Zero Overhead Monitoring for Cloud-native Infrastructure using RDMA

Zhe Wang, *Shanghai Jiao Tong University;* Teng Ma, *Alibaba Group;* Linghe Kong, *Shanghai Jiao Tong University;* Zhenzao Wen, Jingxuan Li, Zhuo Song, Yang Lu, Yong Yang, and Tao Ma, *Alibaba Group;* Guihai Chen, *Shanghai Jiao Tong University;* Wei Cao, *Alibaba Group*

https://www.usenix.org/conference/atc22/presentation/wang-zhe

This paper is included in the Proceedings of the 2022 USENIX Annual Technical Conference.

July 11–13, 2022 • Carlsbad, CA, USA

978-1-939133-29-8

Open access to the Proceedings of the 2022 USENIX Annual Technical Conference is sponsored by

Zero Overhead Monitoring for Cloud-native Infrastructure using RDMA

Zhe Wang[1], Teng Ma[2], Linghe Kong[1], Zhenzao Wen[2], Jingxuan Li[2], Zhuo Song[2], Yang Lu[2]
Yong Yang[2], Tao Ma[2], Guihai Chen[1], Wei Cao[2]
[1]*Shanghai Jiao Tong University*
[2]*Alibaba Group*

Abstract

Cloud services have recently undergone a major shift from monolithic designs to microservices running on the cloud-native infrastructure, where monitoring systems are widely deployed to ensure the service level agreement (SLA). Nevertheless, the traditional monitoring system no longer fulfills the demands of cloud-native monitoring, which is observed from the practical experience in Alibaba cloud. Specifically, the monitor occupies resources (*e.g.*, CPU) of the monitored infrastructure, disturbing services running on it. For example, enabling monitor causes jitters/declines of online services in Alibaba's "double eleven" shopping festival with high loads. On the other hand, the quality of service (QoS) of monitoring itself, which is vital to track and ensure SLA, is not guaranteed with the high loaded system.

In this paper, we design and implement a novel monitoring system, named ZERO, for cloud-native monitoring. First, ZERO achieves zero overhead to collect raw metrics from the monitored hosts using *one-sided* remote direct memory access (RDMA) operations, thus avoiding any interferences to cloud services. Second, ZERO adopts receiver-driven model to collect monitoring metrics with high QoS, where credit-based flow control and hybrid I/O model are proposed to mitigate network congestion/interference and CPU bottlenecks. ZERO has been deployed and evaluated in Alibaba cloud. Deployment results show that ZERO achieves no CPU occupation at the monitored host and supports $1 \sim 10k$ hosts with $0.1 \sim 1s$ sampling interval using single thread for network I/O.

1 Introduction

Recent shifts in the production cloud environment from monolithic designs to microservice-based architecture [33,34] have made cloud-native infrastructure the cornerstone of cloud computing services. The cloud-native applications consist of thousands of single-concern, loosely-coupled microservices running on containerized platforms [76]. The underlying systems are treated as disposable and immutable, finally enabling highly available, flexible and scalable cloud services.

In order to ensure the service level agreement (SLA) [52], the whole infrastructure is monitored with not only the upper-layer application metrics, but also the fundamental system metrics [84]. The novel cloud-native infrastructure, however, brings new challenges/demands to cloud-native monitoring, along with two major issues in commercial deployments.

First, traditional monitoring systems [10,11,65] occupies host (physical/virtual machine, PM/VM) resources to collect, process and upload metrics (Figure 1), which inevitably causes resource contentions with cloud services — enabling monitors causes jitters/declines of online services in Alibaba "double eleven" shopping festival (Figure 3). To ensure service SLA with resource constraints, the deployed monitor at the host should have no resource occupation.

Second, the quality of service (QoS) of monitoring itself is not guaranteed, which fails to support massive metrics with rapid variations in cloud-native monitoring. The latency/throughput of monitoring jitters severely due to the high system loads or small CPU quota set by the cloud provider (Figure 4). However, monitoring system with high QoS is vital to track and ensure SLA of monitored services [80,84].

To resolve the limitations of traditional monitoring system and fulfill the demands of cloud-native monitoring, we design and implement a novel ZERO monitoring system in this paper. ZERO proposes a receiver-driven model, which collects raw metrics from the monitored host via *one-sided* RDMA operations, *i.e.*, RDMA read. Based on the ZERO framework, the monitoring system is expected to achieve no CPU occupation at the monitored host, low latency and high throughput, finally avoiding any interferences to services and fulfilling the QoS requirements of large-scale distributed monitoring.

However, there still exist several challenges to achieve the above goals. As shown in Figure 1, traditional monitor collects and processes raw metrics from the monitored processes, then upload metrics to the remote host, which inevitably causes CPU occupations. How to manage memory regions of system/application metrics and expose them to the remote host, finally achieving zero-overhead monitoring via RDMA read, is challenging. On the other hand, the remote

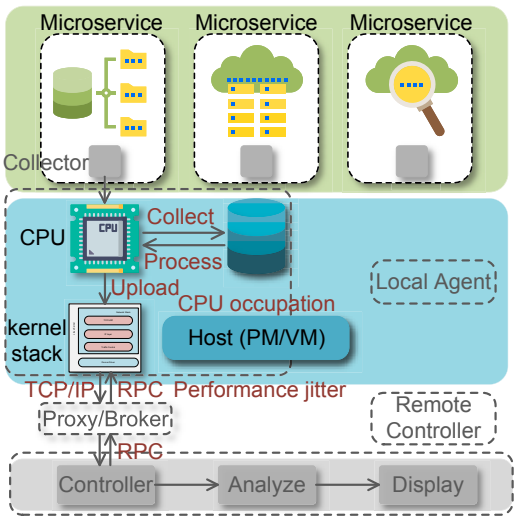


Figure 1: Traditional monitoring System.

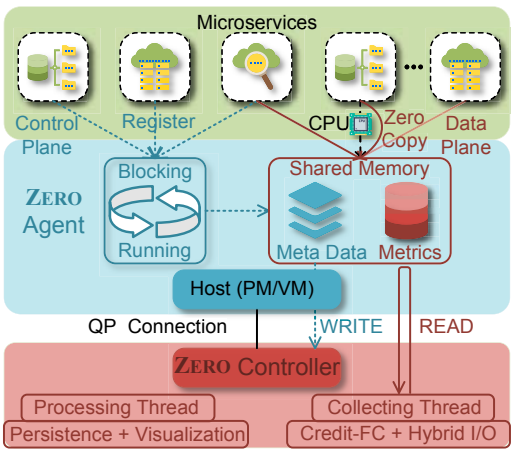


Figure 2: ZERO monitoring system.

monitoring host becomes bottlenecks in the receiver-driven model, as a tradeoff of offloading local monitoring overhead to the remote host. The remote host performs RDMA read on many monitored hosts, resulting in incast problem [85] [1]. The remote host not only collects metrics, but also processes raw metric for further operations, all of which are CPU intensive. How to enable large-scale monitoring with network/CPU bottlenecks is challenging as well.

To access raw metrics with no CPU occupation, ZERO proposes the novel control plane and data plane. For the ease of clarity, we separate ZERO into local agent and remote controller (Figure 2). To achieve high scalability in reliable connection (RC) mode [26, 82], system/application metrics are managed by one agent and share one queue pair (QP) connection. In the control plane, ZERO agent provides universal interfaces for systems/applications to register the memory regions of their metrics at the RDMA NIC (RNIC). The metadata of these metrics are recorded at the control region. ZERO controller can thus acquire metadata of metrics from the control region as the prerequisite to access raw metrics. All metrics only need to register once if the metadata is not updated, after which ZERO agent enters blocking mode. In the data plane, the memory regions of metrics (data region) are exposed to the agent process via shared memory, finally to the remote controller. The ZERO controller can thus perform RDMA read on the data region directly without involving memory copies and CPU usages at the monitored host. As a result, ZERO achieves disposable overhead in the control plane and zero overhead in the data plane.

To deal with the network/CPU bottlenecks at the controller, ZERO proposes credit-based flow control (Credit-FC) and hybrid I/O model. We observe that the receiver-driven model is superior to the traditional sender-driven model (*i.e.*, agent actively uploads metrics as per heartbeat): i) collecting metrics on demand to ensure the QoS of monitoring on latency; ii) limiting the total in-flight data of concurrent flows to avoid network congestion/interference. Accordingly, ZERO proposes Credit-FC to mitigate the incast problem while fulfilling the latency/throughput requirements of monitoring. On the other hand, ZERO introduces hybrid I/O model (a combination of event driven and busy polling mechanisms) and adopts thread dispatching to remedy the CPU bottlenecks of collecting and processing metrics, respectively.

As case studies, we integrate application (Redis [12]) metrics, system (kernel/containers [27, 76]) metrics and eBPF [3] metrics into the ZERO framework to demonstrate its generality and flexibility. We also share our experience about building large-scale monitoring system using RDMA.

The major contributions of this paper are summarized as follows:

- We propose the first zero-overhead monitoring system, ZERO, to resolve limitations of traditional monitoring system in cloud-native monitoring.
- We tackle several challenges of zero-overhead monitoring, including data plane with no CPU involvement, network congestion/interference caused by monitoring traffics, and CPU bottlenecks at the controller.
- We have deployed and evaluated ZERO in Alibaba cloud. ZERO achieves no CPU occupation at the monitored host and supports $1 \sim 10k$ hosts with $0.1 \sim 1s$ sampling intervals. We also share our experience with ZERO.

The paper is organized as following. Section 2 introduces the background and motivation. Section 3 proposes zero-overhead monitoring. Section 4 designs and implements ZERO framework. Section 5 presents case studies. Section 6 evaluates the proposed design. Section 7 introduces the experience and future work. Section 8 discusses related works and Section 9 concludes this paper.

[1] Incast problem happens when multiple senders transfer data to one receiver simultaneously.

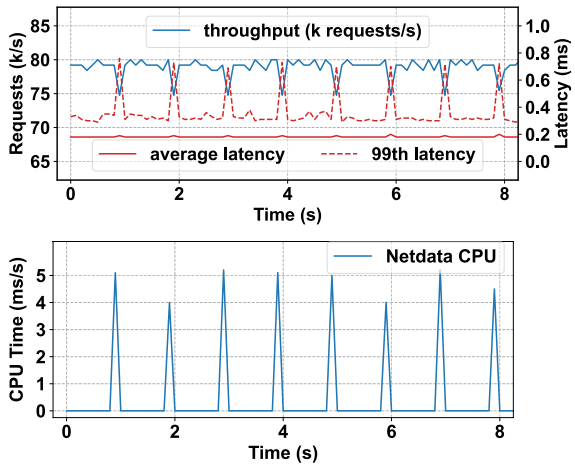


Figure 3: Monitor interfering with services.

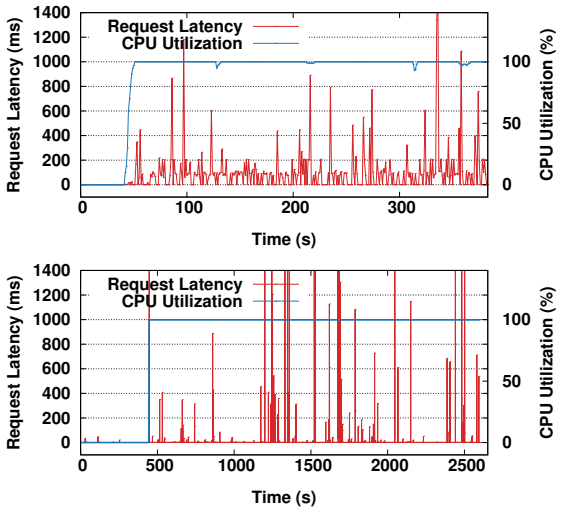


Figure 4: Monitoring jitters with high system loads in Netdata (upper) and Prometheus (bottom).

2 Background and Motivation

In this section, we further elaborate cloud native monitoring and the inherent limitations of traditional monitoring system.

2.1 Cloud-Native Monitoring

Monitoring system deployed at the cloud-native infrastructure, namely, cloud-native monitoring, is indispensable to ensure the SLA of cloud services. Monitor collects the bottom-layer system metrics, such as the utilization of physical resources (CPU, memory, *etc.*). Based on system metrics, monitoring system performs health checks on the underlying system, makes early alert and provides suggestions to administrators [18]. Furthermore, by analyzing the historical resource consumption and performance variations, cloud providers improve system utilization and lower operational expenses (OpEx) [27, 28, 41]. On the other hand, the upper-layer application metrics, *e.g.*, requests per second of key-value service [12,45], directly reflect user activities and functional state of applications. The monolithic applications are decoupled to thousands of microservices [33], all of which are monitored to track and ensure the SLA.

The novel cloud-native applications together with the essential infrastructure bring new challenges/demands to cloud-native monitoring, along with two major issues in commercial deployment.

How to avoid monitor interfering with services? Microservices have much stricter requirements of QoS compared with typical applications [33]. However, the cloud-native environment is highly resource constrained. For example, Alibaba cloud adopts mixed deployment of CPU-intensive online service [42] and I/O-intensive batch jobs [93] at the same host, to maximize resource utilization [43, 78] and reduce long-term capital expenses (CapEx). Microsoft Azure also reports that 80% of VMs only have $1 \sim 2$ vCPU cores [27]. The monitor should have no CPU occupation to avoid contentions with cloud services.

How to ensure QoS of monitoring? Cloud-native monitoring needs to support massive metrics with rapid variations. Cloud providers, such as Alibaba, Netflix and Uber, need to monitor millions of metrics with hundreds to thousands of microservices [80]. Besides, these application metrics (financial transactions, social network and e-commerce [67, 94]) and system metrics (CPU, memory, network [6, 54]) have rapid variations with a time scale of seconds/milliseconds. To track and ensure SLA of services, monitoring requires high QoS from the perspective of latency and throughput.

2.2 Traditional Monitoring System

We next introduce traditional distributed monitoring systems [10, 51, 65] and elaborate their limitations as the motivation of ZERO. As shown in Figure 1, multiple collectors acquire application/system metrics via specific interfaces, meanwhile raw metrics are processed as the final outcomes (Section 5). After that, the collected metrics are uploaded to the remote controller for further analysis and visualization. Each step occupies host CPUs for memory copy, calculation, network transmission, *etc.*, which inevitably interfering with services running on the host. On the other hand, traditional monitor massively relies on the kernel's TCP/IP network stack to transmit metrics. However, kernel data processing overhead has became the main bottleneck of end-to-end latency/throughput [90]. State-of-the-art works thus offload network functionality from kernel to user-space network stack [39, 53, 64] or hardware [35, 44, 70].

We next elaborate the limitations of traditional monitors in our real deployment. We consider two representative open-

sourced monitors, i.e., Netdata [10] and Prometheus [11]. We observed that enabling monitors causes jitters of online service in Alibaba "double eleven" shopping festival with high loads. We use Netdata to monitor a high-loaded host running Redis services with 1s sampling interval. As shown in Figure 3, monitor process causes jitters of Redis service, *i.e.*, the throughput declines by 6.25% while tail latency increases by 2× periodically, due to the CPU occupation in each monitoring cycle. We analyze that reasons for such "interference spikes" are two folds. First, the deployment of service and monitor processes may adopt default CPU scheduling or specific CPU bonding [48], where contentions happen when service/monitor processes are scheduled/bonded to the same CPU core. Second, the CPU utilization keeps on high water level and exhibits burst natures, especially during sales promotion with high loads. Thus, the duty-cycled monitoring process with slight CPU occupation ($1 \sim 5\%$) already causes severe interference (Figure 3). The CPU breakup of monitor shows that the uploading phase occupies $5 \sim 10\%$ of the total CPU utilization while the collecting phase occupies the majority. On the other hand, the QoS of monitoring is highly affected by the system load or CPU quota set by the cloud operator. The latency of monitoring increases by more than 10×, when host CPU is saturated or CPU quota is reached (Figure 4). Assigning dedicated cores for monitoring may avoid these problems, however, causes large wastes of resources and high CapEx. Besides, the monitoring process may be blocked to request metrics via service interface (Section 5).

3 Zero-overhead Monitoring

To avoid resource contentions with services and ensure QoS of monitoring, we propose the zero-overhead monitoring system, namely ZERO, for cloud-native monitoring. ZERO exploits features of monitoring metrics and RDMA capability in modern data center. The basic idea is that most of raw metrics are counters updated at fix memory regions — the remote controller can thus obtain these metrics by performing RDMA read on these memory regions — without any CPU involvement at the monitored side. We next elaborate possibilities and challenges of realizing ZERO.

Metric features. ZERO is based on two features of monitoring metrics. First, most of the monitoring metrics are counters. Systems/applications update these metrics at fixed memory region after initialization. For instance, the number of stored/evicted key-value pairs in Redis and the number of sent/received packets recorded by kernel stack are all counters. Second, processing of raw metrics is simple algebraic calculation and can be offloaded to the controller. For example, Redis exports statistic data on the raw metrics and per-CPU counters are summed to get the final kernel metrics. These features are general to system/application metrics and ZERO can thus support various metrics as an universal framework (Section 5).

RDMA support. ZERO then leverages *one-sided* RDMA operation to read raw metrics/counters, to achieve no CPU/kernel involvements at the monitored host. RDMA has been widely used in data centers and provides new characteristics of low latency (as low as 1 μs), high bandwidth (more than 100Gbps) and kernel/CPU bypass. RDMA supports both *one-sided* and *two-sided* operations. The one-sided operations directly operate on the remote memory via *read* and *write* without involving the remote server's CPU. To perform one-sided RDMA operations, one needs to register the *memory region* (MR) at the RNIC of remote host and acquire the generated *remote protection key* (rkey). The two-sided operations, *i.e.*, *send* and *recv*, communicate via an interface similar to socket. In the following paper, we refer to RDMA read, write, send and recv as READ, WRITE, SEND and RECV, respectively. RDMA hosts create queue pairs (QP) consisting of a send queue and a receive queue, then post RDMA operations on send/receive queue to communicate with the remote host. RDMA transport supports reliable or unreliable connection (RC/UC) and unreliable datagram (UD). One-to-one connections between QPs are required in RC/UC mode, whereas one-to-many communication is supported in UD mode. Different transport types support different subsets of RDMA operations, and READ operation is only supported in RC mode.

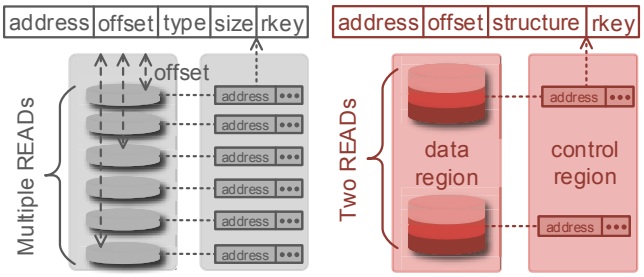
Challenges. While ZERO is expected to achieve zero-overhead monitoring, there still exist two challenges to make the idea practical. First, we observe that most CPU time of traditional monitor are spent on collecting metrics from system/application processes (Section 2.2). ZERO also needs to eliminates such overheads in its data plane, besides the transmission overheads offloaded to RNIC. Second, controller is bottlenecked on both network and CPU with large number of monitored hosts, as all monitoring overheads are offloaded to the remote controller. With all these challenges, the key innovation of Zero lies in effectively exploiting one-sided RDMA and designing the separate control/data plane to realize zero-overhead monitoring. Zero further incorporates several designs to resolve practical issues (network congestion/interference, scalability) in distributed monitoring.

4 Design and Implementation

In this section, we present the overview of the ZERO framework. We then introduce the design and implementation of ZERO in details.

4.1 Overview

As shown in Figure 2, ZERO proposes the novel control plane and data plane to collect raw metrics without CPU involvements at the monitored host. ZERO adopts receiver-driven model to collect metrics from large number of hosts, and deals with the network and CPU bottlenecks at the controller.



(a) Inefficient READ. (b) Efficient READ.

Figure 5: READ w/o (left) or w/ (right) memory management.

The scalability of RDMA-based system is constrained by the on-chip memory (SRAM) of RNIC [26, 82]. To achieve high scalability at the controller, all metrics at the host are managed by one agent and share one QP connection. In the control plane, ZERO agent provides interface for systems and applications to register the memory region of their metrics (data region) at the RNIC. The metadata of metrics (*e.g.*, address, rkey) are written into the control region. ZERO controller can thus obtain metadata by reading the control region and access to raw metrics via reading the data region. The agent process is blocked unless systems/applications need to register/update metrics. In the data plane, the memory region of metrics are exposed to the agent process via shared memory, finally to the remote controller. The controller performing READ on the data region to acquires metrics, which achieves zero copy and no CPU involvement at the monitored host. In the real deployment, the control-plane overhead is usually disposable — the metrics only need to register once — then keeps in use or updates after a long time. The data plane has no CPU occupation as expected.

ZERO supports large-scale monitoring via single controller. ZERO adopts thread dispatching, where only $1 \sim 2$ threads are used to collect metrics and the rest cores are used to process metrics in parallel for further operations, *e.g.*, visualization and persistence. ZERO controller uses receiver-driven model, *i.e.*, issuing READ requests on the monitored hosts to collect raw metrics, which is superior to the traditional sender-driven model. ZERO achieves high monitoring QoS by posting READ requests on demand and avoids network congestion/interference by controlling the total in-flight data. Accordingly, credit-based flow control (Credit-FC) and hybrid I/O model are proposed to remedy the incast problem and CPU bottleneck, respectively.

4.2 ZERO Control Plane and Data plane

We introduce the ZERO control plane and data plane, together with ZERO framework usage and interface in this subsection.

Control plane. ZERO agent deals with registration requests from systems/applications, which uses UNIX domain socket for inter-process communications (IPC). To handle the request, the MRs of monitoring metrics are registered at the RNIC and the metadata of MRs (*e.g.*, address, type, size, rkey) are recorded at the control region (Figure 5a). Note that the MR is pinned after metrics are registered and will be released only after metrics expire. ZERO agent registers the control region at the RNIC and builds a QP connection with the remote controller in advance. The controller can thus acquire the metadata of metrics by reading and parsing the control region, then access raw metrics.

The control plane has disposable overhead. ZERO agent inevitably occupies host resources to deal with registration requests. However, we observe that the control-plane overhead is disposable for most of metrics. We divided metrics into three types according to the practice in Alibaba cloud. First, metrics of the underlying systems and persistent services, *e.g.*, database and storage services [56, 91], are usually immobile once registered. Second, some services may be dynamically created/destroyed. For example, e-commerce services are periodically expanded/shrunk according to the number of users shopping online [60]. Third, some user requests are served by ephemeral serverless *functions* to mitigate the cost of long-lived services with intermittent activities [25, 33].

Generally, ZERO agent enters into blocking mode with no CPU occupation (Figure 2). When services change, ZERO agent resumes to (de)register metrics and update the corresponding control region. For the first two cases, ZERO agent handles registration requests and updates control regions infrequently with negligible overhead. ZERO controller also only reads control regions once in a long period. In the serverless case, ZERO agent may frequently (de)register metrics of serverless *functions*. ZERO agent further uses WRITE to write the updated metadata into the remote control region (Figure 2). ZERO controller can thus obtain raw metrics by only reading the data region with one RTT. Note that the overhead of ZERO using WRITE or SEND/RECV is still much lower than that of the traditional monitor (Section 6.2).

Data plane. To eliminate the overhead of collecting metrics from multiple processes and avoid frequent memory copies, ZERO exclusively adopts shared memory in its data plane. Specifically, ZERO agent uses `mmap` operations, which are invoked when registering metrics, to expose MRs of metrics (data region) to the agent process, finally to the remote controller. The `mmap` operation takes 4KB page as the basic unit. However, metrics are not necessarily locating at the page header (Figure 5). ZERO agent calculates the page header of metrics to `mmap` their pages. As shown in Figure 5, ZERO agent records the page address and page offset in the control region. To protect data region from being modified by local agent or malicious remote host, ZERO sets read-only access to the data region via `mmap` flags and uses the rkey mechanism inherently supported by RDMA. ZERO controller obtains raw metrics by performing READ on the data region.

The data plane has zero overhead. The data plane achieves zero CPU occupation, zero copy, and no extra memory footprint at the monitored host, via the shared memory design.

```
// type one, specifying attributes of variables
struct disk my_disk{
    .disk           = "sda",
    .hash    = 0x000f3456, ...
} __attribute__((section(".zero_init"));
//type two, using allocator
struct disk *my_disk = zero_malloc(sizeof(struct disk));
```

Figure 6: Management interface for two types of metrics.

An alternative solution of copying metrics when updating causes frequent CPU occupations for memory copies, and extra memory footprints. Besides, ZERO data plane ensures the read-write consistency between remote and local memory. Most of application/system metrics are defined as atomic variables, which are updated atomically in the shared memory. The atomic update only needs $1\sim3ns$ in Intel Haswell architecture [74] — three orders of magnitude lower than that of RDMA operations ($1\mu s$) — the memory consistency is guaranteed between update and READ. For non-atomic variables, ZERO uses bit flags to indicate the states of updating. FaRM [31] and Pilaf [68] use a similar method to ensure data consistency with READ. ZERO controller will check whether metrics are read correctly via bit flags and retry in the next cycle. We eliminate all synchronous locks for zero overhead, which may cause inconsistency under rare race conditions while ensuring accuracy for most cases.

Memory management is indispensable. While local host achieves zero overhead in ZERO data plane, we observe that simply performing READ on massive metrics cannot achieve desirable performance. Because metrics are distributed across the process/kernel space with discrete memory addresses, ZERO controller needs to read large number of entries in control region as well as metrics in data region (Figure 5a). However, the bandwidth of READ falls rapidly and the latency is nearly doubled when the number of MRs increases from 100 to $10k$, due to evictions in the RNIC SRAM [26, 82]. ZERO introduces memory management to reduce the number of MRs and READ requests required to collect massive metrics. Specifically, ZERO proposes two memory-management mechanisms for two types of metrics (Figure 6). First, many metrics are global variables or data structures. One can mark these metrics by specifying attributes of variables [13]. The compiler will distribute these metrics to the same data segment. Second, metrics are defined as pointers to variables. ZERO provides a memory allocator for these metrics. Specifically, the MRs of metrics are allocated with continuous space via the allocation API. The core idea of both methods is concatenating metrics to the same MR to support massive metrics. Besides, data region is aligned as `struct` and recorded at the control region for the ease of memory parsing at the controller. As shown in Figure 5b, ZERO controller only needs to post one READ request on the same MR to get a list of metrics.

Framework usage and interface. ZERO can be easily deployed at hosts (PM/VM) with RDMA support. ZERO agent and controller need to be initiated at the local and remote host respectively. Systems/applications then invoke agent API to manage and (de)register their metrics. The agent accomplishes all control-plane operations, *e.g.*, `mmap`, updates of control region, when handling (de)register requests.

4.3 Scaling-out Monitoring

We next present how to support large-scale monitoring with single ZERO controller. ZERO proposes credit-based flow control (Credit-FC) and hybrid I/O model, to avoid network congestion/interference and remedy CPU bottlenecks respectively. ZERO controller needs to collect and process metrics from large number of hosts, while fulfilling the monitoring QoS in latency and throughput. To achieve this goal, the controller adopts thread dispatching to collect and process metrics in parallel with individual threads.

Collecting metrics. The controller only assigns $1\sim2$ threads to collect metrics. ZERO achieves high efficient network I/O with single thread by posting READ requests then polling completions on multiple QPs in batch. This is feasible because both `post_send` and `poll_cq` are fast non-block operations. According to our experiment, the batch operations only add negligible latency (tens of μs).

Receiver-driven model is superior to send-driven model. Issuing READ on data region turns out a receiver-driven model to collect metrics from multiple hosts. The receiver-driven model has two benefits compared with the traditional sender-driven model. The controller posts READ requests on demand to meet the target latency or updating frequency in monitoring. Besides, it also facilitates to avoid network congestion/interference by limiting the total in-flight data of concurrent flows. We next intuitively formulate the scale-out ability of such receiver-driven model. The monitored hosts have different requirements in terms of updating interval U and data size S, *i.e.*, controller needs to collect S bytes in every U seconds for a specific host. Assuming the bandwidth B of the receiver is fully utilized, the maximum number of supported hosts $n = B \times U/S$. Our deployment shows that ZERO supports at least $1k$ hosts with 128KB metrics and $100ms$ sampling interval.

Credit-based flow control. Concurrent READ requests generate burst network traffics, which are transmitted from multiple hosts to the controller simultaneously, resulting in severe incast problems. ZERO introduces Credit-FC to remedy the incast problem, which works as follows.

Large-sized READ requests are segmented into fix-sized fragments with 4KB page size. ZERO chooses such moderate size due to three considerations: i) page is the basic unit of shared memory, which can accommodate $1k$ 32-bit metrics and fulfill the demands of most services; ii) the number of READ requests is bounded as the total size of metrics in

single host is general hundreds of KBs; iii) the small size facilitates congestion control in severe incast.

Subsequently, credits are used to limit the total in-flight data of concurrent flows (identified by a QP). Posting READ requests or polling completion events will consume or regain credits for the target flow. The state-of-the-art works [44, 70] adopt bandwidth-delay product based flow control (BDP-FC), which bounds the in-flight data per flow by the BDP of the network. However, BDP is large enough to cause congestion and trigger explicit congestion notification (ECN) packets or priority-based flow control (PFC) pause frames [95], with large number of concurrent flows (Figure 11c). ZERO proposes Credit-FC to limit the total in-flight data. Specifically, the credit of each flow C_f is set to T/n, where T is the total credit and n is the number of concurrent flows. Finally, Credit-FC effectively avoids triggering ECN/PFC (Section 6.3) and network interference with service traffics (Section 7).

Hybrid I/O model. To avoid the thread performing network I/O being saturated, ZERO proposes the hybrid I/O model incorporating event driven and polling mechanisms, which is similar to NAPI in Linux [73]. The event-driven I/O can effectively avoid CPU occupation for busy polling. Each QP is associated with an event channel to notify a new (first) completion event. The I/O multiplexing interface, *e.g.*, `epoll`, is used to listen the `fd`s of multiple event channels. The collecting thread blocks until completion events are notified from one or multiple QPs, then `polling` the in-flight requests of the corresponding QPs. We observe that the event-driven model effectively reduces CPU utilization with large number of in-flight requests (Figure 12b). However, when the monitored data size is too small with only several requests after segmentation, the `epoll` syscall and thread blocking incur high variations in READ latency (Figure 12a). The controller thus uses busy `polling` for hosts with small number of requests. In the hybrid I/O model, ZERO assigns two threads to perform event-driven polling and busy polling, respectively. Note that both threads share Credit-FC.

Processing metrics. The controller dispatches multiple threads to processing raw metrics in parallel. Specifically, the MRs of the collected metrics are placed into the appropriate queue where each MR can be handled by one of multiple processing threads. Each MR concatenated by a list of metrics is parsed as `struct` directly according to the metadata recorded at the control region. The parsed metrics are then processed by reproducing the same calculations which are originally performed by the monitored host. Finally, the controller imports processed metrics into InfluxDB [7] for persistence and uses Grafana [5] for visualization.

5 Case Study

In this section, we present how to integrate application/system metrics into the ZERO framework using three typical cases of Redis [12], Linux kernel [81] and eBPF [3].

Redis Case. Redis [14] has been widely deployed in Alibaba cloud as database, cache, and message broker, providing low-latency in-memory data structure storage services. Traditional monitor acquires Redis metrics by requesting `INFO` interface of Redis server. Traditional monitor thus occupies the resource of Redis server and the host to obtain metrics. As a comparison, ZERO only needs to register metrics and requires no resource occupation for collecting metrics. There exist more than two hundred metrics in each Redis service instance. The naive implementation is performing READ on these metrics one-by-one (see Figure 5a), resulting in high latency and CPU utilization. To resolve this problem, we use allocation API (type two) to allocate and structuralize Redis metrics with continuous memory. As shown in Figure 5b, ZERO only needs to register and perform READ once for each Redis instance in our implementation. ZERO controller then parses metrics as `struct` according to the memory structure.

Linux Kernel Case. Linux kernel exports system metrics to user space via `proc` interface, which creates files under `/proc` directory and bonds corresponding kernel functions. Traditional monitor usually needs to read hundreds of `proc` files to get all system metrics, which incurs extra overhead for user/kernel-space processing. With ZERO framework, kernel metrics are registered at the ZERO agent then exported to the ZERO controller directly. We use ZERO to monitor metrics managed by container namespace for the container-based services [1]. Kernel metrics are usually implemented as lock-free per-CPU counters to avoid locking overhead. ZERO controller needs to READ all replications of metrics in each CPU core, locating at separate pages. Production cloud environment adopts fine-grained resource assignment and isolation, in which more than 90% hosts only have $1 \sim 4$ CPU cores [27]. ZERO only needs $1 \sim 4$ requests to obtain kernel metrics.

eBPF Case. The extend Berkeley Packet Filter (eBPF) [3, 66] is an evolving technology, which can dynamically attach program to running kernel for tracing, instructing, and even controlling the kernel code path. eBPF has been widely used in cloud computing for monitoring [6, 54], networking [38, 83], virtualization [21, 22] and security [29, 30]. We use eBPF to monitor traffics and retransmissions of large number of TCP connections in MaxCompute [32, 93] service. eBPF provides in-kernel data structure, called `map`, to enable control and data messages delivery within kernel or between kernel and user space. eBPF attaches probes to kernel/application functions at runtime and exports metrics, events and histograms to eBPF `map`. User process reads the entry of eBPF `map` via syscall. However, reading large number of entries incurs large overhead due to frequent syscalls. To integrate eBPF into ZERO framework, eBPF `array map` is adopted, which supports `mmap` operations (from Linux kernel 5.5+) and can thus export its memory to ZERO agent directly.

Name	Nodes	Hosts	OS kernel	Intel Xeon CPU code	Mellanox NIC	Protocol	ECN	PFC
Cluster1	65 × PMs	1024 × VMs	Linux 5.5	E5-2682 (64 cores)	2 × 25GbE ConnectX-4 Lx	RoCEv1/2	✗	✓
Cluster2	9 × PMs	1024 × Containers	Linux 3.10	Platinum 8369B (64 cores)	200GbE ConnectX-6 Dx	RoCEv1/2	✓	✓

Table 1: Deployment environment.

6 Evaluation

6.1 Evaluation Setup

In our evaluation, we adopt a multi-phase deployment with two typical clusters, as summarized in Table 1. Initially, we deploy Zero in a test environment (Cluster1) with a rational scale that mimics the production environment for demonstration. We next deployed ZERO in a public-cloud environment (Cluster2), which covers common cloud services in production. The deployment scale has continued to grow according to the feedback of canary testing and the actual demands of services. In Cluster1, services operates in guest VMs with 4 vCPU cores. The RNICs are virtualized and assigned to VMs via passthrough [87]. In Cluster2, services are deployed in containers running on bare-metal servers [92]. Each VM/container is monitored independently to evaluate the scalability of ZERO. Note that both configurations are typical in Alibaba cloud-native platform [15]. We use ZERO to monitor typical services, *e.g.*, Redis [14], container [1] and MaxCompute [32, 93].

We evaluate ZERO performance from three aspects:

- *CPU Utilization:* The CPU utilization is defined as occupied CPU time per second. We use `perf` tool to measure the CPU utilization of both ZERO agent and controller. We verify the CPU involvement of monitor in control plane and data plane. We also need to concern about the CPU utilization of ZERO controller for scalability.
- *Latency:* The latency of ZERO is the time used to READ all metrics from the monitored host. For traditional monitor, the latency consists of time used to collect/process all metrics and time used to upload all metrics. We verify whether monitor can meet up the updating frequency of metrics by latency.
- *Throughput:* The throughput is the collected bytes per second during each monitoring period. We verify whether monitor can support massive metrics by throughput.

We test the impact of the following parameters on monitoring performance:

- *Sampling Interval:* The required sampling interval is determined by the updating frequency of metrics. We evaluate ZERO with $10 \sim 1000ms$ sampling intervals and use $1000ms$ by default.
- *Number of Instances/Connections/Requests:* All of three parameters impact the CPU utilization and data size of monitoring. The number of service/container instances varies from 10 to 40, with a default value of 10. The number of TCP connections varies from $1k$ to $16k$, with a default value of $1k$. The number of work requests varies from 8 to 128 for a host, with a default value of 32.
- *Number of Hosts:* One ZERO controller is deployed to monitor multiple hosts running ZERO agent. We increase the number of hosts (from 64 to 1024) to evaluate the scalability of ZERO framework.

Metric	Monitor	Redis	Kernel	eBPF
Total Latency (ms)	Baseline	$0.7 \sim 19.3$	$0.5 \sim 1.6$	$0.8 \sim 12.5$
	ZERO RPC	$0.08 \sim 0.18$	$0.14 \sim 0.36$	$0.10 \sim 1.02$
	ZERO	$0.05 \sim 0.14$	$0.07 \sim 0.23$	$0.08 \sim 0.87$
Agent CPU Utilization (%)	Baseline	$0.5 \sim 45$	$0.01 \sim 4$	$0.08 \sim 6$
	ZERO RPC	$0.01 \sim 0.55$	$0.08 \sim 0.9$	$0.05 \sim 0.68$
	Control plane	$0.05 \sim 0.07$	$0.8 \sim 1.5$	$0.04 \sim 0.05$
	Data plane	0	0	0

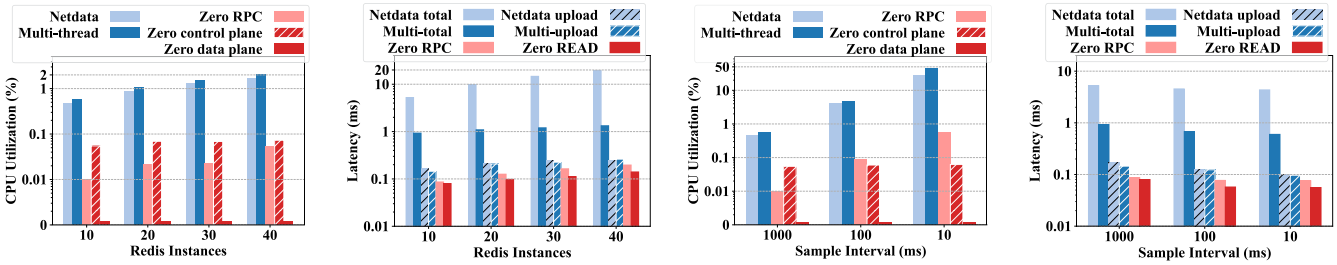
Table 2: Summary of ZERO overhead.

We use the state-of-the-art monitoring system named Netdata [10], existing *NX* monitor in Alibaba cloud, and ZERO RPC as comparison benchmarks:

- *Netdata:* Netdata is widely used by cloud providers, e.g., Amazon Web Services (AWS) and Microsoft Azure [8]. Netdata integrates application/system metrics in one agent by requesting application interface or reading `proc` files respectively. The agent uploads metrics to the controller as per heartbeat or the controller acquires metrics via sending RPC requests to the agent.
- *NX:* NX is a network monitoring tool deployed in Alibaba cloud. NX exports network metrics as logs, e.g., info of TCP connections, then uploads the collected metrics via log service. We have integrated ZERO framework into the NX monitor to improve its performance.
- ZERO *RPC:* ZERO RPC adopts the same data plane to access raw metrics. However, ZERO RPC is implemented via SEND/RECV instead of READ. The controller issues RPC requests to the agent, after which agent returns metrics as response. We explore design alternations of two-sided RDMA via ZERO RPC.

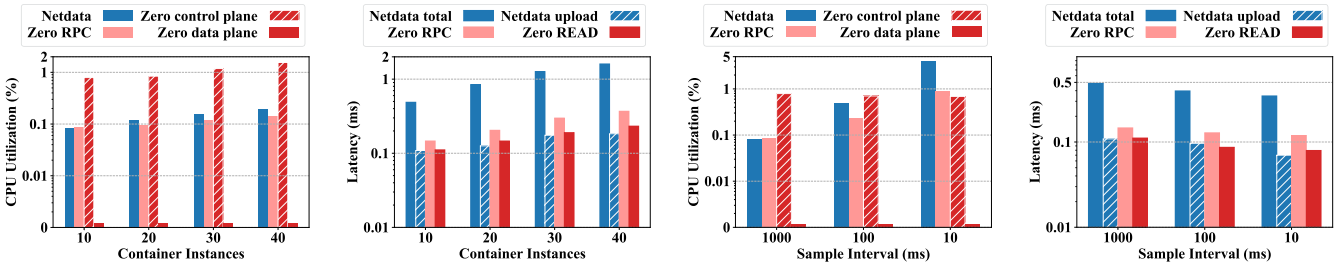
6.2 ZERO Overhead

We evaluate the CPU utilization and latency of ZERO in monitoring Redis, container and MaxCompute services. We present overall performance and summarize two key observations, followed by detailed micro-benchmarks for each case.



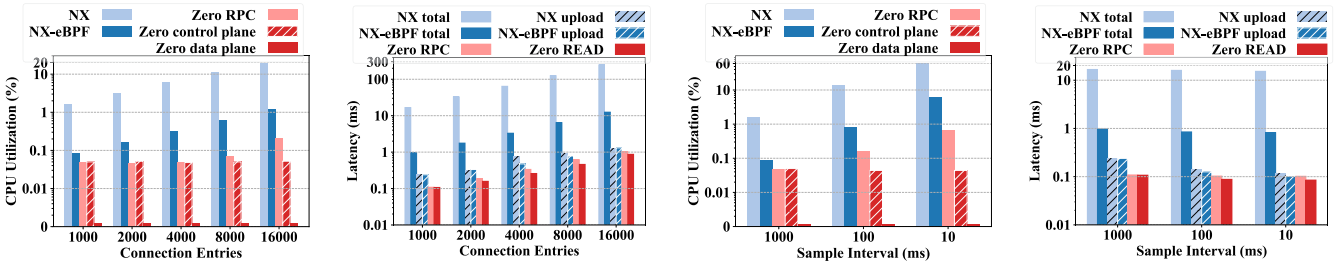
(a) Agent CPU vs. Instances. (b) Latency vs. Instances. (c) Agent CPU vs. Intervals. (d) Latency vs. Intervals.

Figure 7: Monitor performance with $10-40$ Redis instances and $10-1000ms$ sampling interval.



(a) Agent CPU vs. Instances. (b) Latency vs. Instances. (c) Agent CPU vs. Intervals. (d) Latency vs. Intervals.

Figure 8: Monitor performance with $10-40$ container instances and $10-1000ms$ sampling interval.

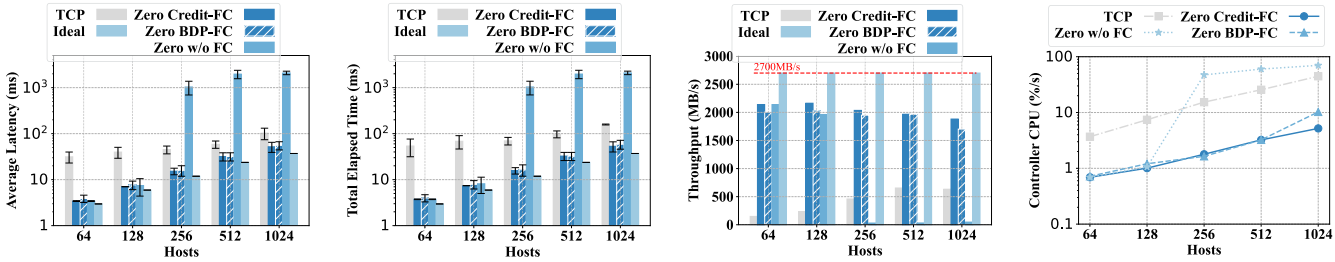


(a) Agent CPU vs. Entries. (b) Latency vs. Entries. (c) Agent CPU vs. Intervals. (d) Latency vs. Intervals.

Figure 9: Monitor performance with $1-16k$ TCP connections and $10-1000ms$ sampling interval.

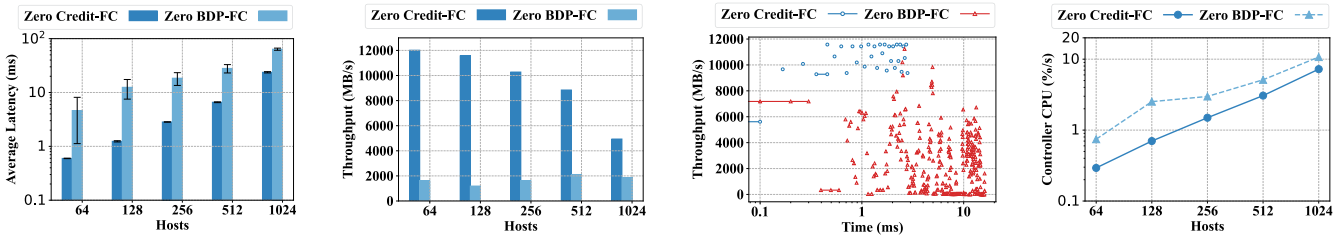
Overall. Table 2 summaries the overhead of ZERO monitor. We focus on the monitoring latency and the CPU utilization of ZERO agent. Both Netdata or NX are referred as baselines. First, ZERO monitor reduces latency by one/two order of magnitudes compared with baselines. Our following breakdown of total latency reveals that baseline methods spend most of time to collect metrics from system/application processes. The TCP-based baselines actively upload metrics and achieve similar latency in uploading phase as ZERO READ. ZERO RPC has higher latency than ZERO READ because each RPC requires at least two RTTs [45, 68]. Second, ZERO agent achieves disposable overhead in control plane and zero overhead in data plane. The CPU utilization of ZERO control plane only increases slightly when registering more MRs, which is not affected by the sampling interval. The CPU utilization of ZERO data plane is always zero as expected. On the contrast, the CPU utilization of baselines reaches very high values with lower sampling intervals. ZERO RPC eliminates the overhead of collecting metrics, however, the CPU utilization for posting SEND/RECV requests and polling completions still increases with lower sampling intervals. In summary, the benefit of ZERO to cloud services is enabling higher SLA of infrastructure, which effectively avoids performance jitters caused by CPU interference. Zero also improves monitoring performance, which reduces latency by $1\sim2$ orders of magnitude and increases throughput by $3\sim6\times$ (Section 6.3).

Redis Case. The monitoring performance of Redis case is shown in Figure 7. Netdata uses single or multiple threads to collect metrics from multiple Redis instances. With the increment of service instances, the CPU utilization of single- and multi-thread Netdata increase from $0.4\sim0.5\%$ to $1.5\sim2\%$, which already severely interferes Redis services (Figure 3).



(a) Average latency vs. Hosts. (b) Total latency vs. Hosts. (c) Throughput vs. Hosts. (d) Controller CPU vs. Hosts.

Figure 10: Monitor performance with 64 – 1024 hosts × 128KB data in Cluster1.



(a) Average latency vs. Hosts. (b) Throughput vs. Hosts. (c) Single run with 256 Hosts. (d) Controller CPU vs. Hosts.

Figure 11: Monitor performance with 64 – 1024 hosts × 128KB data in Cluster2.

With 10× lower sampling interval, the Netdata CPU increases 10×, while the gap of CPU utilization between single- and multi-thread becomes larger. However, the multi-thread Netdata achieves 0.7 ∼ 1*ms* latency (Figures 7b and 7d). Netdata thus trades off CPU utilization with latency. On the contrary, ZERO has negligible CPU utilization (< 0.1%) in its control plane, which is a one-off expense independent of sampling intervals as shown in Figure 7c. ZERO data plane has zero CPU overhead denoted by the tiny pillar in Figures 7a and 7c. ZERO reduces latency by one/two order of magnitudes compared with Netdata, as shown in Figures 7b and 7d.

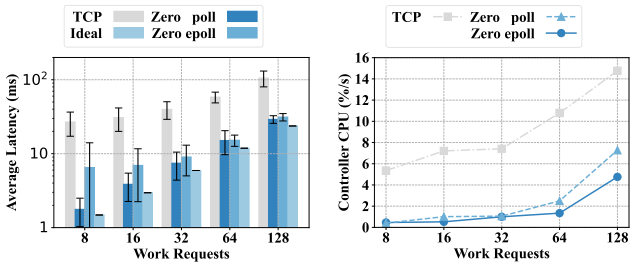
Linux kernel case. Figure 8 illustrates the performance of Linux kernel case, where ZERO monitors multiple container instances. As shown in Figures 8a and 8c, the CPU utilization of the ZERO data plane is zero. The CPU utilization of ZERO control plane is high, because of the frequent invocations of `mmap` when registering per-CPU kernel metrics. However, the total CPU time of ZERO control plane is fixed, *i.e.*, 3.4 ∼ 14.7 ms to register 10 ∼ 40 instances. As shown in Fig 8c, the overhead of ZERO control plane is only 0.8% independent of sampling intervals, while the Netdata CPU increase to 4% with 10ms sampling interval. ZERO achieves 0.07 ∼ 0.1ms latency, which is an order of magnitude lower than that of Netdata (0.5 ∼ 1.6ms). ZERO may have higher READ latency than the uploading latency of Netdata. Because kernel metrics are per-CPU counters, ZERO controller needs *n* requests to obtain *n* copies of counters in each CPU.

eBPF case. Figure 9 shows the performance of monitoring TCP connections of with NX, NX-eBPF and ZERO. NX monitor has the highest CPU utilization (1.6 ∼ 59%) and latency (16 ∼ 250ms), due to its outdated implementation, which traverses all TCP connections using the kernel `tcp_diag` interface. NX-eBPF introduces eBPF to low the overhead of getting TCP info, while ZERO further eliminates the syscall and memory copy overhead of reading eBPF map. As shown in Figures 9b and 9d, ZERO performs 40%∼80% lower latency than that of NX. Similar with Redis and Linux kernel case, the CPU utilization of ZERO data plane is still zero.

6.3 ZERO Scalability

We then evaluate the scalability of ZERO. The controller adopts receiver-driven model to obtain raw metrics, which issues TCP-based RPC or READ requests to the agent. The collecting phase at the monitored host is omitted to compare the raw performance. The controller adopts busy `polling` by default. The ideal result is obtained via `ib_read_bw` [16]. Our three key observations are summarized as follows.

READ achieves better performance. The TCP-based baseline achieves much higher average latency compared with ZERO (Figure 10). The total elapsed time of collecting metrics from all hosts is usually 2 ∼ 3× the average latency in TCP, resulting in low throughput. We analyze that TCP suffers from low efficient congestion control (CC) [20, 37] and the processing overhead of kernel stack [24]. Both factors incur large delay to concurrent RPC requests, resulting in variations of starting/ending time. On the contrary, ZERO eliminates the overhead of kernel stack via RDMA. The average latency and elapsed time of ZERO are nearly equivalent (Figures 10a and 10b). ZERO also achieves lower CPU compared with the baseline as shown in Figure 10d.



(a) Average latency vs. Hosts. (b) Controller CPU vs. Hosts.

Figure 12: Monitor performance with $8-128$ WRs $\times$ 4KB.

Credit-FC avoids triggering ECN/PFC. In Cluster1, we evaluate the ZERO performance w/ Credit-FC and BDP-FC, and w/o any flow control (FC). For tests w/o FC, the controller posts all requests in the beginning. The BDP-FC adopts a fixed credit of BDP (16KB with 25GbE bandwidth and $5 \sim 6\mu s$ RTT). Interestingly, PFC pause is not triggered until 256 QPs for ZERO w/o FC, due to the large dynamic buffer of switches [17] and the high threshold of PFC pause action (XOFF) [95]. Besides, the RNIC limits the maximum of outstanding READ requests to 16. Accordingly, the total credit is set to $8MB = 128 \times 64KB$. As shown in Figure 10a, both Credit-FC and BDP-FC effectively avoid PFC. The gap between ZERO and ideal is origin from the extra overhead in monitoring (Figure 10c), *e.g.*, virtualization, initializing and posting work requests (WRs) for each VM. The 25GbE bandwidth is still saturated with the $C_f \geq BDP$.

In Cluster2, the network has much higher BDP (96KB with 200GbE bandwidth and $4\mu s$ RTT) and the ECN is enabled by default. We observe a large fraction of ECN marked packets and high latency in BDP-FC (Figure 11a), because the ECN minimum threshold [95] is set to $\sim 1000KB$. Credit-FC still works with 512/1024 QPs due to the posting and arrival delay of READ requests with single thread (Section 4.3), where the build-up buffer should reach $2/4MB$ with the smallest quota of 4KB. We observe similar phenomenon with 1024 QPs in Cluster1. As shown in Figure 11b, Credit-FC only utilizes half of the 200 GbE bandwidth — because the credit is much smaller than the BDP of network to avoid congestion/ECN ($4 \sim 16$KB vs. 96KB) — the bottleneck lies in the switch buffer/threshold instead of end host. The throughput also degrades rapidly because large number of QPs share RNICs at both agent and controller.

We next zoom into the difference between Cluster1 and Cluster2. Cluster1 set much higher PFC threshold than the ECN threshold in Cluster2, resulting in larger credit to saturate the bandwidth. However, the tradeoff is that the pause duration occupies $\sim 99\%$ (Figure 10b) when PFC is triggered. As a comparison, ECN reacts to congestion and recovers traffic rapidly (Figure 11c). Another benefit of Credit-FC is reducing the CPU utilization for busy polling via avoiding network congestion as shown in Figures 10d and 11d.

Hybrid-I/O model is highly CPU efficient. We then evaluate the hybrid-I/O model in Cluster1. We post all requests in the beginning as the maximum outstanding READ requests is bounded to 16 by RNIC. As shown in Figure 12b, the CPU utilization of both epoll and polling increases with large data size, while epoll achieves lower CPU utilization. To highlight the gap, we set lower sampling interval to monitor 128 hosts with 256KB data. Results show that the CPU utilization of epoll and polling are $13.5/40.9\%$ and $20.2/72.6$ respectively with $100/10ms$ interval. However, the event-driven polling has high performance variations with small number of requests (Figure 12a), due to `epoll` syscall and thread blocking. With large-sized data, such overhead is averaged across many in-flight requests. We thus adopt epoll for general cases ($32 \sim 128$ requests) and polling for hosts with small-sized data (< 32 requests).

7 Experience and Future Work

In this section, we share our experience of building large-scale monitoring system using RDMA.

Achieving high scalability and availability. The scalability of RDMA-based distributed systems is limited by the number of QPs [26, 62, 82], which are cached in the limited SRAM of RNIC. Even several works [46, 47] adopt UD to reduce the number of QPs, ZERO uses READ to bypass the monitored host, which is only supported in RC mode. ZERO adopts QP sharing and grouping to remedy the QP constraints. First, ZERO agent manages all system/application metrics in a host sharing one QP connection. Second, ZERO controller monitors a group of QPs/hosts in each period, *e.g.*, 64 hosts with $1 \sim 10ms$ period, then switches to another group of QPs/hosts to avoid frequent QP evictions. To achieve high availability, each agent will build QP connections with at least three controllers in the practical deployment. Similar to most distributed systems, all these controllers run a consensus-based coordination service [40] to detect failures, and ZERO can switch to a standby controller seamlessly when the active controller is down.

Avoiding network interference. In the practical deployment, monitoring traffics co-exist with service traffics and inevitably impacts the network performance of services, due to the contentions at both agent and switches. Note that the controller has no such concerns with dedicated server for monitoring. Before ZERO, existing deployment adopts several mechanisms for traffic isolation, which inevitably brings other side effects. For example, a thorough solution is physically isolating traffics of services and monitoring with independent NICs and links [2]. However, physical isolation incurs large CapEx and is only suitable to high-priority services necessitating high SLA. Another solution is assigning a separate and lower-priority queue for monitoring traffics [35]. The persistent high loads of services may cause starvation of mon-

itoring traffics and losses of data in consecutive monitoring periods. Besides, ZERO built on RDMA is sensitive to such timeouts, which may cause QP state machine errors [72]. We thus abandon the traditional method of traffic isolation and resort to receiver-driven CC to avoid network interference.

ZERO provides a new perspective to mitigate network interference, *i.e.*, limiting the credit of monitoring traffics when co-existing. Specifically, the controller adopts group switching with 64 QPs/hosts in each group. The QPs of next group will be pre-fetched to RNIC SRAM for warming up. The total credit T in Credit-FC is set to 256KB, which maximally adds $\sim 20/10\mu s$ queuing delay with 100/200GbE bandwidth. Note that the maximum build-up queue is much less than 256KB due to the posting and arrival delay (Section 6.3). As a comparison, the traditional send-driven model easily causes network jitters with burst traffics. Besides, such settings have a negligible impact on monitoring QoS and single controller supports $1 \sim 10k$ hosts with $0.1 \sim 1$s intervals. The agent only occupies $0.01 \sim 0.1\%$ bandwidth of the monitored host.

Receiver-driven CC. Compared with existing sender-driven CC, the receiver-driven CC achieves several benefits in monitoring. Existing CC mechanisms, *e.g.*, DCQCN [95] and TIMELY [69], react to congestion after switch buffer/queue reach threshold. Besides, they aim to achieve equal bandwidth sharing across multiple flows, and cannot avoid interference between service traffics and monitoring traffics. As a comparison, the receiver-driven CC avoids network congestion and interference in advance by limiting the total in-flight data of monitoring. On the other hand, existing CC mechanisms, *e.g.*, DCQCN and HPCC [57], are complex in deployment and requires ECN or in-network telemetry (INT) supports from switches. However, ECN or INT capability are not always supported, *e.g.*, in Cluster1. The Credit-FC in our deployment is simple and effective to avoid triggering PFC/ECN.

We also observe several limitations of current Credit-FC. It only adopts credit without pacing [50] and cannot support massive concurrent flows with a 4KB transmission fragment. Besides, it is not a universal CC mechanism for data-center traffics, which dedicates to avoid network congestion and interference caused by monitoring traffics in the ZERO framework. In our future work, we will try to resolve these limitations from two aspects. First, we will consider both host bandwidth and ECN threshold [88] with a combination of credit- and pacing-based CC, to achieve full bandwidth utilization while avoiding network congestion. Second, we will explore the universal receiver-driven model in cloud networks, which has the benefits of CPU offloading via RDMA and more convenient CC [77].

8 Related Work

One-sided RDMA. In the system area, it is a trend to leverage one-sided RDMA operations to bypass the server CPUs. As pioneering works, Pilaf [68] enables the clients to directly read data from the server memory via RDMA read. Clients use CRC64 to search for the inconsistency of data caused by the possible read-write races on the server. RFP [79, 86] explores one-sided RDMA to provide another alternative solution for RPC, which uses RDMA read to fetch the response result. On the other hand, several works [46, 61, 86] explore how to optimize the raw performance of one-sided RDMA. To the best of our knowledge, ZERO is the first work to leverage one-sided RDMA for distributed monitoring.

Monitoring system. There are plenty of works targeting for the design of monitoring system [65]. Yet, all these works focus on data analytic [51, 80], tracing bugs [59, 63] and visualization [71]. Distinct from these works, where monitors are tightly coupled with the monitored applications and hosts, ZERO decouples the monitor from the monitored infrastructure and eliminates the monitoring overhead completely.

Cloud-native monitoring. Netdata [10] enables users to quickly identify and troubleshoot issues, and make data-driven decisions according to the pre-built visible dashboards. Prometheus [11] is an open-sourced monitoring system with complete ecosystem to extract time series data from the cloud-native applications, and it focuses on collecting metrics via a powerful query language called PromQL. Stackdriver [4] is the logging and monitoring solution of Google, which is integrated tightly into Google Cloud. Likewise, ZERO is deeply used in the cloud-native ecosystem of Alibaba cloud.

9 Conclusion

We propose the ZERO monitoring system framework, exploiting one-sided RDMA read for remote monitoring. ZERO achieves zero-overhead monitoring via the novel control plane and data plane. ZERO supports large-scale distributed monitoring via credit-based FC and hybrid I/O model. ZERO thus paves the way for integrating RDMA into the monitoring systems, which desires to benefit from the high performance of RDMA while avoiding poor scalability. We deploy ZERO in Alibaba cloud-native platform to evaluate its performance. The deployment results show that ZERO resolves interference problem of traditional monitor and easily fulfills both latency and throughput requirements in cloud-native monitoring.

10 Acknowledgment

We sincerely thank the anonymous shepherd and reviewers for their insightful comments and feedback. This work was supported in part by NSFC grant 62141220, 61972253, U1908212, 72061127001, 62172276, 61972254, the Program for Professor of Special Appointment (Eastern Scholar) at Shanghai Institutions of Higher Learning, Alibaba Innovative Research (AIR) Program. Corresponding author: Linghe Kong (linghe.kong@sjtu.edu.cn).

References

[1] Alibaba Cloud Container Service for Kubernetes. https://www.alibabacloud.com/en/product/kubernetes, Dec. 2020.

[2] Best Practices of ECS Container Network Multi-NIC Solution. https://www.alibabacloud.com/blog/593997, Dec. 2020.

[3] Extended Berkeley Packet Filter. https://ebpf.io/, Dec. 2020.

[4] Google cloud's operations suite (formerly stackdriver). https://cloud.google.com/products/operations, Dec. 2020.

[5] Grafana. https://grafana.com/, Dec. 2020.

[6] Hubble. https://github.com/cilium/hubble, Dec. 2020.

[7] InfluxDB. https://www.influxdata.com/, Dec. 2020.

[8] Install netdata on cloud providers. https://learn.netdata.cloud/docs/agent/packaging/installer/methods/cloud-providers, Dec. 2020.

[9] MaxCompute - Conduct large-scale data warehousing with MaxCompute. https://www.alibabacloud.com/product/maxcompute, Dec. 2020.

[10] Netdata - Monitor everything in real time for free with Netdata. http://www.netdata.cloud, Dec. 2020.

[11] Prometheus - Monitoring system & time series database. https://prometheus.io/, Dec. 2020.

[12] Redis. https://redis.io/, Dec. 2020.

[13] Specifying Attributes of Variables. https://gcc.gnu.org/onlinedocs/gcc-3.2/gcc/Variable-Attributes.html, Dec. 2020.

[14] ApsaraDB for Redis. https://www.alibabacloud.com/product/apsaradb-for-redis, June. 2021.

[15] Cloud-native applications management. https://www.alibabacloud.com/en/solutions/container, June. 2021.

[16] OFED performance test suite. https://github.com/linux-rdma/perftest, June. 2021.

[17] Packet buffer of switches. https://https://people.ucsc.edu/~warner/buffer.html, June. 2021.

[18] Giuseppe Aceto, Alessio Botta, Walter De Donato, and Antonio Pescapè. Cloud monitoring: A survey. *Computer Networks*, 57(9):2093–2115, 2013.

[19] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: Towards high-performance serverless computing. In *USENIX ATC*, 2018.

[20] 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.

[21] Nadav Amit and Michael Wei. The design and implementation of hyperupcalls. In *USENIX ATC*, 2018.

[22] Ashish Bijlani and Umakishore Ramachandran. Extension framework for file systems in user space. In *USENIX ATC*, 2019.

[23] Qingchao Cai, Wentian Guo, Hao Zhang, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, Yong Meng Teo, and Sheng Wang. Efficient distributed memory management with RDMA and caching. *Proceedings of the VLDB Endowment*, 11(11):1604–1617, 2018.

[24] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. Understanding host network stack overheads. In *ACM SIGCOMM*, 2021.

[25] Paul Castro, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. The rise of serverless computing. *Communications of the ACM*, 62(12):44–54, 2019.

[26] Youmin Chen, Youyou Lu, and Jiwu Shu. Scalable RDMA RPC on reliable connection with efficient resource sharing. In *EUROSYS*, 2019.

[27] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *SOSP*, 2017.

[28] Christina Delimitrou and Christos Kozyrakis. Hcloud: Resource-efficient provisioning in shared cloud systems. In *ASPLOS*, 2016.

[29] Henri Maxime Demoulin, Isaac Pedisich, Nikos Vasilakis, Vincent Liu, Boon Thau Loo, and Linh Thi Xuan Phan. Detecting asymmetric application-layer denial-of-service attacks in-flight with finelame. In *USENIX ATC*, 2019.

[30] Luca Deri, Samuele Sabella, and Simone Mainardi. Combining system visibility and security using eBPF. In *ITASEC*, 2019.

[31] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. Farm: Fast remote memory. In *USENIX NSDI*, 2014.

[32] Yihui Feng, Zhi Liu, Yunjian Zhao, Tatiana Jin, Yidi Wu, Yang Zhang, James Cheng, Chao Li, and Tao Guan. Scaling large production clusters with partitioned synchronization. In *USENIX ATC*, 2021.

[33] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud and edge systems. In *ASPLOS*, 2019.

[34] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices. In *ASPLOS*, 2019.

[35] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. RDMA over commodity ethernet at scale. In *ACM SIGCOMM*, 2016.

[36] Jing Guo, Zihao Chang, Sa Wang, Haiyang Ding, Yihui Feng, Liang Mao, and Yungang Bao. Who limits the resource efficiency of my datacenter: An analysis of alibaba datacenter traces. In *IEEE/ACM IWQoS*, 2019.

[37] Sangtae Ha, Injong Rhee, and Lisong Xu. Cubic: a new TCP-friendly high-speed TCP variant. *ACM SIGOPS operating systems review*, 42(5):64–74, 2008.

[38] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. The eXpress data path: Fast programmable packet processing in the operating system kernel. In *ACM CONEXT*, 2018.

[39] Michio Honda, Felipe Huici, Costin Raiciu, Joao Araujo, and Luigi Rizzo. Rekindling network protocol innovation with user-level stacks. *ACM SIGCOMM Computer Communication Review*, 44(2):52–58, 2014.

[40] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX ATC*, volume 8, 2010.

[41] Seyyed Ahmad Javadi, Amoghavarsha Suresh, Muhammad Wajahat, and Anshul Gandhi. Scavenger: A black-box batch workload resource manager for improving utilization in cloud environments. In *ACM SOCC*, 2019.

[42] Congfeng Jiang, Yitao Qiu, Weisong Shi, Zhefeng Ge, Jiwei Wang, Shenglei Chen, Christophe Cerin, Zujie Ren, Guoyao Xu, and Jiangbin Lin. Characterizing co-located workloads in Alibaba cloud datacenters. *IEEE Transactions on Cloud Computing*, 2020.

[43] Anshul Jindal, Vladimir Podolskiy, and Michael Gerndt. Performance modeling for cloud microservice applications. In *ACM/SPEC ICPE*, 2019.

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

[45] Anuj Kalia, Michael Kaminsky, and David G Andersen. Using RDMA efficiently for key-value services. In *ACM SIGCOMM*, 2014.

[46] Anuj Kalia, Michael Kaminsky, and David G Andersen. Design guidelines for high performance RDMA systems. In *USENIX ATC*, 2016.

[47] Anuj Kalia, Michael Kaminsky, and David G Andersen. Fasst: Fast, scalable and simple distributed transactions with two-sided RDMA datagram RPCs. In *USENIX OSDI*, 2016.

[48] Junaid Khalid, Eric Rozner, Wesley Felter, Cong Xu, Karthick Rajamani, Alexandre Ferreira, and Aditya Akella. Iron: Isolating network-based {CPU} in container environments. In *USENIX NSDI*, 2018.

[49] Ricardo Koller and Dan Williams. Will serverless end the dominance of linux in the cloud? In *ACM HOSTOS*, 2017.

[50] 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.

[51] Mahendra Kutare, Greg Eisenhauer, Chengwei Wang, Karsten Schwan, Vanish Talwar, and Matthew Wolf. Monalytics: online monitoring and analytics for managing large scale data centers. In *International Conference on Autonomic Computing*, 2010.

[52] Sándor Laki, Gergő Gombos, Péter Hudoba, Szilveszter Nádas, Zoltán Kiss, Gergely Pongrácz, and Csaba Keszei. Scalable per subscriber QoS with core-stateless scheduling. *ACM SIGCOMM Demo*, 1:84–86, 2018.

[53] Sándor Laki, Dániel Horpácsi, Péter Vörös, Róbert Kitlei, Dániel Leskó, and Máté Tejfel. High speed packet forwarding compiled from protocol independent data plane specifications. In *ACM SIGCOMM*, 2016.

[54] Joshua Levin. Viperprobe: Using eBPF metrics to improve microservice observability, 2020.

[55] Bojie Li, Tianyi Cui, Zibo Wang, Wei Bai, and Lintao Zhang. Socksdirect: Datacenter sockets can be fast and compatible. In *ACM SIGCOMM*, 2019.

[56] Feifei Li. Cloud-native database systems at Alibaba: Opportunities and challenges. *Proceedings of the VLDB Endowment*, 12(12):2263–2272, 2019.

[57] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, et al. HPCC: High precision congestion control. In *ACM SIGCOMM*. 2019.

[58] Qixiao Liu and Zhibin Yu. The elasticity and plasticity in semi-containerized co-locating cloud workload: a view from alibaba trace. In *ACM SOCC*, 2018.

[59] Chang Lou, Peng Huang, and Scott Smith. Understanding, detecting and localizing partial failures in large system software. In *USENIX NSDI*, 2020.

[60] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. Characterizing microservice dependency and performance: Alibaba trace analysis. In *ACM SoCC*, 2021.

[61] Teng Ma, Kang Chen, Shaonan Ma, Zhuo Song, and Yongwei Wu. Thinking more about RDMA memory semantics. In *IEEE International Conference on Cluster Computing (CLUSTER)*, 2021.

[62] Teng Ma, Tao Ma, Zhuo Song, Jingxuan Li, Huaixin Chang, Kang Chen, Hai Jiang, and Yongwei Wu. X-RDMA: Effective RDMA middleware in large-scale production environments. In *IEEE International Conference on Cluster Computing (CLUSTER)*, 2019.

[63] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. Pivot tracing: Dynamic causal monitoring for distributed systems. In *SOSP*, 2015.

[64] Ilias Marinos, Robert NM Watson, and Mark Handley. Network stack specialization for performance. *ACM SIGCOMM Computer Communication Review*, 44(4):175–186, 2014.

[65] Matthew L Massie, Brent N Chun, and David E Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817–840, 2004.

[66] Steven McCanne and Van Jacobson. The BSD Packet Filter: A new architecture for user-level packet capture. In *USENIX winter*, 1993.

[67] Bradley Miles and Dave Cliff. A cloud-native globally distributed financial exchange simulator for studying real-world trading-latency issues at planetary scale. *arXiv preprint arXiv:1909.12926*, 2019.

[68] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using one-sided RDMA reads to build a fast, cpu-efficient key-value store. In *USENIX ATC*, 2013.

[69] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. TIMELY: RTT-based congestion control for the datacenter. In *ACM SIGCOMM*, 2015.

[70] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. Revisiting network support for RDMA. In *ACM SIGCOMM*, 2018.

[71] Alex Page, Tolga Soyata, Jean-Philippe Couderc, Mehmet Aktas, Burak Kantarci, and Silvana Andreescu. Visualization of health monitoring data acquired from distributed sensors for multiple patients. In *IEEE GLOBECOM*, 2015.

[72] Waleed Reda, Marco Canini, Dejan Kostić, and Simon Peter. {RDMA} is turing complete, we just did not know it yet! In *USENIX NSDI*, 2022.

[73] Jamal Hadi Salim, Robert Olsson, and Alexey Kuznetsov. Beyond softnet. In *5th Annual Linux Showcase & Conference*, 2001.

[74] Hermann Schweizer, Maciej Besta, and Torsten Hoefler. Evaluating the cost of atomic operations on modern architectures. In *IEEE PACT*, 2015.

[75] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. Legoos: A disseminated, distributed os for hardware resource disaggregation. In *USENIX OSDI*, 2018.

[76] Zhiming Shen, Zhen Sun, Gur-Eyal Sela, Eugene Bagdasaryan, Christina Delimitrou, Robbert Van Renesse, and Hakim Weatherspoon. X-containers: Breaking down barriers to improve performance and isolation of cloud-native containers. In *ASPLOS*, 2019.

[77] Arjun Singhvi, Aditya Akella, Dan Gibson, Thomas F Wenisch, Monica Wong-Chan, Sean Clark, Milo MK Martin, Moray McLaren, Prashant Chandra, Rob Cauble, et al. 1RMA: Re-Envisioning Remote Memory Access for Multi-Tenant Datacenters. In *ACM SIGCOMM*, 2020.

[78] Akshitha Sriraman, Abhishek Dhanotia, and Thomas F Wenisch. Softsku: optimizing server architectures for microservice diversity@ scale. In *ISCA*, 2019.

[79] Maomeng Su, Mingxing Zhang, Kang Chen, Zhenyu Guo, and Yongwei Wu. RFP: When RPC is faster than server-bypass with RDMA. In *EUROSYS*, 2017.

[80] Jörg Thalheim, Antonio Rodrigues, Istemi Ekin Akkus, Pramod Bhatotia, Ruichuan Chen, Bimal Viswanath, Lei Jiao, and Christof Fetzer. Sieve: actionable insights from monitored metrics in distributed systems. In *ACM/IFIP/USENIX Middleware Conference*, 2017.

[81] Linus Torvalds. The linux edge. *Communications of the ACM*, 42(4):38–39, 1999.

[82] Shin-Yeh Tsai and Yiying Zhang. Lite kernel RDMA support for datacenter applications. In *SOSP*, 2017.

[83] Marcos AM Vieira, Matheus S Castanho, Racyus DG Pacífico, Elerson RS Santos, Eduardo PM Câmara Júnior, and Luiz FM Vieira. Fast packet processing with eBPF and XDP: Concepts, code, challenges, and applications. *ACM Computing Surveys*, 53(1):1–36, 2020.

[84] Hanzhang Wang, Phuong Nguyen, Jun Li, Selcuk Kopru, Gene Zhang, Sanjeev Katariya, and Sami Ben-Romdhane. Grano: Interactive graph-based root cause analysis for cloud-native distributed data platform. *Proceedings of the VLDB Endowment*, 12(12):1942–1945, 2019.

[85] Haitao Wu, Zhenqian Feng, Chuanxiong Guo, and Yongguang Zhang. ICTCP: Incast congestion control for TCP in data-center networks. *IEEE/ACM transactions on networking*, 21(2):345–358, 2012.

[86] Yongwei Wu, Teng Ma, Maomeng Su, Mingxing Zhang, Kang Chen, and Zhenyu Guo. RF-RPC: Remote fetching RPC paradigm for RDMA-enabled network. *IEEE Transactions on Parallel and Distributed Systems*, 30(7):1657–1671, 2018.

[87] Xin Xu and Bhavesh Davda. A hypervisor approach to enable live migration with passthrough sr-iov network devices. *ACM SIGOPS Operating Systems Review*, 51(1):15–23, 2017.

[88] Siyu Yan, Xiaoliang Wang, Xiaolong Zheng, Yinben Xia, Derui Liu, and Weishan Deng. ACC: Automatic ECN tuning for high-speed datacenter networks. In *ACM SIGCOMM*, 2021.

[89] Jian Yang, Joseph Izraelevitz, and Steven Swanson. Filemr: Rethinking RDMA networking for scalable persistent memory. In *USENIX NSDI*, 2020.

[90] Kenichi Yasukata, Michio Honda, Douglas Santry, and Lars Eggert. StackMap: Low-latency networking with the OS stack and dedicated NICs. In *USENIX NSDI*, 2016.

[91] Chaoqun Zhan, Maomeng Su, Chuangxian Wei, Xiaoqiang Peng, Liang Lin, Sheng Wang, Zhe Chen, Feifei Li, Yue Pan, Fang Zheng, et al. Analyticdb: Real-time olap database system at alibaba cloud. *Proceedings of the VLDB Endowment*, 12(12):2059–2070, 2019.

[92] Xiantao Zhang, Xiao Zheng, Zhi Wang, Hang Yang, Yibin Shen, and Xin Long. High-density multi-tenant bare-metal cloud. In *ASPLOS*, 2020.

[93] Zhuo Zhang, Chao Li, Yangyu Tao, Renyu Yang, Hong Tang, and Jie Xu. Fuxi: a fault-tolerant resource management and job scheduling system at internet scale. In *Proceedings of the VLDB Endowment*, volume 7, pages 1393–1404, 2014.

[94] Hao Zhou, Ming Chen, Qian Lin, Yong Wang, Xiaobin She, Sifan Liu, Rui Gu, Beng Chin Ooi, and Junfeng Yang. Overload control for scaling wechat microservices. In *ACM SOCC*, 2018.

[95] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale RDMA deployments. In *ACM SIGCOMM*, 2015.