $StartTime$ and $EndTime$. Each counter is identified by a descriptive $Name$, such as "SwitchX_PortY_TCP".

AddCounter(Name, Filter) & RemoveCounter(Name) are used to dynamically add or remove fine-grained load counters. The $Filter$ parameter is the same as above.

Probe(Format, Interval, Count) is used to launch guided probes. The probing results can later be retrieved via $GetTrace()$ and $GetCounter()$. The $Interval$ and $Count$ parameters specify the frequency and total number of probes to send. The $Format$ parameter specifies the format of the probe packets, including the L3 and L4 headers. It is similar to the $Filter$ parameter described above, with a minor change to support packet encapsulation. For example, "ip.src == SIP1,SIP2 && ip.dst == DIP1,DIP2 && ip.proto == 6" defines an IP-in-IP encapsulated TCP probe, whose outer source IP is SIP1, outer destination IP is DIP1, inner source IP is SIP2 and inner destination IP is DIP2.

EnableDbg(Servers, Filter) & DisableDbg(Servers, Filter) are used to mark or unmark packets with the debug bit on certain servers. The $Filter$ parameter is the same as above. The controller accepts the $EnableDbg()$ request if the total amount of traced traffic does not exceed system capacity.

5. EVERFLOW APPLICATIONS

Writing applications using Everflow APIs is straightforward. We now present several example applications to debug the network faults described in §2.

Latency profiler. Many DCN services, *e.g.,* search and distributed memory cache, require low latency. To find out why the latency between any pair of servers is too high, the latency profiler will first mark the debug bit of the TCP SYN packets between the two servers. From the traces of these packets, it knows the network devices on the path and then launches guided probes to measure the per-hop latency. Guided probing measures the roundtrip latency of each link instead of the one-way latency. This degree of localization suffices in practice. With this localization information, the profiler can quickly identify the network devices that cause the problem.

Packet drop debugger. Packet drops can severely degrade application performance, causing low throughput, timeouts or even unreachability. They are notoriously difficulty to debug as they happen due to many different reasons such as congestion, software bugs, or configuration errors. Our packet drop debugger routinely examines the packet traces that show packet drops. Given such a trace of a packet p, the debugger will infer the next hop switch S_n based on the last hop where p is captured. For example, S_n can be inferred either from p's output interface at the last hop switch or from the DCN topology and routing. After that, it will inject guided probes to S_n to determine whether the drops are persistent and, if so, whether the drops are random or have any patterns (*e.g.,* specific 5-tuples).

Loop debugger. Loops are uncommon in DCNs. However, when they do appear, they can cause unnecessary waste of resources or connectivity problems. Our loop debugger watches for packet traces that contain a loop. When a loop is detected, it first injects guided probes to see if the loop is persistent. If so, it reports the list of devices in the loop to the operators who can then break the loop by disabling one of the device interfaces. During this process, the debugger can continue to inject guided probes until the loop disappears.

ECMP profiler. In DCNs, switches often use ECMP to split traffic to the next hops. The load split may be uneven due to poor hash functions or routing problems, causing link congestion. For each switch, our ECMP profiler will monitor the aggregate load of all the links. When an uneven load split is detected, it will drill down through more fine-grained load counters to find out whether the uneven split impacts all traffic or just a subset (*e.g.,* the traffic from/to certain prefixes). The profiling results help the operators quickly detect and localize the problem.

RoCEv2-based RDMA debugger. RoCEv2-based [18] RDMA (Remote Direct Memory Access) is an emerging protocol for achieving high throughput (40 Gbps) and ultra-low latency (several microseconds) with low CPU overhead. By leveraging PFC (Priority-based Flow Control) to enable a drop-free Ethernet fabric, the RDMA protocol implementation can be simplified and offloaded to the NIC. However, in our DCNs, we find that RDMA sometimes cannot attain its ideal performance due to software bugs in the NIC. Debugging these problems is hard because the NICs are built by third-party vendors and we have limited means to instrument the RDMA code on the NICs.

We build a RDMA debugger in Everflow. It traces all the control packets related to RDMA, such as PFC and NACK (Negative Acknowledgement). The control packet traces offer a reliable and yet independent way not only to observe the actual behavior of RDMA flows but also to debug the implementation issues inside the third-party vendor's code.

6. IMPLEMENTATION

The entire Everflow system is implemented in roughly 10K lines of code in C++. The five Everflow applications are written in roughly 700 lines of code in C# in total. Below, we omit the details of the controller (whose implementation is fairly straightforward) and the reshuffler (which is similar to the Mux described in Duet [12]); we focus on other aspects instead.

6.1 Switch configurations

By default, we configure rules in the TCAM table to match on TCP SYN/FIN/RST flags. We use a bit in the DSCP field as the debug bit, and n bits in the IPID field to sample 1 out of 2^n packets. For example, by configuring a rule to match on 10 bits in the IPID field, we will sample 1 out of 1,024 packets. Since every switch has the same rules, the set of sampled packets will be consistent across all switches. For any encapsulated packet, we configure rules to match on its inner TCP/IP headers to ensure that it is sent to the same

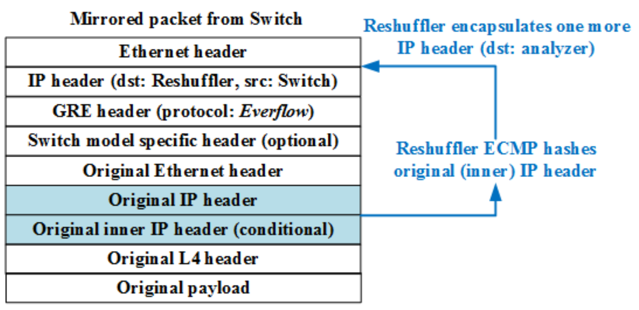

Mirrored packet from Switch
Ethernet header
IP header (dst: Reshuffler, src: Switch)
GRE header (protocol: Everflow)
Switch model specific header (optional)
Original Ethernet header
Original IP header
Original inner IP header (conditional)
Original L4 header
Original payload
Reshuffler encapsulates one more IP header (dst: analyzer)
Reshuffler ECMP hashes original (inner) IP header

Figure 6: Format of mirrored packet

analyzer as its original packet (§3.2). Finally we configure rules to match on Ethernet type 0x8808 (L2 control packets including PFC), TCP port 179 (BGP packets), and RDMA NACK [17]. The total number of rules is around 20, which consumes only a small fraction of the TCAM table.

When a packet matches any rule, the switch will mirror it and encapsulate the mirrored packet using GRE (Generic Routing Encapsulation). Fig. 6 shows the format of the GRE packet, where the source IP is the switch loopback IP, the destination IP is the VIP of the reshufflers, and the payload is the original packet (starting from the L2 header). Inside the GRE header, there is a protocol field which is used to indicate that this is an Everflow mirrored packet. We configure every switch with a blacklist rule to prevent mirroring a mirrored packet.

"Match and mirror" is completely done in a switch's data plane. This implementation leverages the huge packet processing capacity of switching ASIC and incurs zero overhead on a switch's CPU.

6.2 Guided prober

The key function of a prober is to inject any desired packet into any target switch S. It uses the raw socket APIs to craft arbitrary packet fields, *e.g.,* the IP and L4 (TCP, UDP, ICMP, etc.) headers. The crafted packet p has the debug bit set to enable tracing and carries a signature in the payload so that it can be easily identified by the Everflow analyzer. To send p to S, we leverage the decapsulation capability that is widely available on commodity switches. We first create the probe packet p' by encapsulating p with S's loopback IP as the destination, and send p' out. We also configure a rule on S to decapsulate any encapsulated packet destined to S's loopback IP address. Thus upon receiving p', S will decapsulate p' into p and then process p according to the normal forwarding logic.

In fact, we can extend the technique above to instruct p' to follow any desired route by encapsulating p multiple times. This can be used to measure the latency of any link (S_1, S_2) as shown in Fig 2(b). We simply need to craft p with S_1 as the destination, encapsulate it with S_2 as the destination, and encapsulate it again with S_1 as the destination. The resulting p' will follow the route $S_1 \rightarrow S_2 \rightarrow S_1$ as required in §3.2.

To prevent guided probe packets from interfering with server applications, we deliberately set their TCP or UDP checksum incorrectly so that they will be discarded by servers.

6.3 Analyzer

The analyzers use a custom packet capturing library to capture mirrored packets. The library supports RSS (Receiver Size Scaling) [2] which allows an analyzer to receive packets using multiple CPU cores. The library hashes packets to CPU cores based on source and destination IPs, using inner source and destination IPs if packets are encapsulated. We run multiple analysis threads to maximize throughput.

6.4 Storage

The Everflow storage is built on top of SCOPE [6]—a scalable, distributed data processing system. In SCOPE, data is modeled as tables composed of rows of typed columns. These tables can be processed by SQL-like declarative scripts, which support user-defined operators, such as extractors (parsing and constructing rows from a file), processors (row-wise processing), reducers (group-wise processing), and combiners (combining rows from two inputs).

We store packet traces in a multi-column table, where each row corresponds to a packet trace. The columns contain three parts of information about the packet trace. The first part is the full packet content. The packet header and payload are stored in separate columns to simplify processing. The second part is the per-hop information, *e.g.,* timestamp, TTL, source MAC address and DSCP-ECN. Since the number of hops is variable, we combine all the per-hop information into one column. The last part is the metadata of the trace, including the trace length, whether it is a guided probe, or whether it has a loop or a drop. The traces can be retrieved by the controller based on the $filter$, $condition$, and time range defined in §4.2.

We also store the counters from all the analyzers in a table. Each row in the table represents one snapshot of a counter from an analyzer. Besides the key and value of the counter, a row also contains the analyzer's ID and the timestamp when the snapshot was taken. To respond to a counter query, the controller will sum up the counter values of the rows that match the given counter name and time range.

7. DEPLOYMENT AND EXPERIENCE

We deployed Everflow in two Microsoft DCN clusters in August 2014. The first one is a full deployment in a pre-production cluster (Cluster A) with 37 switches. The second one is a pilot deployment to 440 out of more than 2,500 switches in a production cluster (Cluster B). Both clusters carry traffic for many DC applications. In addition, we also enabled Everflow on certain production switches on demand to debug live incidents. Currently, we are extending Everflow deployment to more switches and clusters. In the following, we will share our experience in using Everflow to debug a variety of common DCN faults (§2).

7.1 Latency problem

A multi-tier search application complained about large latency jitters between a set of servers. The application allocates a strict per-tier latency bound of 1 ms, which includes server processing time. Due to the use of ECMP routing in our DCN (Fig 7(a)), there are hundreds of links that could cause this problem. It was extremely difficult for operators to identify the culprit switch or link.

One possibility was to use traceroute to infer per-link latency. However, traceroute measures the RTT between a probing server and a switch, which is impacted by all the links on the forward and reverse paths. Moreover, because the traceroute probe is processed by the switch CPU, the measured RTT will be inflated by the switch control plane delay which may be highly variable. Thus, the RTT measured by traceroute will be very noisy and even unusable.

Our latency profiler was called for help. It marked the debug bit for the application traffic sent or received on the afflicted servers, and learned the links on the path. It subsequently sent out guided probes to measure all the relevant links in parallel. Fig 7(b) plots a portion of the latency timeseries on a subset of the links. Intermittently, Link A would have a much larger latency variability than the other links, sometimes even approaching 0.8 ms. This left little time for servers to complete their computations. After further investigation, the operators confirmed that this problem was caused by a misconfigured ECN (Explicit Congestion Notification) threshold that allowed excessive queue build up.

Summary. Unusually high latency is one of the most common problems in DCNs and is hard to debug using conventional tools. The fine-grained latency measurements provided by Everflow help operators localize the problem in minutes instead of hours.

7.2 Packet drops

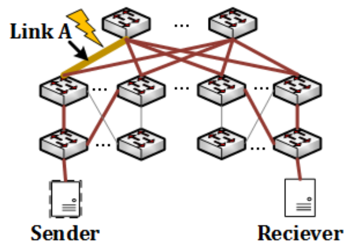
7.2.1 Blackhole

Many clients of an internal web service reported that a fraction of their connection establishment requests were encountering timeouts. However, a few retrials of the connection requests would eventually succeed. This led to the violation of the SLAs (Service Level Agreements) of the web service and raised alerts to the network operator.

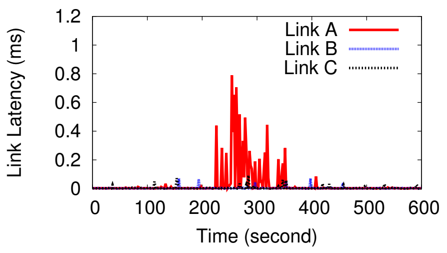
The operator suspected that this problem was due to packet drops. However, he had no clue where the drops happened because the web service was behind a VIP (virtual IP) which was hosted on multiple software Muxes. A client's request could be sent to any one of the Muxes and then redirected to any one of the many DIPs (direct IPs). This means the drops could happen anywhere between the clients and the Muxes, on one of the Muxes, or between the Muxes and the DIPs.

He first checked the error counters on the Muxes but found no problem. Then he checked the counters on a number of switches, some of which showed minor drops. But none of the switch counters was significant enough for him to reach a reliable conclusion. At this point, he was stuck and had to elevate the alert.

We ran the packet drop debugger to investigate this issue. From the drop debugger, we observed that many SYN packet traces that did not reach the DIPs after traversing the Muxes and that all these abnormal traces stalled one hop before the



(a) Many links may be responsible for the latency problem



(b) Latency profiling results

Figure 7: Profiling link latency

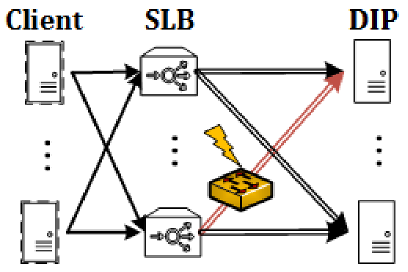


Figure 8: A switch blackholes packets to a DIP behind a SLB Mux

same switch S. The drop debugger subsequently launched guided probes, with the same 5-tuples as the dropped SYN packets, to S to validate the drops. The results showed that all the SYN packets destined to a specific DIP were dropped by S, suggesting that S was blackholing the packets to the DIP (Fig 8).

Informed by our analysis, the operator analyzed the suspect switch thoroughly and found that one of the forwarding entries in the TCAM table was corrupted. Any packet that hit the corrupted entry would be dropped. Even worse, the switch did not report such drops in any counter. After a reboot, the switch started behaving normally and the timeout problem went away as well.

7.2.2 *Silent packet drop*

Many DC applications in Cluster B detected abnormally high levels of TCP retransmissions. At the same time, the network-wide end-to-end monitoring system also showed that the ping probes between certain server pairs were experiencing around 2% loss rate. This caused severe performance degradation to the applications running in the network.

As was usual, the operator started with the switch counters but did not see anything particularly wrong. Given that the end-to-end probes did show drops, he suspected that this was caused by one switch silently dropping packets. The conventional approach to debug such a problem is to use tomography, *e.g.,* inferring the rough location of the culprit switch by sending many pings and traceroutes to exhaustively explore the paths and links in the network. After narrowing down to a few suspects, the operator would try to pinpoint the culprit by disabling and enabling the suspects one by one. However, since the DCN has nearly 25,000 links, such a brute-force approach will impose significant overhead and can easily take several hours.

Our packet drop debugger simplified the debugging process. From all the packet traces in the storage, it first extracted the ones that encountered drops during the same time period. Based on where the problematic traces stopped, the debugger quickly identified one suspect switch which appeared to be the next hop of most of the problematic traces. It then sent 1,000 guided probes to test each interface on the switch. In the end, the root cause was found to be one faulty interface on the switch that silently dropped a fraction of packets at random. All the application alerts stopped after the faulty interface was disabled.

7.2.3 *Packet drops on servers*

In this incident, one internal application alerted that the throughput between its senders and receivers was 15% lower than normal. The application used UDP as the transport protocol and had many senders and receivers. All the receivers reported roughly equal amount of throughput loss. To isolate the UDP packet drops, the sender's team, receiver's team and network team were involved.

Due to the large number of senders, the sender's team could only sample a few senders and capture packets on those senders. From the captured traces, they confirmed that the packets were indeed sent. The receiver's team did the same thing on several receivers and found that packets were partially received. However, because the loads on the receivers were pretty high, they could not determine whether the missing packets were dropped by the network or by the receivers' NIC. At this stage, neither the network nor the receiver's teams could be released from debugging.

To find out where the packets were dropped, the network team used the Everflow drop debugger to mark the debug bit on a subset of the senders' traffic. From the resulting packet traces, the debugger showed that all the marked UDP packets successfully traversed the network and reached the last hop ToR (Top-of-Rack) switches to which the receivers were connected. Thus, the packet drops were happening at the receivers. After more investigation, the problem was found to be a recent update on the senders that caused some messages to use a new format which were not recognized by the receivers. Without Everflow, the network team would have a hard time proving that the drops were not in the network.

Summary. Packet drop is a common source of performance faults in DCNs. Localizing drops is extremely challenging given the large number of devices involved (servers, switches, Muxes, etc.). Everflow offers an efficient and reliable way to track packets throughout the network and dramatically simplifies the drop debugging process.

7.3 Loop

Under normal circumstances, we do not expect to see any loops in a DCN. Surprisingly, the Everflow loop debugger did catch quite a few loop incidents in Cluster B. All the loop traces showed the same pattern — they involved a packet that kept bouncing back and forth between a SLB Mux and a DIP until its TTL became 0.

Figure 9 shows how a loop forms. Initially a request packet to VIP_0 is sent to a SLB Mux, which then encapsulates and sends the packet to a DIP. When the DIP server receives and decapsulates the packet, it finds that the inner header destination IP (VIP_0) does not match its designated VIP. Instead of discarding the packet, the DIP server throws the packet back to the network. As a result, this packet is sent back to the Mux again, forming a persistent loop.

We call it an *overlay* loop since it results from inconsistent views between two servers (Mux and DIP). Unlike a network-level loop, an overlay loop cannot be detected by examining the forwarding table on switches. In fact, the forwarding tables are all correct. At the same time, an overlay loop can result in unnecessary waste of network and server resources or even unavailability. In these incidents, the loops caused 30x amplification of the affected traffic. The traffic trapped in the loops accounted for roughly 17% of the total traffic received by the problematic DIPs.

Summary. Contrary to conventional wisdom, loops can form in a DCN due to the wide use of server overlays that participate in packet forwarding. Everflow provides an effective way to detect and debug such loops, which allows the loops to be fixed before they trigger severe failures.

7.4 Load imbalance

The SNMP link monitor reported imbalanced loads on the links from four leaf switches to a ToR switch in Cluster A. Fig 10(b) shows the part of the topology that is related to this incident. Because the loads on these links were affected by all the upstream switches of the ToR, the operator had no clue about why this imbalance happened.

The Everflow ECMP profiler was used to debug this problem. It started by requesting fine-grained load counters from the Everflow analyzers and breaking down the load by traffic towards physical server IP prefixes *vs.* traffic towards SLB VIP prefixes. It found that the traffic to physical servers was indeed balanced, but the VIP traffic was doing just the opposite. To understand this phenomenon, the profiler further broke down the traffic by individual VIP prefixes under the ToR. It turned out that only the traffic destined to three VIP prefixes was imbalanced. As shown in Fig 10(a), the load of the three VIP prefixes on link $Leaf_4 \rightarrow ToR$ is 2.6 times that on link $Leaf_3 \rightarrow ToR$.

A follow-up investigation showed that this incident was due to a failure in BGP route advertisement. Specifically, the first three leaf switches failed to advertise the routes for these three VIP prefixes to some of its upstream spine switches (Fig 10(b)). This led to the uneven split of the traffic going to the three VIP prefixes, starting from the spine switches down to the leaf switches.

Summary. Load imbalance can arise due to a variety of reasons. The Everflow ECMP profiler supports flexibly classifying the traffic and providing detailed load information per traffic class. Therefore, it can not only detect load imbalance incidents but also help identify their causes. The latter cannot be easily done based on the aggregate load counters.

7.5 Low RDMA throughput

Our DCN is deploying RoCEv2-based RDMA. However, the RDMA engineering team found that RDMA performed poorly when there were a small amount of packet losses (*e.g.,* due to a switch interface bug or packet corruption). For example, even with 0.01% loss rate, the throughput of a RDMA flow would drop below 10 Gbps (i.e., less than 25% of the optimal throughput of 40 Gbps). Such degradation was far worse than expected.

Initially, the engineers attempted to debug this problem by capturing the RDMA packets on both ends of the servers using a customized tool provided by the RDMA NIC vendor. However, due to the high data rate, the tool would impose significant capturing overhead on the NIC and also miss a fraction of the packets. As a result, when the engineers examined the packet dumps and saw no bad symptom, they could not tell whether this was because the flow behavior had changed (due to the extra capturing overhead) or because there was not enough diagnostic information. Moreover, the tool can capture only L3 packets but not L2 packets like PFC.

We used our RDMA debugger to investigate this problem. The throughput of an RDMA flow is affected by three types of control packets: PFC (Priority-based Flow Control), NACK (Negative Acknowledgment) and CNP (Congestion Notification Packet), all of which are captured by Everflow. From the control packet traces, we saw an unexpected correlation between the PFC and NACK packets. When a receiver generated a NACK in response to a packet loss, it almost always generated another PFC packet with a long PAUSE period. The effect of the PAUSE was propagated hop-by-hop back to the sender,[3] and ultimately slowed the sender.

We reported this problem to the NIC vendor who later found a bug in their NIC driver. This bug caused the NIC to halt the buffer, and subsequently triggered a PFC with a long PAUSE period during the NACK generation.

Summary. Everflow provides a reliable and yet independent way to observe and debug network protocol misbehaviors. This is particularly useful when host-based capturing of protocol packets is too expensive (*e.g.,* for RDMA) or unavailable (*e.g.,* for PFC).

8. SYSTEM EVALUATION

We evaluate Everflow's capacity, overhead and deployment cost and show that it can easily scale to a large DCN with 10K switches, 250K links and 100 Tbps traffic. In the analysis below, we ignore the switch overhead because

[3]To avoid buffer overflow, a switch uses PFC to force its upstream neighbor (another switch or a server NIC) to PAUSE data transmission.

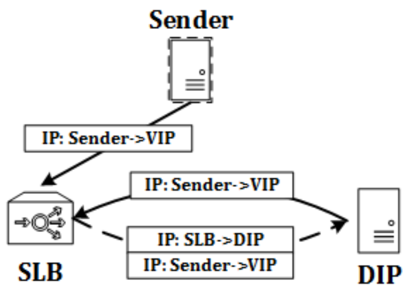
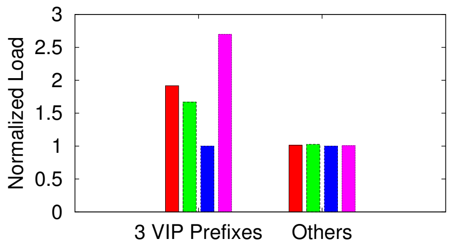
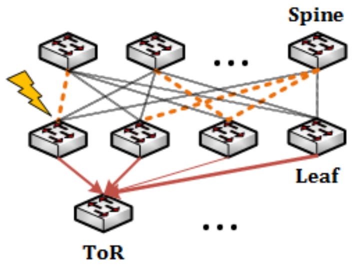


Figure 9: An overlay loop formed between a SLB Mux and a DIP



(a) Traffic to 3 VIP prefixes is imbalanced



(b) Imbalance due to prefix announcement failure from leaf switches to spine switches

Figure 10: Load imbalance incident

match-and-mirror is done completely at the switch's data plane and incurs zero overhead on the switch's CPU (§6.1).

8.1 Capacity

Analyzer. We use servers with Xeon 16-core, 2.1 GHz CPU, 128 GB memory and 10 Gbps Ethernet NIC. Depending on the packet size, the bottleneck may be the CPU (capped by number of packets per second, or *pps*) or Ethernet NIC.

For small packets, our packet capturing library achieves around 600K pps per thread. With RSS support, which scales capturing throughput linearly with the number of CPU cores, our library is capable of capturing 4.8M pps using 8 CPU cores.[4] Because our analysis task is light weight, one CPU core can easily analyze more than 600K pps. Thus we run 8 capturing threads and 8 analysis threads, with one thread per core. In total, each server can process ***4.8M pps*** with 16 cores.

For large packets, the NIC becomes the bottleneck. If all packets are 1,500 bytes (standard MTU in DCNs), a server with a 10 Gbps NIC will handle only $10Gbps/8/1500 = 0.83Mpps$. Meanwhile, memory is never a bottleneck. Since the analyzer only buffers packets that arrive in the last second (§4), maximum memory usage is $10Gbps \times 1s = 1.25GB$.

Reshuffler. The reshuffler processes packets at a switch's line rate. Using a switch with $128 \times 10Gbps$ ports, a reshuffler can support 1.28 Tbps or 10 billion pps.

8.2 Overhead

Bandwidth. From our current deployment, the mirrored traffic volume is only ***0.38%*** of the DCN traffic and the average mirrored packet size is 156.4 bytes. Currently we do not truncate packets due to the limitations of commodity switches used in our DCN. Given 100 Tbps of DCN traffic, the mirrored traffic requires ***380 Gbps*** network bandwidth and ***300M pps*** CPU processing power.

Storage. We breakdown the storage overheads into counters and traces. Because the number of links is much larger than the number of switches or Muxes, the link counters dominate all the other counters. For each link, we have 6 load counters (3 in each direction) and 4 latency counters (for latency histogram). Given 250K links, the total number of counters is 2.5M. Since each counter takes 20 bytes and the update interval is 10 seconds, each analyzer consumes only $2.5M \times 20B/10s = 5MB/s$ for uploading all the counters into storage.

In our deployment, analyzers record less than 0.01% of all traces, since most normal traces are aggregated into counters (§4.1). Thus uploading packet traces requires only $380Gbps \times 0.01\%/8 = 4.75MB/s$ in total. Such a data rate can be easily handled by the distributed storage service.

Query delay. The response time of answering a query by an Everflow application mainly depends on the data volume being queried. We benchmark the *GetTrace()* query on 10 GB traces (roughly equal to the amount of traces generated in 30 minutes), and the SCOPE job finishes in one minute. We observe similar query delay for *GetCounter()*. For all the incidents presented in §7, we never need to query more than 10 GB of data.

8.3 Deployment cost

Since most mirrored packets are small, the analyzers are bottlenecked by the CPU. Given a total of 300M pps (or 380 Gbps) mirrored traffic, we need $300Mpps/4.8Mpps = 63$ analyzers. Furthermore, a single switch-based reshuffler is sufficient to handle all the mirrored traffic. We currently use two reshufflers for failure resilience.

9. RELATED WORK

Our work builds on several themes of related work.

Mirroring. Like Everflow, Planck [31] and PacketHistory [14] monitor network through switch mirroring. In Planck, switches mirror traffic at over-subscribed ports to a directly attached server. It focuses on packets at a single device instead of network-wide packet traces. PacketHistory mirrors all packets at switches to remote servers. For large scale DCNs, its approach incurs significant overhead to transmit and process all traffic. It may also cause congestion and hurt ongoing applications. In comparison, to lower overhead, Everflow uses "match" to select the right traffic to mirror.

Sampling. Many recent works [7, 8, 29, 33, 36] focus on sampling network traffic. NetFlow [7] and OpenSketch [36]

[4]Advanced packet capturing libraries like DPDK [1] NetMap [32] and WireCap [34] can support more than 20Mpps using 8 cores.

aggregate information (*e.g.,* number of bytes) at flow level and do not provide fine-grained, packet-level information. sFlow [29] randomly samples packets at switches. It only provides per-device information and cannot track packet paths across multiple devices. Another work [8] proposes sampling packets with a consistent hash function. It does not cover what packets to sample or how to analyze sampled packets. In contrast, Everflow presents a flexible and rule-based end-to-end solution that traces packets of interest across the network and analyzes the traces in real time.

Probing. Per-packet tracing tools such as traceroute, ICMP, and Tulip [25] can track a packet's path in the network. A recent work [27] proposes using IP pre-specific timestamp option to dissect RTT, but is limited to specific portions of the path in the Internet. These works can be classified as out-of-band diagnosis because they use packets that are different from the actual application. In comparison, Everflow focuses on in-band tracing.

OFRewind [35] proposes "record and replay", which is another way of probing. Everflow has a similar capability, but it is more flexible than merely replaying past traffic. For example, Everflow can instrument a probe to bounce between switches for measuring latency.

Further, existing tools cannot trace beyond middleboxes (*e.g.,* load balancers and VNet gateways), which can encapsulate or modify packets. Everflow correlates packets entering and leaving middleboxes to construct full packet traces.

Fault detection. Several works [9, 10, 22, 23] detect network faults by checking routing configuration and policy. They analyze routing rules and validate compliance with network policy (*e.g.,* no forwarding loops). They rely on control plane information and focus on verifying the forwarding tables. As a result, they cannot diagnose packet level problems. Other works monitor virtual switches [26] or use operational data, such as router syslog [3, 11, 15, 30] or AS route dissemination [13], for fault detection. They are complementary to Everflow because they focus on different aspects than Everflow, which targets fine-grained packet-level faults based on data plane tracing.

10. CONCLUSION AND FUTURE WORK

We present Everflow, a scalable packet-level telemetry system for large DCNs. Everflow leverages switch's "match and mirror" capability to capture consistent traces of packets across DCN components and injects guided probes to actively replay packet traces. For fault debugging applications, Everflow provides flexible query interfaces and configurable APIs on top of an efficient and light weight analytics pipeline. Through deployment in Microsoft's DCNs, Everflow demonstrated its utility as an indispensable tool for troubleshooting DCN faults.

In the future, we plan to extend Everflow to make use of the features offered by new programmable switching ASIC [4, 20], including hardware timestamping and packet metadata matching. Switch timestamping will dramatically simplify the measurement of link roundtrip latency, and packet metadata matching (*e.g.,* based on checksum error or parity error) will provide insight into the reason of packet drops. We also plan to extend Everflow to the cloud provider's wide area network [16, 19], where, similar to DCNs, servers and network traffic are under its control.

Acknowledgements

We thank our Microsoft colleagues, especially Paul Wang, Yongqiang Xiong, George Chen, Kamil Cudnik and Ahsan Arefin, for their help in building and deploying Everflow. We also thank the reviewers and our shepherd Geoff Voelker for their constructive feedback. Y. Zhu, H. Zheng and B. Y. Zhao are supported in part by NSF grants IIS-1321083, CNS-1224100, and CNS-1317153.

11. REFERENCES

[1] Data plane development kit. http://www.dpdk.org/.

[2] Receive side scaling. https://msdn.microsoft.com/en-us/library/windows/hardware/ff567236(v=vs.85).aspx.

[3] A. Arefin, A. Khurshid, M. Caesar, and K. Nahrstedt. Scaling data-plane logging in large scale networks. In *MILCOM*, 2011.

[4] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *SIGCOMM*, 2013.

[5] J. Case, M. Fedor, M. Schoffstall, and J. Davin. RFC 1157: Simple network management protocol.

[6] R. Chaiken, B. Jenkins, P.-Å. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. Scope: easy and efficient parallel processing of massive data sets. *VLDB*, 2008.

[7] B. Claise. RFC 3954: Cisco systems netflow services export version 9 (2004).

[8] N. G. Duffield and M. Grossglauser. Trajectory sampling for direct traffic observation. *IEEE/ACM Trans. Netw.*, June 2001.

[9] S. K. Fayaz and V. Sekar. Testing stateful and dynamic data planes with flowtest. In *HotSDN*, 2014.

[10] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein. A general approach to network configuration analysis. In *NSDI*, 2015.

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

[12] R. Gandhi, H. H. Liu, Y. C. Hu, G. Lu, J. Padhye, L. Yuan, and M. Zhang. Duet: Cloud scale load balancing with hardware and software. In *SIGCOMM*, 2014.

[13] N. Gvozdiev, B. Karp, and M. Handley. Loup: who's afraid of the big bad loop? In *HotNets*, 2012.

[14] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *NSDI*, 2014.

[15] C.-Y. Hong, M. Caesar, N. Duffield, and J. Wang. Tiresias: Online anomaly detection for hierarchical operational network data. In *ICDCS*, 2012.

[16] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer. Achieving high utilization with software-driven WAN. In *SIGCOMM*, 2013.

[17] Infiniband Trade Association. InfiniBand Architecture Volume 1, General Specifications, Release 1.2.1, 2008.

[18] Infiniband Trade Association. Supplement to infiniband architecture specification volume 1 release 1.2.2 annex A17: RoCEv2 (ip routable ROCE), 2014.

[19] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: Experience with a globally-deployed software defined WAN. In *SIGCOMM*, 2013.

[20] V. Jeyakumar, M. Alizadeh, Y. Geng, C. Kim, and D. Mazières. Millions of little minions: Using packets for low latency network programming and visibility. In *SIGCOMM*, 2014.

[21] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken. The nature of datacenter traffic: measurements & analysis. In *IMC*, 2009.

[22] P. Kazemian, M. Chan, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real time network policy checking using header space analysis. In *NSDI*, 2013.

[23] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. Veriflow: Verifying network-wide invariants in real time. In *NSDI*, 2013.

[24] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, N. Gude, P. Ingram, E. Jackson, A. Lambeth, R. Lenglet, S.-H. Li, A. Padmanabhan, J. Pettit, B. Pfaff, R. Ramanathan, S. Shenker, A. Shieh, J. Stribling, P. Thakkar, D. Wendlandt, A. Yip, and R. Zhang. Network virtualization in multi-tenant datacenters. In *NSDI*, 2014.

[25] R. Mahajan, N. Spring, D. Wetherall, and T. Anderson. User-level Internet path diagnosis. In *SOSP*, 2003.

[26] V. Mann, A. Vishnoi, and S. Bidkar. Living on the edge: Monitoring network flows at the edge in cloud data centers. In *COMSNETS*, 2013.

[27] P. Marchetta, A. Botta, E. Katz-Bassett, and A. Pescapé. Dissecting round trip time on the slow path with a single packet. In *PAM*, 2014.

[28] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, and R. Kern. Ananta: cloud scale load balancing. In *SIGCOMM*, 2013.

[29] P. Phaal, S. Panchen, and N. McKee. RFC 3176: Inmon corporation's sflow: A method for monitoring traffic in switched and routed networks, 2001.

[30] T. Qiu, Z. Ge, D. Pei, J. Wang, and J. Xu. What happened in my network: mining network events from router syslogs. In *IMC*, 2010.

[31] J. Rasley, B. Stephens, C. Dixon, E. Rozner, W. Felter, K. Agarwal, J. Carter, and R. Fonseca. Planck: Millisecond-scale monitoring and control for commodity networks. In *SIGCOMM*, 2014.

[32] L. Rizzo. netmap: A novel framework for fast packet I/O. In *USENIX ATC*, 2012.

[33] J. Suh, T. Kwon, C. Dixon, W. Felter, and J. Carter. Opensample: A low-latency, sampling-based measurement platform for SDN. In *ICDCS*, 2014.

[34] W. Wu and P. Demar. Wirecap: a novel packet capture engine for commodity NICs in high-speed networks. In *IMC*, 2014.

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

[36] M. Yu, L. Jose, and R. Miao. Software defined traffic measurement with opensketch. In *NSDI*, 2013.