# Buffer-based End-to-end Request Event Monitoring in the Cloud

Kaihui Gao[⋆†], Chen Sun[†], Shuai Wang[⋆], Dan Li[⋆],
Yu Zhou[†], Hongqiang Harry Liu[†], Lingjun Zhu[†], Ming Zhang[†]
[⋆]*Tsinghua University* [†]*Alibaba Group*

## Abstract

Request latency is a crucial concern for modern cloud providers. Due to various causes in hosts and networks, requests can suffer from request latency anomalies (RLAs), which may violate the Service-Level Agreement for tenants. However, existing performance monitoring tools have incomplete coverage and inconsistent semantics for monitoring requests, resulting in the difficulty to accurately diagnose RLAs.

This paper presents BufScope, a high-coverage request event monitoring system, which aims to capture most RLA-related events with consistent request-level semantics in the end-to-end datapath of request. BufScope models the datapath of request as a buffer chain and defines RLA-related events based on different properties of buffers, so as to *end-to-end monitor* the root causes of RLA. To achieve *consistent semantics* for captured events, BufScope designs a concise request-level semantic injection mechanism to make events captured in networks have the victim requests' ID, and offloads the realization to SmartNICs for *low overhead*. We have implemented BufScope on commodity SmartNICs and programmable switches. Evaluation results show that BufScope can diagnose 95% RLAs with <0.07% network bandwidth overhead and <1% application throughput decline.

## 1 Introduction

With the emergence of cloud-native architecture [1], application-layer requests (*e.g.,* RPC, HTTP, and RESTful requests) become a fundamental component in the cloud [2]. The request latency is the total elapsed time across a request end-to-end datapath, including the application, the end-host network stack and the underlying network. Since request latency directly affects the performance of many distributed applications [3], it has become a crucial concern [4–6] for cloud providers. Besides, request-level information is the tie between the tenants and the cloud providers. For instance, when a request (*e.g.,* search, storage I/O) encounters a surge of latency, the tenant will provide the operators with the request-level descriptive information to diagnose the anomaly [7].

Request latency anomalies (RLAs), which cause long-tailed request latency, are not rare in clouds. According to the data of a block storage cluster with over 40,000 servers from a prominent global cloud provider *Alibaba*, we observe that across all the 440 million RPC requests in one hour, 0.01% of them (44K) suffer from a latency of >100 ms, which violates the Service-Level Agreement (SLA) of the storage service. Cloud providers need to accurately diagnose RLAs to explain SLA violations, otherwise revenue loss will be caused [8].

However, accurately diagnosing RLAs is challenging, because it requires high coverage for request-level abnormal events (*i.e., request events*). Specifically, cloud providers must be able to capture as many abnormal events that happen on the *end-to-end* datapath of requests as possible, *e.g.,* data pause, congestion, drop, *etc.*, which are direct triggers of RLAs. Moreover, the captured events should be mapped to *request-level semantics* (*e.g.,* RPC ID), rather than the flow- or packet-level. In practice, it is non-trivial to extract the request-level semantics from flow- or packet-level data.

Unfortunately, existing performance or latency monitoring tools are far from satisfying the preceding requirements for diagnosing RLAs. Specifically, though distributed application performance tracing tools [9–16] can provide any request-level timing data, they cannot capture the request events below the application layer; Network performance monitoring tools [8, 17–25] can capture flow-level events that happen in the network stack or the underlying networks, such as delayed ACK, packet drop, and path change, but the captured events have no request-level information, so the events captured in applications and networks cannot be associated.

Since these existing monitoring tools have incomplete coverage and inconsistent semantics, the cloud providers often get into trouble when diagnosing RLAs. For instance, based on the RLA information reported by the tracing tool, the application owners or tenants often naturally blame server and network team [8, 24]. However, due to the mismatch of monitoring semantics, these teams have to associate the events obtained from their own monitoring tools with the RLA through coarse-grained time-correlation methods [24], which is not

only inefficient, but also inaccurate (§2.1).

The fundamental reason why existing tools fail to achieve the high coverage is that the traditional data plane in datacenter networks is a black box [8]. The request events which happen in the data plane cannot be detected and parsed as flexibly as those in the host software. Consequently, network monitoring tools typically capture accessible and coarse-grained flow- or packet-level events. This makes it difficult to end-to-end monitor RLAs with consistent request-level semantics. Fortunately, recent advances on the commodity programmable data plane provides a new foundation to improve the situation.

This paper presents BufScope, a high-coverage request event monitoring system. The main idea of BufScope is to translate most RLAs to buffer-related abnormal events, monitor all buffers in the request's end-to-end datapath, and capture all buffer-related abnormal events with consistent request-level semantics. Specifically, BufScope achieves *end-to-end monitoring* and *consistent request-level semantics* through two core designs which keep *low overhead* in mind.

**(i) Buffer event modeling.** The main purpose of buffer is to deal with the mismatch between upstream and downstream processing rates. If upstream or downstream processing has an anomaly, the buffer will reveal the corresponding abnormal events, such as queue buildup, data pause, and packet out-of-order [26]. Based on the operational experience in *Alibaba*, we observe that most (>90%) RLAs reveal abnormal events in buffers (§3.1). The remaining (<10%) RLAs that come from NIC flapping, link corruption, bugs, *etc.*, are very difficult and inefficient to cover. For low overhead consideration, in this work we only cover RLAs with buffer-related abnormal events. Thus, BufScope models the end-to-end datapath of request as a buffer chain (§3.2), including the application, network stack, NIC and switch, and monitors RLA-related abnormal events that happen in all the buffers.

However, these buffers may have different RLA-related abnormal events, and pre-defining all the events for all types of buffers is challenging. For example, in lossy Ethernet, packets may be dropped before entering the buffer, while in lossless Ethernet, the upstream switch will pause packet forwarding to avoid the packet drop in the downstream switch. In response, BufScope uniformly classifies all buffers in both hosts and networks according to three properties, including priority awareness, order sensitivity, and enqueue feature. Based on these properties, BufScope defines a complete buffer event library, including packet drops, congestion, pause, out-of-order and priority contention (§3.3). Then, operators can monitor the corresponding events in a buffer based on its properties.

**(ii) Request-level semantic injection.** The lack of request-level semantics in the network is because the request header may not exist in the packet payload; even if it exists, its location in the payload is not fixed. This causes commodity programmable switch to fail to extract the request-level information when generating abnormal events. In response, BufScope designs a concise semantic injection mechanism, which just inserts the offset of the first request header (if it is in the payload) at the end of the packet header (§3.4). Then, based on the location-specific information, programmable switches can iteratively parse all request identifier in a packet.

A straightforward approach to injecting request-level semantic is to implement the function in the end-host network stack. However, the overhead of this strawman design is significant for applications that adopt run-to-completion (RTC) model (§5.3). RTC model packs the entire logic (including application and network stack processing) in one single thread to achieve ultra-low latency, which is quite common for large-scale datacenter applications [7, 27]. To reduce the impact of request-level semantic injection on the application performance, we offload the operation to hardware.

We have integrated BufScope in an open-source RPC system and *Alibaba*'s production storage application with Broadcom PS225 SmartNICs and Barefoot Tofino switches. Testbed-based evaluation shows that BufScope can diagnose 95% RLAs (64% for the combined solution of existing state-of-the-art monitoring tools [8, 15, 21]) with <0.07% network bandwidth overhead (>4% for the baseline) and <1% application throughput decline (4.3% for the baseline).

## 2 Background and Motivation

In this section, we firstly use representative experiences of *Alibaba*to demonstrate the RLAs in production. Then, we analyze the limitations of existing monitoring tools to accurately diagnose RLAs. Finally, we present this paper's motivation.

### 2.1 RLAs in the Cloud

There are generally two types of performance anomalies for one request: connectivity loss and RLA. The former means the client loses connectivity to the remote server for seconds to minutes, due to issues such as hardware failure, program corruption, or network outage. The latter type of anomaly means that, even though the request can be finally completed, the latency for this request is larger than expected, which compromises the SLA. We focus on the monitoring of RLA, which is easy to happen but very difficult to mitigate. This is because RLAs are usually caused by abnormal events in a shorter time-scale with randomness, leaving minor fingerprints for monitoring and locating. Potential root causes could be polling hang and badly-tuned network stack parameters in hosts, congestion and packet drop in networks [8, 17, 24], *etc.*

To understand the impact of RLA on application performance, we have conducted experiments using the *Alibaba*'s production block storage application, which adopts RPC framework, user-level network stack and RTC model. One front-end server constantly performs 4KB file read operations from the storage back-end over RPC request. We simulate RLAs by adding microsecond-level latency to the RPC pro-

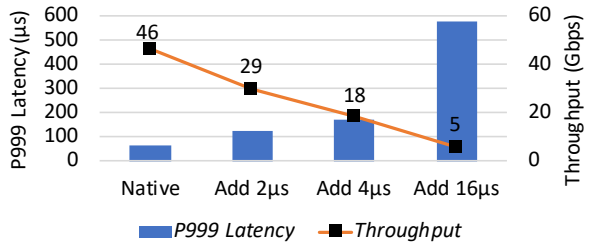



Figure 1: Impact of RLAs on application performance.

cessing logic, and measure its affection on the application regarding to end-to-end tail latency and overall throughput.

As shown in Figure 1, a 2$\mu s$ latency added to every single RPC could be amplified to around 50$\mu s$ increased end-to-end tail latency, and jeopardize the overall throughput by up to 36.9%, which are even worse under severe RLA. The reason why RLAs could severely compromise the application performance is that once the logic processes RPCs slower than NIC bandwidth (50Gbps in this experiment), lots of packets would jam the NIC buffers and be dropped, causing the network stack to retransmit massive packets and slow down, and leading to severe performance decline.

Such a performance decline would violate the SLA of cloud service, which requires an explanation in practice. However, cloud providers often face difficulties in the explanation, we list two representative real cases to show this.

**Case-1: Is the network congestion causing the RLA?** A tenant reported persistently low transmission rate between two VMs. The tenant naturally blames the network, since network congestion could slow down the sending rate. Then, network operators first retrieved switch queue and drop statistics. Data showed that the network utilization remained low and no packets were lost during that period. They have to reproduce the case using the VM's trace. However, the real cause is the limited TCP receive buffer in the host, which may be caused by CPU polling hang. It is hard for the network operators to claim their innocence unless they could detect the real cause by end-to-end monitoring.

**Case-2: Is the cause in the network or end-host?** A tenant reported an unexpected latency glitches (100s of ms) of a read request. To diagnose the RLA, the storage application owners first checked the request's trace obtained by their tracing tool and found that the interval between the request send and receive exceeded the expectation. Then, the application owners passed the ticket to the server and network team. Operators of the server team look up the *second-level* logs of kernel, CPU *etc.*, based on the timestamp in the ticket; Operators of the network team query the *flow-level* monitoring system if packet losses or network faults have occurred in the request's flow. Even if these queries have results, neither team has high confidence to claim their innocence due to the mismatch of monitoring granularity among these teams.

The above two cases indicate that cloud providers need a confident and accurate end-to-end monitoring tool to improve the efficiency of the explanation for SLA violations.

## 2.2 Limitations of Existing Monitoring Tools

From the two cases above, we can also see that the challenges of RLA diagnosis stem from the variety of the locations of root causes. Thus, accurate RLA diagnosis requires monitoring tools to have high coverage for the causes. However, existing monitoring tools have two limitations to achieve it:

*(i) Incomplete coverage.* Existing tools monitor partial datapath of requests, which either focus on application layer tracing/logging [3, 9–11, 16, 28–30], transport layer monitoring [17, 21, 24, 25, 31], network monitoring [8, 18, 19, 26, 32–35], or partial combination [34, 36–38]. In addition, existing tools define a separate set of abnormal events based on their monitoring goals. For example, Dapper [21] infers TCP performance events (*e.g.,* non-backlogged, congestion and delayed ACK) by analyzing packet statistics; Trumpet [23] leverages triggers at end-hosts to monitor network-wide events (*e.g.,* burst, heavy flow and congestion); NetSeer [8] monitors flow-level abnormal events (*e.g.,* packet drop, queuing, detouring and pause) in networks; performance profiling tools (*e.g.,* Perf [39]) can analyze events (*e.g.,* CPU cycles, page fault and cache miss) that occur during program execution; tracing tools [9–11, 14, 15] record timing data about requests and provide API to monitor application-specific annotations. Overall, there is no tool that can capture all RLA-related events in the end-to-end datapath of request so far, causing that operators have no confidence to claim their innocence (*e.g.,* Case-1).

*(ii) Inconsistent semantics.* Since these existing tools have different focuses, cloud providers have to combine multiple monitoring tools in production to cover the datapath of request as fully as possible. For example, tracing [9–11, 14, 15] is used in the application layer to track the performance of requests, and network monitoring tools [8, 18, 19] are used in the underlying network to record flow-level abnormal events. However, these monitoring tools have inconsistent semantics, the abnormal events captured by them cannot be correlated, leading to the failure of this combination (*e.g.,* Case-2). We obtain the production storage application and anonymized request traces, and run them on our testbed for 6 hours (§5.1). We use existing monitoring tools to capture abnormal events during that period and try to diagnose the cause of detected slow RPCs. Unfortunately, existing monitoring tools fail to explain a large portion of slow RPCs. First, request-level events and timing data collected by application tracing tool are too coarse-grained and incomplete, and can only explain 28% RLAs. Then, based on the time-correlation methods, flow-level events captured by network monitors can only infer 36% more RLAs, leaving 36% RLAs inexplicable.

## 2.3 Motivation

To accurately diagnose all RLAs, it is necessary to monitor the entire life cycle of all individual requests. Thus, our goal is to design a high-coverage request event monitoring sys-

tem which can monitor RLA-related events with consistent request-level semantic in the end-to-end datapath of request.

The fundamental limitation of existing performance monitoring tools to achieve our goal is that, they cannot uniformly model the data plane in network hardware and the datapath in end-host software. Unlike the software, traditional fixed-function data plane in network only provides limited accessibility for packets and black-box visibility [8].

With the development of commodity programmable hardware, which has been widely deployed in modern cloud, we see the opportunity of completely realizing our goal. We believe this choice is rational because of two unique advantages of programmable hardware. First, with the help of programmable switches and NICs, fine-grained abnormal events in networks can be easily detected, parsed and reported [8]. It makes monitoring the data plane in networks as flexible as monitoring the datapath in software. Second, SmartNICs show promising capability to offload CPU-consumption tasks [40, 41]. By leveraging them, we can achieve the consistent request-level monitoring semantics with low overhead.

## 3 Design

This section first outlines the overview of BufScope, then elaborates BufScope's design to achieve high coverage with low overhead. Finally, it shows how cloud providers can use events captured by BufScope to diagnose and mitigate RLAs.

### 3.1 Overview

The crux to make operators accurately and confidently judge where and how a request gets disturbed is to track the RLA-related events that directly happen to the request's traffic.

**Insight.** To understand the distribution of the RLA's causes and corresponding abnormal events, we have analyze almost 500 typical incident tickets of one-day's RLAs from *Alibaba*'s block storage service, which were troubleshooted by manual debugging. We present the root causes and the locations exposed anomaly in Figure 2. The root causes spread across the datapath of request, such as polling hang in hosts, incast in NICs, and burst in switches. We derive our insight that most (>90%) RLAs expose anomalies in buffers. The remaining (<10%) RLAs come from NIC flapping, network update, bugs, *etc.*, which requires hardware- or program-specific monitoring, and is hard to cover using a general solution. Besides, given that buffers are where the request's traffic stays and latency rises [26], BufScope chooses the buffer as the key object to monitor the most RLA-related events.

**Design goals.** BufScope is a request event monitoring system, which aims to achieve high coverage for RLAs' root causes. Specifically, the design of BufScope needs to achieve the following three requirements. First, BufScope should be able to monitor the request's end-to-end datapath for RLA-related events. Second, all events captured by BufScope need to have

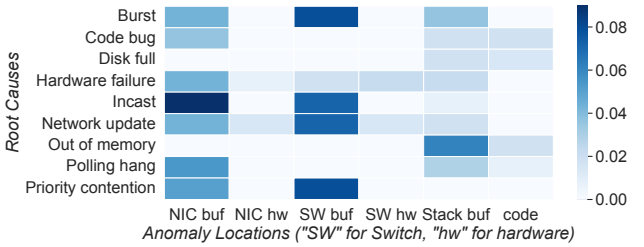



Figure 2: Heatmap for root causes and anomaly locations of RLAs.

consistent request-level semantics to correlate the events happening in the hosts and networks. Finally, in order to reduce the impact on the performance of the monitored application, BufScope must be designed with low overhead.

**Challenges.** It is highly challenging to achieve the above requirements:

- *End-to-end monitoring:* For generality, BufScope needs to model a unified buffer chain for various communication frameworks and underlying networks. However, buffers are various and have different RLA-related events. Therefore, BufScope needs to define a complete event library, which contains all events that will occur in all types of buffers.
- *Consistent request-level semantics:* The uncertainty of location of request header in packet makes it hard for the commodity programmable switches to parse out the request-level information when generating events. Thus, BufScope needs to design a novel mechanism to inject request-level semantics into the specific location of packets.
- *Low overhead:* The semantic injecting mechanism of the strawman solution consumes valuable CPU resources for the RTC application, and degrade the application performance [10]. Thus, BufScope must be designed to reduce the semantic injecting overhead as much as possible.

**Architecture.** We present BufScope's architecture in Figure 3. Following the buffer chain model (§3.2), BufScope's agents monitor buffers along the datapath of request, including applications, network stack, NICs and switches, and capture the corresponding events according to the type of buffers (§3.3). Besides, in order to record the victim request identifier when generating events in networks, BufScope enables request-level semantic injection in sender side, and offloads it into SmartNICs to reduce overhead (§3.4). For efficient event collection (§3.3), in the software and SmartNIC, the BufScope agents execute event collection asynchronously with respect to the monitored application; In the programmable switch, after events are generated by the detection logics in the ingress and egress pipeline, the egress agent hands the events to the switch control plane for further processing, such as deduplication, batching and reporting. Finally, events from these components are associated in *Event Collector* based on the request identifier, to diagnose RLAs (§3.5).

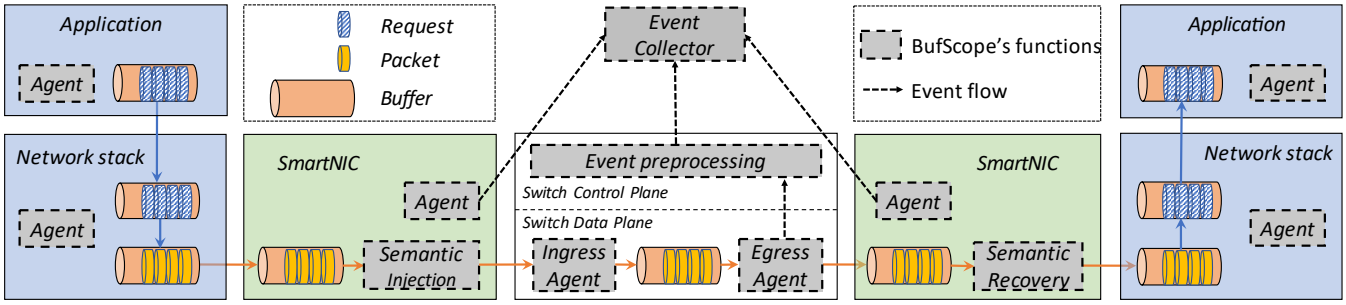



Figure 3: The architecture of BufScope.

## 3.2 Buffer Chain Modeling

Buffer in both of hosts and networks is where the request's traffic stays and is the main source of abnormal request latency rising [26], which is also proved by the production data of *Alibaba*. Thus, our key design choice is ignoring the complexity of programs, function calls and hardware faults, and closely monitor all buffers in the end-to-end datapath of requests instead, to cover the most RLA-related events.

The first step is to identify the buffers in the datapath of highly diversified Layer-7 frameworks [2, 6, 42, 43]. Existing frameworks rely on different network stacks (such as kernel-bypass network stack [44], and kernel-based network stack [43]) and different threading models (such as run-to-completion model and pipeline model). To maintain its generality, BufScope is challenged to model the datapath of various Layer-7 frameworks as a uniform composition of buffers.

We address the challenge by analyzing programs of various available Layer-7 frameworks, buffers, and their connections across the end-to-end datapath of request. We observe that one single request follows one unified chain of buffers across its entire life cycle. Therefore, we construct a *buffer chain* model as shown in the *Buffer* diagram of Figure 3.

There exist three-part buffers in the buffer chain, including host buffers, NIC buffers, and switch buffers. ① Host buffers are used to maintain messages from/to the application, as well as packets that are formed by decomposing messages (or packets that will be constructed into messages). Besides, some applications also have buffers, such as MessageQueue [45]. ② Sending side NIC buffer keeps packets delivered from the transport layer, and regularly schedules packets out into networks. Meanwhile, receiving side NIC buffer often stores packets from the network, and wait for the end-host network stack to pull packets or actively write packets into host memory. ③ Network switch buffers keep packets for switching, once the packets cannot be instantaneously forwarded, *i.e.,* a packet will be buffered or dropped in the egress queue of the port that connects the chosen next-hop.

By using different I/O techniques (*e.g.,* zero-copy), the exact number of buffers a request will experience may differ from this model. Essentially, this model provides a methodology for monitoring the end-to-end datapath of request.

## 3.3 Event Definition & Generation

Event definition and generation will determine the effectiveness and overhead of BufScope. BufScope designs them based on the principles of high coverage and low overhead.

**Buffer classification**. The buffer chain includes diverse types of buffers, which have different RLA-related abnormal events. Taking the switch buffer as an example. For lossy Ethernet, when the queuing length of a switch buffer exceeds a certain threshold, subsequent arrival packets will be dropped instead of entering the buffer, incurring a *drop* event. For lossless Ethernet, when the buffer of a downstream switch is congested, the upstream switch will pause packet forwarding until the downstream switch buffer has space for new packets, causing a *pause* event. Another example, order-sensitive buffers, such as TCP receiving buffer, may encounter head-of-line blocking (HOL), while order-insensitive buffers do not. BufScope is challenged to thoroughly analyze all types of buffers, and define the events for them respectively.

To address the challenge, we observe that though there exist various types of buffers, they could be classified according to three key properties, *i.e.,* priority awareness, order sensitivity, and enqueue feature. Priority awareness is a property for a buffer with multiple queues. If strict priority is maintained across different queues (*i.e.,* priority-aware), packets in a low priority queue will have to wait for the high priority queue to drain. Otherwise, packet dequeuing follows the FIFO principle (*i.e.,* priority-unaware). Order sensitivity refers to whether a buffer maintains strong orders of arrived packets before popping them for subsequent processing. Enqueue features are different for lossy and lossless buffers as mentioned above.

**Event definition.** According to the different type of the three properties, we define five kinds of buffer events which may cause RLAs, as shown in Table 1. We not only consider the occurrence of events, but also capture the detailed causes.

- *Priority contention.* This type of event is triggered in priority-aware buffer (*i.e.,* multi-level priority queue) when the lengths of higher-priority queues exceed a certain threshold, blocking the packets in low-priority queues for a long time. Inappropriate priority allocation may cause RLAs [46]. Conversely, FIFO buffers always first forward the packets that arrive earlier, and don't have this event.

Table 1: Buffer event definition. "•" means that the cause for this event happens right within this buffer, "←" means that the cause happens before this buffer (in a preceding buffer or program), and "→" means that the cause happens after this buffer.

| Property | Type | Event | Triggering Condition | Cause Location | Event Information |
|---|---|---|---|---|---|
| Priority awareness | priority-unaware | - | - | - | - |
| | priority-aware | *priority contention* | Queuing delay exceeds a threshold & Lengths of higher-priority queues exceed a threshold | • | - Request ID<br>- Egress queue<br>- Lengths of higher-priority queues<br>- Queuing delay |
| Order sensitivity | order-sensitive | *out-of-order* | Inconsecutive sequence number | ← | - Request ID<br>- ID of out-of-order request<br>- Queuing delay |
| | order-insensitive | - | - | - | - |
| Enqueue feature | lossy | *drop* | Queues are about to be full or already full | • | - Request ID<br>- Egress queue<br>- Egress port<br>- Ingress port |
| | lossless | *pause* | Receiving a PAUSE signal | → | - Request ID<br>- Egress queue<br>- Egress port<br>- Queuing delay |
| - | - | *congestion* | Queuing delay exceeds a threshold & no PAUSE signal | • | - Request ID<br>- Egress queue<br>- Egress port<br>- Queuing delay |

- *Out-of-order.* This type of event is triggered in order-sensitive buffers such as TCP receiving buffer. Packets have to be delivered to the applications in the same order as they are sent. That is, the packets that have been received by the order-sensitive buffers have to be delayed before receiving the packets sent earlier. In contrast, the order-insensitive buffers don't have the out-of-order event.
- *Drop.* This event happens in lossy buffers when queues are about to be full or already full. Packet drops would incur packet retransmission and may result in RLAs.
- *Pause.* This type of event happens in lossless buffer. Once the buffer occupancy of the downstream switch exceeds a specific threshold, then the downstream switch sends a PAUSE signal to the upstream switch. The latter will pause packet forwarding until a RESUME signal is received. This increases the delay of the paused packets.
- *Congestion.* This type of event could happen to any kinds of buffers. Congestion is defined as the situation where the queuing delay exceeds a threshold, and is not due to PAUSE. This could be caused by the mismatch between upstream and downstream processing or transmission rates.

Based on that, we could predict the RLA-related events that will occur in a buffer, and deploy monitoring mechanism accordingly. Note that the events are not exclusive with each other, multiple events may be captured by BufScope simultaneously, such as congestion and priority contention.

**Event generation.** Event generation, which includes the event detection and collection, could degrade the monitored application performance due to its expensive operation, *e.g.,* generating unique ID, writing disk and *etc.* [10]. Thus, they must be low overhead in BufScope. Here, we describe how they are designed in the hosts, SmartNICs and switches, respectively.

In the end-host and SmartNIC, event detection in software is straightforward. In order to reduce the impact of event collection on application performance, BufScope's agent daemon executes disk write asynchronously. Then, the agent daemon sends the event logs to the *Event Collector* in bulk.

In the programmable switch, event detection needs to be implemented entirely in the data plane. Packets that experience *pause* or *drop* were detected in the ingress pipeline and MMU, respectively. For *priority contention* and *congestion*, we record the length of queues, ingress and egress timestamps through INT (in-band network telemetry) [47], and judge whether packet's queuing delay and length exceed the thresholds. After the victim packets are detected, egress agent utilizes a bloom filters to aggregate them into request-level events with flow's 5-tuple and request ID (§3.4) as the key. Then, request events are sent to the switch control plane via data plane generated packets. Finally, the control plane will eliminate false positive in received events, and report them to the *Event Collector* in bulk. Most of the design of event generation in the data plane was proposed in NetSeer [8], and we simply changed the granularity of monitoring events from flow to request. We do not claim any novelty here.

## 3.4 Request-level Semantic Injection

The layered network architecture [48] has been a cornerstone of the Internet and is used in clouds. However, it presents a challenge for performance monitoring of distributed applications. Specifically, these teams, *e.g.,* network (Layer-3), server (Layer-4) and application (Layer-7), often blame each other for diagnosing RLAs due to the inconsistent semantics of their monitoring tools [8, 24]. To address this challenge and improve the accuracy of RLA diagnosis, BufScope needs to

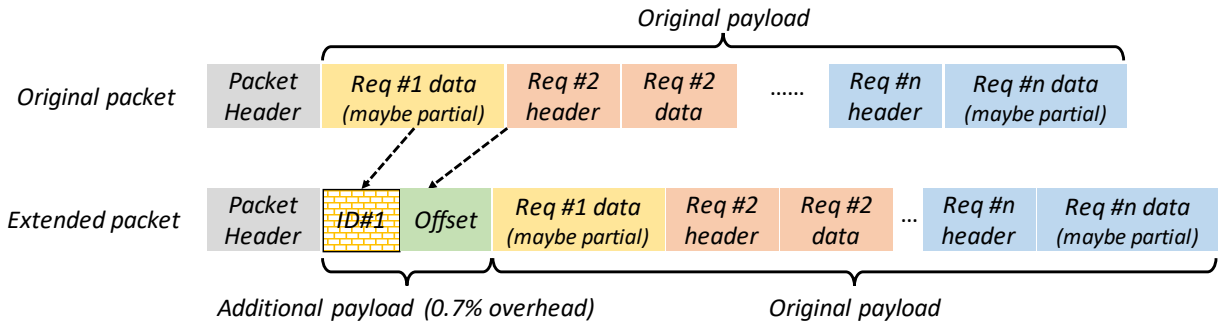



Figure 4: Request-level semantic injection in sender's NIC. (*Req* stands for request.)

map all captured events to the request-level semantics.

**Challenge of request-level semantic parsing.** A request contains two major parts, *i.e.,* header and payload. The header includes request ID, type (request/response), length, and other metadata, while payload holds the actual content of the request. The network stack is responsible for encapsulating the request into packets, with the request as the packet payload. In this process, multiple requests are considered as a byte stream and packed into packets. One large request could be carried by multiple (maybe >1,000 for storage applications) packets, while multiple (maybe >10) small requests may be consolidated in one packet. Therefore, each packet may not carry or carry multiple request headers in practice.

Realizing request-level semantic parsing in networks is non-trivial. The request header may be anywhere in the packet payload, which makes the commodity switch unable to parse the request IDs. A straightforward solution is inserting all IDs appeared in the packet before the packet payload, so that programmable switches could match and derive request IDs. However, this solution results in significant overhead. First, one request ID has 8 bytes in gRPC [43]. The standard MSS of TCP packets is 1460 bytes. Therefore, inserting 10 request IDs would introduce a 5% bandwidth overhead. This overhead is ever-present, and can lead to bandwidth degradation and packet loss under traffic bursts. Furthermore, performing lots of memory copy operations seriously wastes CPU resources.

**Concise request-level semantic injection.** To address this challenge, we leverage the fact that request *ID* and *length* are already carried in the request headers packed in packet payloads. Thus, we choose to insert the *offset* field (2 bytes) of the first complete request header at the beginning of the packet payload, as shown in Figure 4. If there are other request headers, we can use the *length* field in their headers to iteratively parse their indexes. By performing such concise operation, we explicitly maintain the request-level information in a way which programmable switches (*e.g.,* P4-16 [49]) could easily parse, while introducing very little additional overhead on performance and bandwidth.

Besides, the first data segment of the payload, *i.e.,* the partial *Req#1 data* in Figure 4, may not has the corresponding header in the current packet, because large request could be carried by multiple packets. Therefore, we should also maintain the identifier of the *Req#1 data*. To this end, we always insert the *ID* field (8 bytes) at the beginning of the packet payload, which records the *Req#1 ID*. It requires us to maintain a stateful variable, which records the ID of the recent request. The bandwidth overhead of our injection solution is only 0.7% $((8+2)/1460)$, which is fixed and negligible.

**Algorithm 1:** Semantic injection in sender's NIC

```
   Input: Packet, last_ID
1  ID#1_index ← tcp_payload_begin;
2  offset_index ← tcp_payload_begin + ID_len;
3  insert_len ← ID_len + offset_len;
4  buf_append(Packet, insert_len);
5  buf_move(ID#1_index + insert_len, ID#1_index);
6  while index++ <tcp_end do
7      if *index = Request.header then
8          if now_ID = NULL then
9              if index = ID#1_index + insert_len then
10                 ID#1_index ← NULL;
11             else
12                 *ID#1_index ← last_ID;
13             *offset_index ← index;
14         *now_ID ← *index;
15 if now_ID = NULL then
16     *ID#1_index ← last_ID;
17     offset_index ← NULL;
18 else
19     last_ID ← *now_ID;
```

**SmartNIC-offloaded semantic injection and recovery.** To reduce the CPU overhead in hosts, we offload the semantic injection to SmartNICs. We present the process of semantic injection in Algorithm 1. For each packet, we first insert a fixed space to store the *offset* and *ID field* (line 1-5). Then we look at three possible scenarios. ① These is no partial *Req#1 data* and there is a complete *Req#2 header* at the beginning of the payload (line 9-10); ② These is partial *Req#1 data* and a complete *Req#2 header* (line 11-14); ③ There are no request headers in this packet (line 15-17), then the stateful variable about the recent request ID does not need to be updated. Otherwise, in the first two scenarios, the variable

records the last request ID in the current packet (line 18-19), which may becomes the recent request ID for the next packets.

We recover the packets in the receiver's SmartNIC, which just removes the content that was inserted in the sender's SmartNIC. Through SmartNIC offloading instead of host CPU processing, semantic injection and recovery can be achieved with little impact on the application performance.

### 3.5 RLA Diagnosis and Mitigation

We introduce how cloud providers diagnose the cause of RLAs according to the events captured by BufScope. First, BufScope correlates events by request ID, and reports the beginning and ending events (possibly guilty) to the operators. Then, since the blocked time is also recorded in the events, operators can clearly see which event is the culprit, even through a request experiences multiple events. We also elaborate on how applications can benefit from these request events.

- *Priority contention.* This type of events suggests that RLAs happen directly in the current buffer. It indicates that queues with higher priorities are jammed. To mitigate this type of RLA, application owners can either upgrade the priority of its requests, or try to reduce the traffic of other applications that enter the high-priority queues.
- *Out-of-order.* This type of event suggests that packets are dropped or detoured in previous buffers or logic. For instance, network packet drop and path change could cause out-of-order in TCP receiving buffer. Application owners could ask network operators for help to debug network device failures, blackholes, or random packet drops. Also, refer to the next item if accompanied by drop events.
- *Drop.* Drop events cause time-consuming retransmission, which could directly cause RLAs. MMU drop is usually caused by burst and incast. Application owners could consider optimizing the traffic pattern through scheduling to reduce TCP incast or congestion possibilities.
- *Pause & Congestion.* These events happen due to the slow scheduling of packets out of the current buffer or the downstream buffer. In this case, BufScope could identify the request that contribute the most to the congestion, *i.e.,* the heavy request, because the heavy request will experiences more congestion events. Then, cloud providers need to evaluate the network architecture and application mixing model.

## 4 Implementation

We have implemented BufScope for a kernel-based RPC framework named Finagle [50] and the kernel-bypass-based *Alibaba*'s block storage application. We use Barefoot Tofino switches and Broadcom PS225 SmartNICs to implement the functions of BufScope in the data plane.

**Requirements.** Because BufScope needs to insert the ID and offset field of the request at the end of the packet header, implementation of BufScope requires application-layer protocol awareness and MTU modification. And kernel-intrusive is needed for monitoring kernel-based application. Finally, BufScope is designed to monitor requests inside the cloud (*i.e.,* east-west traffic) that are typically not encrypted.

**Incremental deployment.** For end-to-end monitoring, multiple teams (*e.g.,* server and network) in the cloud need to monitor the buffers they manage by using BufScope's APIs. However, partial deployment of BufScope still facilitates RLAs diagnosis. With sole support from the server team, semantic injection and in-server event monitoring can still be performed, which helps operators decide whether the root cause locates in the server or not. With sole support from the network team, operators can blame or exonerate the network according to packet- or flow-level events in the network.

**Buffer identification.** BufScope summarizes a basic buffer chain for various applications and network stacks. For kernel-based, there is an application buffer and a socket buffer. For kernel-bypass-based, the zero-copy technology makes the applications may have only one *mbuf* array for DPDK [51]. The manual efforts to identify buffers are small and only need to be done once. Moreover, resource contention in other hardware or OS queues (*e.g.,* CPU, DRAM, and PCIe) will cause slow message processing, resulting in queue buildup in the upstream buffer [52], which can be detected by BufScope.

**Event capturing.** We record the following necessary information for each type of events.

- *Priority contention* (15B): <*ID, egress queue, length of higher-priority queues, queuing delay*>. We measure the queuing delay inside a switch with ingress and egress timestamps. The victim request ID, the timestamps and the length of the higher-priority queue is obtained by INT.
- *Out-of-order* (20B): <*ID, ID of out-of-order request, queuing delay*>. We identify out-of-order requests by observing inconsecutive sequence number in packets, and generate this type of events for latter blocked requests.
- *Drop* (11B): <*ID, egress queue, egress port, ingress port*>. In network, we redirect packets dropped by MMU to a dedicated internal port, and report in egress pipeline [8], then parse the request ID in these packets.
- *Pause* (14B): <*ID, egress queue, egress port, queuing delay*>. For a lossless network, the switch begins to generate pause events immediately after receiving the pause signal.
- *Congestion* (14B): <*ID, egress queue, egress port, queuing delay*>. Congestion events are produced when the queuing delay exceeds a threshold while the length of the higher-priority queue is normal.

To reduce the bandwidth overhead, we leverage lossless *ZigZag Encoding* [53] to compress events. The average length of events is shortened from 15 bytes to 8 bytes. Besides, BufScope allows setting an upper limit on the event generation rate. Once this threshold is exceeded, sampling can be enabled.

**SmartNIC.** We implement the NIC buffer monitoring and semantic injection in the ARM-based SmartNIC. In addition to the cores required for packet forwarding, BufScope

requires only one additional core for event collection and reporting. Note that the ID of the partial *Req#1 data* in retransmitted packets has been missed in NIC. We just set the inserted *ID field* as *NULL* in the retransmitted packets. The effects of this simplification are limited, because the causes of packet retransmission have already been captured (*e.g.,* drop or out-of-order). When enabling TSO, semantic injection is performed after packets are segmented in SmartNIC.

**Switch.** BufScope's ASIC logic can be embedded into original switch programs (*switch.p4* in our experiment) as an extension. We layout the timestamp record and pause detection modules in ingress pipelines, enable drop detection in the MMU, and detect congestion, priority contention in the egress pipeline. After the event is generated by the switch ASIC, event pre-processing and reporting in the switch CPU is similar to that in the NetSeer system [8].

**Event collector.** To timely receive events, we use the servers with 100Gbps NICs in the cluster as the *Event Collector*. To improve the readability and usability of monitored data, Collector aggregate the events in two stages. First, the events captured by all components are aggregated together at per-request granularity. Then, if there is a trace data generated by the tracing tool for the request, the request events timestamp and the associated information are marked in that trace.

## 5 Evaluation

**Environment:** We evaluate BufScope and existing monitoring tools on a testbed with a 4-ary and 3-tier Fat-Tree topology [54] composed of 10 Barefoot Tofino switches and 16 servers. Each server has 192 CPU cores, 64GB RAM, and one Broadcom PS225 SmartNICs (2×25G) [55]. Each SmartNIC possesses eight ARM Cortex-A72 3.0 GHz CPUs and 16GB memory. There are 4 ToR, 4 Aggregate and 2 Core switches. They are interconnected with 100G links, while each ToR connects four servers with 2×25G link.

**Baselines:** Given that none of the existing monitoring tools are designed to monitor the end-to-end datapath of request, we combine multiple state-of-the-art tools to fully cover it:

*(i) Application tracing.* We enable an open-source tracing tool [15] to capture the RPC timing data and application-specific abnormal events in application layer. Because of the large amount of captured data and non-negligible CPU overhead, tracing tools typically require sampling (*e.g.,* 0.1% under high-load services [10]) to reduce impact on the performance of the monitored application. Thus, we set the sampling rate of tracing as 0.1% and 100% for comparison.

*(ii) TCP monitoring.* Dapper [21] is used to diagnose performance problem of TCP in the end-host network stack.

*(iii) Network monitoring.* We deploy NetSeer [8] and packet sampling to capture events in networks and NICs. NetSeer is a flow-level event monitoring system based on programmable data plane. For packet sampling, we can parse the request ID in the mirrored packet offline to get request events. Since it has a large bandwidth overhead under a high sampling rate [8], we configure the sampling rate as 1%.

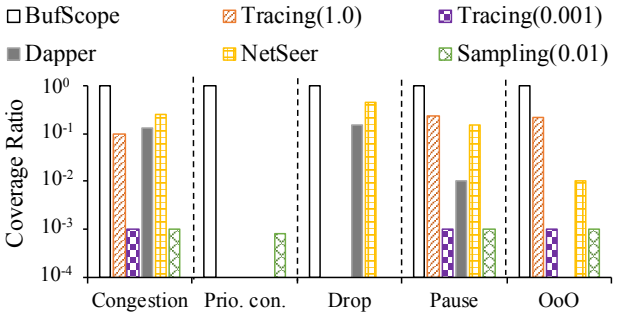


Figure 5: Event coverage ratios. Tracing(sampling rate).

The evaluation needs to answer the following 3 questions.

- **Coverage:** Can BufScope capture most (close to 100%) request events that happen in hosts, NICs, and network switches, and help accurately diagnose the real RLAs?
- **Scalability:** What's the bandwidth overhead of BufScope to deliver events to the *Event Collector*? Can it scale with the increasing datacenter size and bandwidth?
- **Performance overhead:** How to choose an efficient threshold? How does BufScope affect the application performance? How about the impact of each module, including request-level semantic injection by host CPU or SmartNIC?

### 5.1 Coverage

We deploy the storage application (supports RDMA) and Finagle as the monitored applications, and run traffic traces based on four real-world workloads including DCTCP [56], VL2 [57], storage and WEB [58] for 6 hours. We set the average link utilization as 80% to test BufScope's coverage under extreme conditions. *Congestion*, *drop* and *out-of-order* (OoO) are naturally produced. We configure various priority queues to trigger *priority contention*, and enable priority flow control (PFC [59]) in RDMA network to trigger *pause*, which do occur in production environments [8, 37]. We start by evaluating BufScope's capability to fully capture all events along the datapath of request. Next, we compare the proportion of unexplained RLAs of different monitoring tools, and study 2 real RLAs which cause the SLA violations.

**Event coverage.** We enable tracing, Dapper, NetSeer, packet sampling and BufScope to capture events, respectively. We present the event coverage ratios for different types of events in Figure 5. For a fair comparison, we enable tracing tool in this experiment to monitor all buffer events in host buffers that it can cover. Even so, the tracing tool only has visibility into applications, it cannot detect events in the network. Therefore, tracing(1.0) can only cover up to 23% events, while tracing(0.001) can only cover 0.1% events. Dapper analyzes TCP statistics to infer the occurrence of network events, such as congestion and drop, it can only cover up to 15% events.

NetSeer could capture flow-level events in networks, including congestion, pause and drop, but leaves out the priority

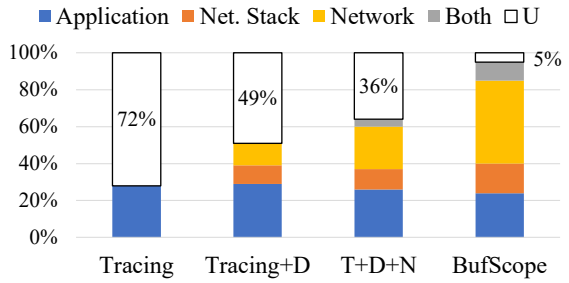


Figure 6: Diagnosing RLAs by different monitoring tools (T:Tracing, D:Dapper, N:NetSeer, U:Unknown).

contention and OoO events. Besides, its captured events miss request-level semantics. Based on the time-correlation methods, NetSeer can only cover up to 45% request events. For packet sampling, if the mirrored packet is lucky enough to encounter an event, then we can parse out the event with request ID. Thus, it only cover <10% events which is always less than its sampling rate. In comparison, BufScope has full coverage for the 5 types of request events happened in both of the end-hosts and networks.

**Diagnosing RLAs.** We try to diagnose the root cause of the slow RPC (*i.e.,*RLAs) detected during that period according to different monitoring tools. Request-level timing data collected by tracing tool with full sampling can only explain 28% RLAs, leaving 72% RLAs undetermined, as shown in Figure 6. Then, server and network monitors capture flow-level events to diagnose RLAs based on the time-correlation methods, can only explain 23% and 13% more RLAs, respectively. Even so, enabling tracing, Dapper and NetSeer (*i.e.,* T+D+N) at the same time still leaves 36% RLAs inexplicable. With BufScope's help, we can tell whether and how much each component is responsible for each slow RPC, and explain much more (95%) RLAs, including those whose causes were unknown with existing monitors, and some RLAs that were caused by multiple components. The remaining 5% of the RLAs did not reveal any events. We speculate that they are caused by hardware-related anomalies.

We reproduce 2 real *Alibaba*'s production RLAs on our testbed with inferred topology, requests pattern, and traffic rate during the incidents, which cannot be captured and explained by existing monitoring tools.

*#1) Polling hang in the host.* When a request encounters more than one anomaly, the challenge in diagnosing is to identify the one that has the greatest impact. For example, one RPC in this experiment encountered congestion in the network and polling hang in the receiver. However, existing monitoring tools cannot capture how much delay each anomaly causes, and treat them equally, resulting in the inefficient diagnostic process. Conversely, BufScope can end-to-end capture latency-critical events with consistent semantics, which can associate events that occur in different components. BufScope found that it was blocked for 5*ms* at the receiver's NIC, and only experienced 80$\mu$s of congestion in the network. Therefore, the reason for the RLA was polling hang. Based on this,

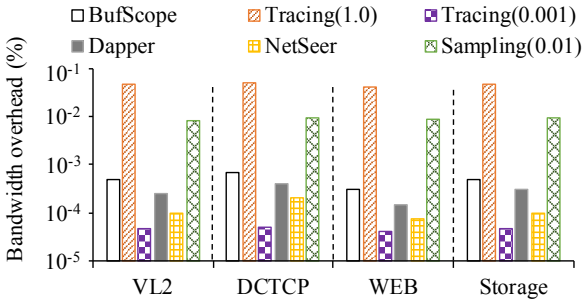


Figure 7: Bandwidth overhead of event collection.

application owners can further analyze the abnormal events in the host and the system log to solve the problem.

*#2) Cascaded priority contention.* In a priority-aware network, it is non-trivial to assign priorities to different applications. An inappropriate allocation can result in SLA miss for low-priority application. Thus, checking the priority contention in the network is an important task of performance monitoring tools. Worse still, a cascading effect would happen when there are multiple queues with diverse priorities. Consider there are three requests, *Req#1*, *Req#2*, and *Req#3* have decreasing order of priority. *Req#1* and *Req#2* contend in an upstream switch $S_1$, while *Req#2* and *Req#3* contend in a downstream switch $S_2$. If *Req#1* in $S_1$ is congested, *Req#2* would be delayed, which then delays *Req#3* in the low-priority queue of $S_2$. To debug the RLAs of *Req#3*, simply observing the priority contention in a switch is not enough. Since BufScope can capture all contention events and their details, we can analyze this cascade effect and find a more effective method to mitigate it.

## 5.2 Scalability

We compared the bandwidth overhead (BO) required by BufScope and baselines to report events or traces during that period. Figure 7 shows that BufScope only incurs <0.07% BO under various real-world workloads, of which 0.02% from host, 0.01% from NICs and 0.04% from switches. For link bandwidth at 100Gbps, the overhead is at most 70Mbps, which is within the capacity of PCIe (18Gbps) and switch CPU (13.4Gbps with 2 cores). In comparison, tracing(1.0) suffers from >4% BO, its each span (*i.e.,* every request) needs 400B on average. Tracing(0.001) needs >0.004% BO. Dapper records only TCP abnormal events and consumes only 0.04% BO. Network packet sampling (0.01) needs $\sim$ 1% BO, which is similar to its sampling rate, because the payload also needs to be recorded to parse request semantics. Because NetSeer captures and reports flow-level events, its event scale and fineness are not as high as BufScope. Thus, NetSeer only incurs $\sim$ 0.01% BO. In summary, T+D+N consumes >4.05% BO.

To further understand the scalability of BufScope, we calculate the monitoring event traffic as well as the processing overhead of BufScope according to the configuration of *Alibaba*'s production datacenters. For a normal 3-tier datacenter, connecting 10,000 servers requires approxi-

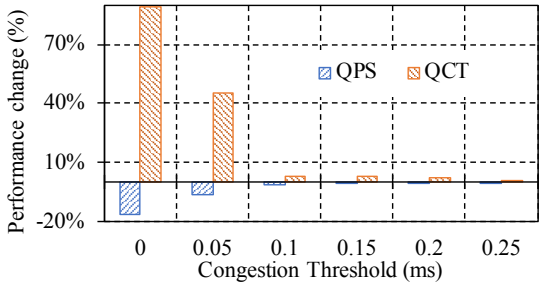



Figure 8: Impact of congestion threshold on performance.

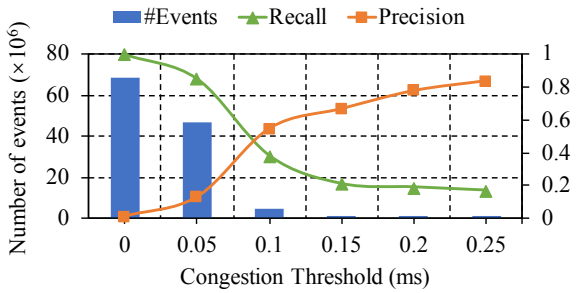



Figure 9: Number of captured congestion events and the two indicators under different thresholds.

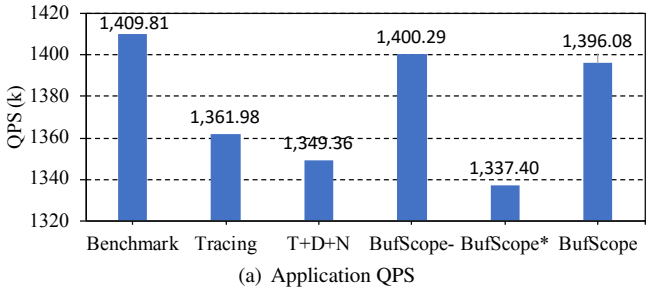
mately 400 switches (3.2Tbps), which produce a maximum of $400 \times 3200Gbps \times 0.07\% = 896Gbps$ monitoring traffic at most. Processing such traffic requires 9 servers with 100Gbps NICs, which implies a 0.09% processing overhead.

## 5.3 Performance Overhead

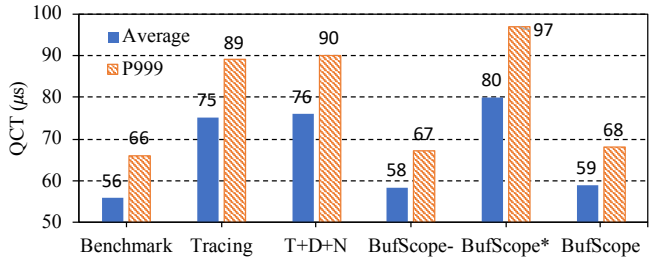
In this experiment, we use as many threads as possible, which perform 4KB file read, to test the extreme performance of the *Alibaba*'s storage application under different monitoring tools. We first show a method for selecting the appropriate congestion event threshold. Then, we evaluate the performance overhead of the per-module and overall BufScope.

**Congestion event threshold determination.** Congestion events appear when the queuing delay exceeds a certain threshold, which we define as the congestion threshold. We pay special attention to congestion events as it occupies a major portion of all events, which will occur in both the hosts and networks. BufScope can harm application performance if too many congestion events are collected. Thus, we run the application for 10 seconds with a full-mesh traffic pattern, and measure the impact on the QPS (Query per Second) and QCT (Query Completion Time) of the application as we vary the congestion threshold. As shown in Figure 8, the larger the threshold, the smaller the performance overhead, because fewer congestion events would be collected. Thus, threshold selection is highly related to the efficiency of BufScope.

Since requests without RLA can also experience light congestion, this means that not all captured congestion events are RLA-related. In this experiment, the captured events are RLA-related when the RLA request experiences only congestion events and no other events. We use two indicators to




(a) Application QPS




(b) Application QCT

Figure 10: Application performance under different tools.

evaluate the efficiency of the congestion threshold. `Recall` represents the proportion of the captured RLA-related congestion events in all real RLA-related congestion events, while `precision` represents the proportion of the captured RLA-related congestion events in all captured congestion events. The higher the threshold, the lower the `recall` and the higher the `precision`. Thus, the selection of the threshold needs to balance these two indicators. Figure 9 shows the changes in the number of events collected, the `recall` and `precision`. We observe that as the threshold increases from 0 to 0.25ms, the `precision` increases from a very low value to nearly 100%, and `recall` drops from 100% to a very low value. In the following experiments, we take 0.1ms as the congestion threshold for high monitoring efficiency.

**Event monitoring in hosts.** Because monitoring in hosts uses expensive CPU resources, we evaluate the performance overhead of only enabling functions of BufScope in hosts (we refer to this variation as BufScope-). As shown in Figure 10, we generate the highest load (*i.e.,* extreme throughput of NIC) with 8 threads to test the application, and obtain the QPS, average and P999 QCT by running 30 seconds. The Benchmark represents the raw performance of the application without any monitoring tools. BufScope- decreases the QPS by 0.7% and increases the P999 QCT by 1.5%. Because BufScope records events asynchronously, most of this overhead comes from event detection, which takes tens of nanoseconds on average. In contrast, the tracing tool generates a trace for each sampled request, which takes sub-microseconds. Thus, only enabling tracing(1.0) in hosts decreases the QPS by 3.4% and increases the P999 QCT by 34.8% under the same load. This demonstrates that BufScope's event-driven approach significantly reduces the performance overhead.

**Semantic injection in network stack.** Next, we enable all monitoring functions of BufScope in hosts, SmartNICs and

switches. In this experiment, we evaluate the impact on the RTC application by implementing the BufScope's semantic injection in the application's network stack, namely BufScope*. As shown in Figure 10, since the semantic injection uses the same thread with the application processing, BufScope* decreases the QPS by 5.1% and increases the P999 QCT by 47.0%. In comparison, the combination of tracing, Dapper and NetSeer, *i.e.,* T+D+N, decreases the QPS by 4.3% and increases the P999 QCT by 36.4%. BufScope*'s performance overhead is large than the combination. The results reveal that the overhead of using the same CPU to perform semantic injection is not negligible, and we need to use offload techniques to reduce the overhead.

**Overall performance overhead of BufScope.** According to BufScope's design, here semantic injection is implemented in the sender's SmartNIC. BufScope only decreases the QPS by 1.0% and increases the P999 QCT by 3.0%. This demonstrates that *SmartNIC-offloaded semantic injection and recovery* can significantly reduce the performance overhead. Compared with T+D+N, BufScope improves the QPS by 3.5% and reduces the P999 QCT by 24.4%. Besides, the performance of BufScope is slightly lower than that of BufScope-. Such performance decline is introduced by our ARM-based implementation in SmartNIC. We will use FPGA-based SmartNIC in the future to further improve processing performance.

## 6 Related Work

There has been a rich literature regarding application monitoring and diagnosis. We classify them into six categories according to their coverage for the request's datapath.

**Tracing-based.** Tracing-based monitoring tools are widely used for large-scale application performance tracing and debugging [9–14,16,28,30,60,61]. By inserting annotations into the execution path of the request, tracing tools can locate the problematic step in application layer, but has no visibility in the network stack and underlying networks. Besides, tracing could provide fine-grained latency statistics, but will actually degrade application performance [10]. Therefore, tracing are often used in an on-demand and sampling way.

**Log-based.** Log analysis is proven effective in many programs or performance debugging scenarios [3, 29, 62–67]. However, logs are often created by CPU, which is proven inefficient and could waste much CPU resources. Therefore, log-based monitoring systems often use second-level monitoring granularity, which will miss a lot of RLAs.

**Network stack-based.** Many researches are trying to monitor network performance on the end-host network stack. For example, some research efforts propose to constantly monitor TCP performance by watching TCP statistics such as timeout and retransmission, and deduce the root cause of RLAs through statistical analytics [17, 21, 31, 68], replay [25], or machine learning [24]. Trumpet [23] leverages triggers at end-hosts to monitor every packet and network-wide events. However, they lack visibility into the network, leading to the incomplete coverage for RLAs. Moreover, they focus on packet- or flow-level event capturing and analysis, and cannot correlate events to the corresponding requests.

**NIC-based.** Simon [35] collects statistics from NICs, and reconstruct flow queuing time, link utilization, link composition, and other statistics. Nevertheless, it could only obtain aggregated statistics with millisecond-level granularity and lose clues for events at fine-timescale such as microsecond-level microbursts. Similarly, it mainly focuses on network events and cannot fully detect host events.

**Network-based.** The network serves as the conjunction component among distributed servers. Thus, many efforts have been devoted to network monitoring by active probing [19, 22], telemetry [18, 32], *etc.* NetSeer [8] leverages programmable switch to monitor flow-level network abnormal events, without the request-level semantics. Retro [33] and Microscope [26] monitor the queue to identify anomalies, which is similar to BufScope's buffer model. However, network-based monitoring tools have no visibility into hosts. Moreover, their combination with tracing cannot improve the accuracy of RLAs diagnosis due to the inconsistent semantics.

**Network and host collaboration.** Recent researches use both the network and end-host to jointly collect, store and analyze data [34, 36–38]. In order to correlate packets' behaviour in end-hosts and networks, they often enable network switches to attach metadata to packets, and extract event in hosts, which will consume a lot of host CPU resources. Besides, these systems did not consider the request-level abnormal events and RLA diagnosis.

## 7 Conclusion

This paper presents a promising way to utilize the programmable data plane to achieve high coverage for request monitoring and accurate RLA diagnosis, by proposing BufScope. Its core idea is to uniformly model the data plane in networks and the datapath in hosts using buffer. It translate most RLAs to buffer-related events, and monitor them in the buffer chain with consistent request-level semantic. Testbed-based evaluations show that BufScope can diagnose 95% RLAs with negligible bandwidth and performance overhead.

## Acknowledgement

We thank our shepherd Dr. Ying Zhang, and the anonymous reviewers for their constructive comments. Dan Li is the corresponding author. This work is supported by the National Key R&D Program of China (2018YFB1800100), Alibaba Innovative Research (AIR) Program, Tsinghua University-China Mobile Communications Group Co.,Ltd. Joint Institute, and the National Natural Science Foundation of China (U21B2022).

## References


[1] CNCF. Cloud native computing foundation: https://cncf.io/, 2021.

[2] Patrick Stuedi, Animesh Trivedi, Bernard Metzler, and Jonas Pfefferle. Darpc: Data center rpc. In *ACM SoCC*, 2014.

[3] Xu Zhao, Yongle Zhang, David Lion, Muhammad Faizan Ullah, Yu Luo, Ding Yuan, and Michael Stumm. lprof: A non-intrusive request flow profiler for distributed systems. In *USENIX OSDI*, 2014.

[4] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D Davis, Sriram Rao, Tao Zou, and Aviad Zuck. Tango: Distributed data structures over a shared log. In *ACM SOSP*, 2013.

[5] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, et al. The ramcloud storage system. In *ACM TOCS*, 2015.

[6] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter rpcs can be general and fast. In *USENIX NSDI*, 2019.

[7] Yixiao Gao, Qiang Li, et al. When cloud storage meets RDMA. In *USENIX NSDI*, 2021.

[8] Yu Zhou, Chen Sun, Hongqiang Harry Liu, Rui Miao, Shi Bai, Bo Li, Zhilong Zheng, Lingjun Zhu, Zhen Shen, Yongqing Xi, et al. Flow event telemetry on programmable data plane. In *ACM SIGCOMM*, 2020.

[9] Rodrigo Fonseca, George Porter, Randy H Katz, and Scott Shenker. X-trace: A pervasive network tracing framework. In *USENIX NSDI*, 2007.

[10] Benjamin H Sigelman, Luiz Andre Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. 2010.

[11] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, et al. Canopy: An end-to-end performance tracing and analysis system. In *ACM SOSP*, 2017.

[12] Arjun Satish, Thomas Shiou, Chuck Zhang, Khaled Elmeleegy, and Willy Zwaenepoel. Scrub: online troubleshooting for large mission-critical applications. In *ACM EuroSys*, 2018.

[13] Dan Ardelean, Amer Diwan, and Chandra Erdman. Performance analysis of cloud applications. In *USENIX NSDI*, 2018.

[14] Uber Technologies. Jaeger: open source, end-to-end distributed tracing. https://www.jaegertracing.io/, 2020.

[15] Twitter. Zipkin. http://zipkin.io/, 2021.

[16] CNCF. Opentelemetry. http://opentelemetry.io/, 2021.

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

[18] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y Zhao, et al. Packet-level telemetry in large datacenter networks. In *ACM SIGCOMM*, 2015.

[19] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, et al. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *ACM SIGCOMM*, 2015.

[20] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. Lossradar: Fast detection of lost packets in data center networks. In *ACM CoNEXT*, 2016.

[21] Mojgan Ghasemi, Theophilus Benson, and Jennifer Rexford. Dapper: Data plane performance diagnosis of tcp. In *ACM SOSR*, 2017.

[22] Yanghua Peng, Ji Yang, Chuan Wu, Chuanxiong Guo, Chengchen Hu, and Zongpeng Li. detector: a topology-aware monitoring system for data center networks. In *USENIX ATC*, 2017.

[23] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. Trumpet: Timely and precise triggers in data centers. In *ACM SIGCOMM*, 2016.

[24] Behnaz Arzani, Selim Ciraci, Boon Thau Loo, Assaf Schuster, and Geoff Outhred. Taking the blame game out of data centers operations with netpoirot. In *ACM SIGCOMM*, 2016.

[25] Yuliang Li, Rui Miao, Mohammad Alizadeh, and Minlan Yu. Deter: Deterministic {TCP} replay for performance diagnosis. In *USENIX NSDI*, 2019.

[26] Junzhi Gong, Yuliang Li, Bilal Anwer, Aman Shaikh, and Minlan Yu. Microscope: Queue-based performance diagnosis for network functions. In *ACM SIGCOMM*, 2020.

[27] Jaehyun Hwang, Qizhe Cai, Ao Tang, and Rachit Agarwal. Tcp = rdma: Cpu-efficient remote storage access with i10. In *USENIX NSDI*, 2020.

[28] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using magpie for request extraction and workload modelling. In *USENIX OSDI*, 2004.

[29] Yongle Zhang, Serguei Makarov, Xiang Ren, David Lion, and Ding Yuan. Pensieve: Non-intrusive failure reproduction for distributed systems using the event chaining approach. In *ACM SOSP*, 2017.

[30] Yongle Zhang, Kirk Rodrigues, Yu Luo, Michael Stumm, and Ding Yuan. The inflection point hypothesis: a principled debugging approach for locating the root cause of a failure. In *ACM SOSP*, 2019.

[31] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, and Alex C Snoeren. Passive realtime datacenter fault detection and localization. In *USENIX NSDI*, 2017.

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

[33] Jonathan Mace, Peter Bodik, Rodrigo Fonseca, and Madanlal Musuvathi. Retro: Targeted resource management in multi-tenant distributed systems. In *USENIX NSDI*, 2015.

[34] Anurag Khandelwal, Rachit Agarwal, and Ion Stoica. Confluo: Distributed monitoring and diagnosis stack for high-speed networks. In *USENIX NSDI*, 2019.

[35] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. Simon: A simple and scalable method for sensing, inference and measurement in data center networks. In *USENIX NSDI*, 2019.

[36] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. Simplifying datacenter network debugging with pathdump. In *USENIX OSDI*, 2016.

[37] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. Distributed network monitoring and debugging with switchpointer. In *USENIX NSDI*, 2018.

[38] Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. Gray failure: The achilles' heel of cloud-scale systems. In *USENIX HotOS*, 2017.

[39] Arnaldo Carvalho De Melo. The new linux'perf'tools. In *Slides from Linux Kongress*, volume 18, 2010.

[40] YoungGyoun Moon, SeungEon Lee, Muhammad Asim Jamshed, and KyoungSoo Park. Acceltcp: Accelerating network applications with stateful {TCP} offloading. In *USENIX NSDI*, 2020.

[41] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. Enabling programmable transport protocols in high-speed nics. In *USENIX NSDI*, 2020.

[42] Apache. Thrift. http://thrift.apache.org/, 2020.

[43] Google. grpc: A high-performance, open source universal rpc framework. https://grpc.io/, 2020.

[44] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mtcp: a highly scalable user-level {TCP} stack for multicore systems. In *USENIX NSDI*, 2014.

[45] Apache. Rocketmq. https://rocketmq.apache.org/, 2021.

[46] Xiaoqi Chen, Shir Landau Feibish, Yaron Koral, Jennifer Rexford, Ori Rottenstreich, Steven A Monetti, and Tzuu-Yi Wang. Fine-grained queue measurement in the data plane. In *CoNEXT*. ACM, 2019.

[47] Changhoon Kim, Anirudh Sivaraman, Naga Katta, Antonin Bas, Advait Dixit, and Lawrence J Wobker. In-band network telemetry via programmable dataplanes. In *ACM SIGCOMM*, 2015.

[48] I Standardization. Iso/iec 7498-1: 1994 information technology–open systems interconnection–basic reference model: The basic model. *International Standard ISOIEC*, 74981:59, 1996.

[49] Mihai Budiu and Chris Dodd. The p416 programming language. *ACM SOSP*, 2017.

[50] Twitter. Finagle. http://twitter.github.io/finagle/, 2021.

[51] DPDK Intel. Data plane development kit. http://dpdk.org, 2014.

[52] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan MG Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, et al. Swift: Delay is simple and effective for congestion control in the datacenter. In *ACM SIGCOMM*, 2020.

[53] Google. Protocol buffers: Encoding: Signed integers. https://developers.google.com/protocol-buffers/docs/encoding#signed_integers, 2021.

[54] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. *SIGCOMM computer communication review*, 2008.

[55] Broadcom. Stingray ps225 smartnic. https://www.broadcom.com/products/ethernet-connectivity/smartnic/ps225, 2020.

[56] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *ACM SIGCOMM*, 2010.

[57] Albert Greenberg, James R Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A Maltz, Parveen Patel, and Sudipta Sengupta. Vl2: a scalable and flexible data center network. In *ACM SIGCOMM*, 2009.

[58] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C Snoeren. Inside the social network's (datacenter) network. In *SIGCOMM*, 2015.

[59] Ieee standard for local and metropolitan area networks–media access control (mac) bridges and virtual bridged local area networks–amendment 17: Priority-based flow control. *IEEE Std 802.1Qbb-2011 (Amendment to IEEE Std 802.1Q-2011 as amended by IEEE Std 802.1Qbe-2011 and IEEE Std 802.1Qbc-2011)*, pages 1–40, 2011.

[60] Úlfar Erlingsson, Marcus Peinado, Simon Peter, Mihai Budiu, and Gloria Mainar-Ruiz. Fay: extensible distributed tracing from kernels to clusters. In *ACM TOCS*, 2012.

[61] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. Pivot tracing: Dynamic causal monitoring for distributed systems. In *ACM TOCS*, 2018.

[62] Xu Zhao, Kirk Rodrigues, Yu Luo, Ding Yuan, and Michael Stumm. Non-intrusive performance profiling for entire software stacks based on the flow reconstruction principle. In *USENIX OSDI*, 2016.

[63] Liang Luo, Suman Nath, Lenin Ravindranath Sivalingam, Madan Musuvathi, and Luis Ceze. Troubleshooting transiently-recurring errors in production systems with blame-proportional logging. In *USENIX ATC*, 2018.

[64] Ding Yuan, Soyeon Park, Peng Huang, Yang Liu, Michael M Lee, Xiaoming Tang, Yuanyuan Zhou, and Stefan Savage. Be conservative: enhancing failure diagnosis with proactive logging. In *USENIX OSDI*, 2012.

[65] Mike Y Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *IEEE DSN*, 2002.

[66] Karthik Nagaraj, Charles Killian, and Jennifer Neville. Structured comparative analysis of systems logs to diagnose performance problems. In *USENIX NSDI*, 2012.

[67] Srikanth Kandula, Ratul Mahajan, Patrick Verkaik, Sharad Agarwal, Jitendra Padhye, and Paramvir Bahl. Detailed diagnosis in enterprise networks. In *ACM SIGCOMM*, 2009.

[68] Behnaz Arzani, Selim Ciraci, Luiz Chamon, Yibo Zhu, Hongqiang Harry Liu, Jitu Padhye, Boon Thau Loo, and Geoff Outhred. 007: Democratically finding the cause of packet drops. In *USENIX NSDI*, 2018.