A Case For Cross-Domain Observability to Debug Performance Issues in Microservices

Ranjitha K[1] Praveen Tammana[1] Pravein Govindan Kannan[2] Priyanka Naik[2]
[1]IIT Hyderabad, India [2]IBM Research - India

***Abstract*—Many applications deployed in the cloud are usually refactored into small components called microservices that are deployed as containers in a Kubernetes environment. Such applications are deployed on a cluster of physical servers which are connected via the datacenter network.**

In such deployments, resources such as compute, memory, and network, are shared and hence some microservices (culprits) can misbehave and consume more resources. This interference among applications hosted on the same node leads to performance issues (e.g., high latency, packet loss) in the microservices (victims) followed by a delayed or low-quality response. Given the highly distributed and transient nature of the workloads, it's extremely challenging to debug performance issues. Especially, given the nature of existing monitoring tools, which collect traces and analyze them at individual points (network, host, etc) in a disaggregated manner.

In this paper, we argue toward a case for a *cross-domain* (network & host) monitoring and debugging framework which could provide the end-to-end observability to debug performance issues of applications and pin-point the root-cause whether it is on the sender-host, receiver-host or the network. We present the design and provide preliminary implementation details using eBPF (extended Berkeley Packet Filter) to elucidate the feasibility of the system.

I. Introduction

With the increase in the number of cloud-based applications and their stringent service level objectives (SLOs), it is important for cloud providers to debug performance problems that violate SLAs in a timely manner. The cloud-based applications have both internal and public-facing applications with stringent time deadlines (order of milliseconds) to serve the user requests. A user request is usually handled by a front-end microservice which communicates with several other microservices recursively using Remote Procedure Call (RPCs). A latency increase in a few RPC calls could have a compounding effect on the associated microservices , resulting in SLA violation for a large number of user requests.

In a cluster managed by Kubernetes orchestrator, an application's microservice (pod) would run as a container on any physical server/ VM (node) and communicate with other microservices deployed at other servers via a data center network. It is extremely difficult to debug the cause of poor performance (*e.g.*, high latency) of RPCs, especially when sporadic. This is because of highly distributed environment where multiple entities are to blame for poor performance.

More specifically, network stalls [12] on Kubernetes cluster are common but extremely hard to debug because of their sporadic nature. One has to suspect several entities in a container-based underlay network such as iptables configuration, load balancer, IP-in-IP tunnel abstractions, host kernel, NICs, and also various events in an IP-in-IP based data center overlay network such as congestion, link failures, routing loops, blackholes. This is followed by investigating deep into one or more entities and events to find the cause: Is the host CPU core frequently interrupted by other workloads or background processes, thus causing sporadic latencies? Are the packets queued in the host NIC for a longer time? Are there many packets contending for a shared network link (microburst or TCP incast)? etc.

Debugging such performance issues requires end-to-end visibility across entities: it starts with collection of traces at multiple points in the call flow followed by pruning, correlation, and dependency analysis. Towards this direction, our approach is to instrument the host kernel network stack and collect traces essential for detecting and debugging such network-level and host-level problems in a timely manner.

However, to realize this idea, there are mainly two challenges. First, as the current 100/400 Gbps NICs are capable of pushing tens of millions of packets to CPU cores, the overheads (*e.g.*, CPU cycles, memory) due to instrumentation should have minimal-to-no impact on packet-processing throughput, while at the same time provide fine-grained packet-level visibility to enable a wide-range of debugging problems. The second challenge is related to getting end-to-end visibility: as orchestrators use packet-level abstractions such as proxies, service-mesh, and VXLAN overlays, packets being sent by a container-based application and that being observed by the data-center network (collected using Netflow [3]/INT [2], etc) are different. Because of this difference, debugging becomes either difficult or infeasible to map the set of applications impacted due to a network-level problem.

To address the first challenge, we design a two-phase debugging system. In the first phase, we maintain per-connection performance metrics (*e.g.*, ongoing RTT), analyze the metrics, and detect problematic connections (*e.g.*, high latency). In the second phase, we collect fine-grained traces of only the problematic connections by enabling event tracing at different layers in the host kernel network stack. To address the second challenge, we implement a mapping primitive that maintains mapping of application packet headers to datacenter network headers (*e.g.*, IP space). Finally, the collected traces and the mapping data can be further analyzed at a cluster level and pin-point the set of application microservices to be blamed for the poor performance of victim microservices affected by them.

As a first step, in this paper, we tackle the first challenge to implement a monitoring tool using eBPF [1] to monitor performance problems such as per-flow RTT. The monitoring tool attaches to the container's virtual interface and detects performance problems. In the evaluation, we demonstrate that such metrics could be passively monitored with a minimal overhead in kernel compared to existing techniques.

II. Related work

The key limitation of the existing techniques is the disaggregated approach of collecting traces and analyzing them at individual points. Hence, the onus is on the operator to aggregate information across the stack. With increasing scale and complexity of data-centers, aggregating information and performing root-cause-analysis is extremely slow, inaccurate and misleading.

On one hand, we have network-level systems which leverage control-plane and data-plane programmability to monitor performance metrics of connections in the network, detect problematic connections, and debug the cause. Recently, several approaches such as per-packet postcards [11], query-driven telemetry systems [17], [10], [15], [13], performance-based diagnosis systems [8], [6], and in-network telemetry systems [19], [20] have been proposed. These systems work at the network-level with the packets from the end-host nodes. However, popular orchestrators such as Kubernetes, Openstack, etc usually perform network abstractions (service-mesh, proxies,etc.) which deploy tunneling/NAT mechanisms to provide applications a seamless way to use private addressing schemes. Thus, Flow-IDs (5-tuple) of a packet header captured at the network-level cannot be directly used to identify the application/container running at the node, making it hard to pinpoint victim application(s) and culprit application(s) at run-time. Additionally, since these systems perform diagnosis on the traces collected at network level, they do not have necessary visibility to debug problems at the host level.

On the other hand, there are systems that collect per-connection performance metrics at the host-level [9], [16], [14], [4]. However, these systems do not provide necessary fine-grained visibility at the container network/data center network to determine whether the problem is at the host-level (container network) or at the network-level.

To provide fine-grained visibility at the host, One naive solution to obtain fine-grained host-level packet data would be to attach packet-level probes using PCAP or AF_PACKET similar to skydive [4] or use a combination of tcpdump/tcptrace [5]. However, this has significant overhead in terms of storage, CPU and throughput. A recent work proposed an eBPF based system called Host-In band Network telemetry (INT) [18]. It could be configured to collect records for every packet at every host and to export records to a central database. However, with the current scale of data centers (several Tbps, and several billions of packets per second), host-INT cannot be easily deployed and used to debug performance issues at fine granularity due to its high collection, processing, and storage overheads. Additionally, due to NAT and packet abstractions at the container network, it cannot correlate flows with network INT records to provide end-to-end observability.

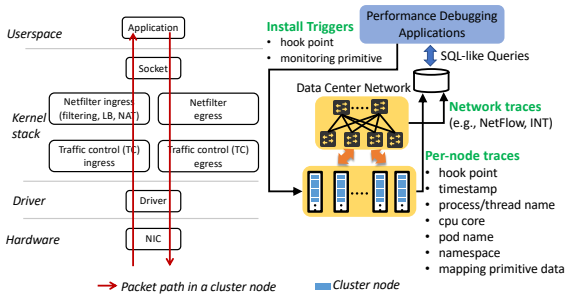


Fig. 1: System workflow

III. Design

Our design instruments the host kernel network stack to collect performance metrics of each connection and event traces of only problematic connections at different layers in the host kernel stack. The metrics would enable faster detection of poor performance whereas the traces enable fine-grained visibility and fast debugging. To realize this, we design two in-kernel light-weight primitives and a tracer. Figure 1 shows the overall system workflow with packet path (with hook points) within the host at the left.

Monitoring primitive. In the first phase, this in-kernel primitive processes packet headers and maintains performance metrics of every connection. To be specific, the metrics include average network RTT, application latency, and throughput, etc. These metrics are computed by processing packet headers of both incoming and outgoing packets. The headers include source ip, destination ip, source port, destination port, protocol, tcp seq no, tcp ack, and packet timestamp, etc. The metrics are exposed to an user space process that periodically collects (pull model) them and identifies troubled connections violating application SLAs. As an alternative, one can install a trigger which sends an alert (push model) to the user space process whenever the performance metrics exceed pre-defined thresholds. This will potentially reduce data collection and processing overheads in the user space.

Tracer. In the second phase, we enable event tracing only for poorly performing applications (pod) at different layers in the host kernel network stack at all nodes where application is deployed. Compared to the approach that collects per-packet traces of all applications at all nodes, our approach selectively traces a small subset of applications, thus significantly reduces storage and processing overheads and scales well. The traces include layer name, time stamp, process/thread name, cpu core processing application traffic, etc., and the layers include socket-level, TC-level, driver, and NIC hardware. The traces at host-level can be further analyzed to identify cross layer events causing the observed performance drop.

Mapping primitive. Similar to the trace collection at host-level, traces can be collected at network-level using well known techniques like Netflow [3], INT [2], etc and pin-point the location (Host vs. Network). However, IPs in application

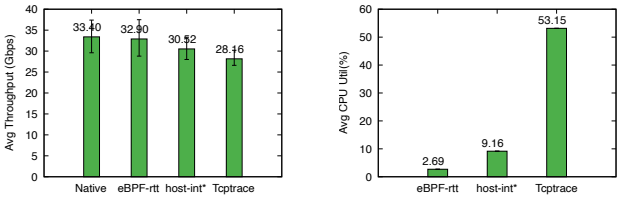
(pod) space are different from the underlying physical network IP space. This is because cloud-native orchestrators employ IP-in-IP encapsulation or/and NAT which map packets in pod space to cluster space and vice-versa. If traces are collected in the network, then they miss the application (pod) context, thus making it either hard or impossible to debug network events (*e.g.*, congestion). For example consider culprit pods' traffic (*e.g.*, heavy flows) collide with victim pods' traffic (*e.g.*, mice flows). If such congestion events are transient (*e.g.*, microburst, TCP incast) in the network, without application context, we cannot pin-point the set of culprit application pods and take some action to fix the problem. To address this problem, we design a mapping primitive that maintains application context (packet headers) to network context mapping state in the kernel (virtual and external interfaces) and exposes this translation to user space. The mapping state retrieved at small timescales would enable identification of application traffic (tenant's private IPs) causing performance problems (*e.g.*, congestion) at the network-level.

IV. Implementation & Evaluation

We implement a prototype for *monitoring primitive* using eBPF [1]. The monitoring primitive is configured to perform continuous per-flow RTT (Round-Trip Time) monitoring which is one of the key indicators of an application's performance [7]. We compute RTT in eBPF by maintaining the timestamp of egress TCP packets using (Flow-ID (5-tuple), Sequence) in an LRU (Least Recently Used) hash table (supported natively in eBPF), and calculate the RTT when the corresponding ACK packet is received. Finally, a moving average of RTT is maintained in a separate hash table per-flow. This RTT value can be used as an indicator to trigger perform tracing and debugging. We attach the RTT-monitoring eBPF program to *ingress* and *egress* TC hook of the interfaces.

We evaluate the continuous RTT-monitoring on a physical testbed consisting of a pair of bare-metal servers connected using 40G links. We start multiple parallel TCP flows using iperf3 to saturate the link. We measure the average throughput over 5 separate runs for 100 seconds each. We plot the throughput achieved with continuous RTT-monitoring by attaching the eBPF program (eBPF-rtt) to the tc hook compared to the native throughput. Additionally, we compare against host-INT[1] and with TCPdump/TCPtrace in Figure 2.a. We do not observe any significant degradation in throughput of the eBPF-based RTT monitoring primitive compared to native throughput. However, we observe upto 10% and 20% throughput drop when RTTs are computed using host-INT and tcpdump/tcptrace respectively. We measure the CPU overhead incurred using *mpstat* every second throughout the experiment and compare against tcpdump/tcptrace approach to compute RTT. In Figure 2.b, eBPF-based RTT observes only 2.69% increase in CPU utilization compared to host-INT and tcpdump/trace which observes an increase of 9.16% and 53.15% respectively.

[1]RTT is computed at user-space with the INT reports which are generated and sent using a ring-buffer for each packet



a. Comparison Average throughput

b. Comparison of CPU overhead

Fig. 2: Comparison of overheads of RTT monitoring

V. Conclusion & Future Work

Given the widespread adoption of microservice based cloud deployments, we put forward a case to build cross-domain observability framework to debug performance issues of such dynamic applications. We present the design of the framework, and demonstrate the feasibility by implementing one of the primitives of the framework using eBPF.

In future, we plan to implement a trigger mechanism based on the increase in moving average of RTT. This trigger would then perform eBPF-based event-tracing based on the physical IP by implementing the *Mapping primitive* across the cluster. This will provide the ability to observe the latency in pod, network, and host-space to narrow down the area of interest.

References

[1] eBPF. https://ebpf.io/.
[2] In-band Network Telemetry. https://p4.org/assets/INT-current-spec.pdf.
[3] NetFlow. https://en.wikipedia.org/wiki/NetFlow.
[4] Skydive Real-time network analyzer. http://skydive.network/index.html.
[5] Tcpdump/tcptrace. https://fasterdata.es.net/performance-testing/network-troubleshooting-tools/tcpdump-tcptrace.
[6] Closed-loop network performance monitoring and diagnosis with SpiderMon. In *NSDI 22*, 2022.
[7] X. Chen, H. Kim, J. M. Aman, W. Chang, M. Lee, and J. Rexford. Measuring tcp round-trip time in the data plane. In *SPIN*, 2020.
[8] R. J. et al. Burstradar: Practical real-time microburst monitoring for datacenter networks. In *APSys*, 2018.
[9] M. Ghasemi, T. Benson, and J. Rexford. Dapper: Data plane performance diagnosis of tcp. In *SOSR*, 2017.
[10] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger. Sonata: Query-driven streaming network telemetry. In *SIGCOMM*, 2018.
[11] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *NSDI*, 2014.
[12] T. Julienne. Debugging network stalls on Kubernetes. https://github.blog/2019-11-21-debugging-network-stalls-on-kubernetes/.
[13] P. G. Kannan, N. Budhdev, R. Joshi, and M. C. Chan. Debugging transient faults in data centers using synchronized network-wide packet histories. In *NSDI 21*, 2021.
[14] Y. Li, R. Miao, M. Alizadeh, and M. Yu. DETER: Deterministic TCP replay for performance diagnosis. In *NSDI*, 2019.
[15] Y. Li, R. Miao, C. Kim, and M. Yu. FlowRadar: A better NetFlow for data centers. In *NSDI*, 2016.
[16] M. Moshref, M. Yu, R. Govindan, and A. Vahdat. Trumpet: Timely and precise triggers in data centers. In *SIGCOMM*, 2016.
[17] S. e. a. Narayana. Language-directed hardware design for network performance monitoring. In *SIGCOMM*, 2017.
[18] T. Osiński and C. Cascone. Achieving end-to-end network visibility with host-int. In *ANCS*, 2021.
[19] P. Tammana, R. Agarwal, and M. Lee. Simplifying datacenter network debugging with pathdump. In *OSDI*, 2016.
[20] P. Tammana, R. Agarwal, and M. Lee. Distributed network monitoring and debugging with switchpointer. In *NSDI*, 2018.