Designing a Lightweight Network Observability agent for Cloud Applications

Pravein Govindan Kannan[1] Shachee Mishra Gupta[1] Dushyant Behl[1] Eran Raichstein[1] Joel Takvorian[2]

[1] IBM Research
[2] Red Hat

Abstract. Applications are increasingly being deployed on the cloud as microservices using orchestrators like Kubernetes. With microservices-type deployment, performance and observability are critical requirements, especially given the SLAs and strict business guarantee requirements (latency, throughput, etc) of requests. Network observability is an imperative feature that every orchestrator needs to incorporate to provide the operators visibility into the network communication between the services deployed and the ability to provide necessary metrics to diagnose problems.

In this paper, we propose a lightweight network observability agent *netobserv-ebpf-agent*[3] built using eBPF, that can be deployed in various environments (K8s, Bare-metal, etc) and runs independent of the underlying network datapath/ Container Network Interfaces (CNIs) deployed by the orchestrator. *netobserv-ebpf-agent* monitors the network traffic in each host-nodes' interfaces running in the cluster and summarizes the necessary information of the traffic workloads with very minimal overhead. We articulate the design decisions of *netobserv-ebpf-agent* using measurements which maximize the performance of the datapath. Our evaluations show that *netobserv-ebpf-agent* offers significant performance benefits against the existing systems, while keeping the CPU and memory overheads lower by a magnitude. *netobserv-ebpf-agent* is available in open source and is officially released as part of *Red Hat OpenShift* Container Platform.

1 Introduction

With the growing cloud deployment of applications using microservices [30] running on container orchestration platforms like Kubernetes, more focus has been recently on observability. Observability is the ability to know and interpret the current state of the deployment, and technology to realize if something is amiss. With several applications demanding strict communication guarantees [31] i.e., SLAs in terms of down times, latency, throughput, etc, network-level observability is a highly imperative feature that needs to be provided by the orchestration platforms either natively or using third-party plugins/operators. Network observability additionally helps operators to tune network provisioning based on the application workloads, detecting heavy workloads and additionally provide

[3] https://github.com/netobserv/netobserv-ebpf-agent

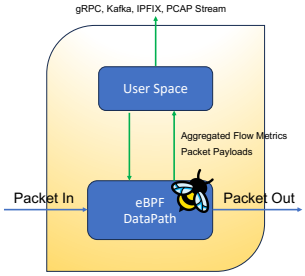


Fig. 1: The architecture of *netobserv-ebpf-agent*

application-level insights (such as percentage of web traffic [28, 1], video streaming [27], etc).

The core part of every network observability tool is the ability to monitor and collect network metrics in a transparent manner. The network metrics typically provide insights like the number of packets/bytes exchanged between two IP endpoints [14]. Traditionally, host-based tools like libpcap-based pmacct citepmacct, and Tcpdump [22] are used. OpenVSwitch (OVS) [29] also has been used to expose information as sFlow or NetFlow records [17] in deployments where the Container Network Interfaces (CNI) are OVS-based [18].

Recently eBPF [8] (extended Berkeley Packet Filter) has emerged as a popular technology to implement various networking features like monitoring, filtering, etc at the end-hosts kernel. eBPF enables custom programs to be run along with the kernel in a safe and sandboxed manner. The major advantages of using eBPF are its performance, programmability, and flexibility. It allows programs to be hooked at certain points along the network data path like socket system calls, Traffic Control (TC) and Express Data Path (XDP [26]) to capture network traffic and insights at different vantage points irrespective of the underlying datapath/CNI. eBPF uses data structures called maps which allow the kernel programs to share data with user-space applications.

Recently, several open source tools [4, 5, 2, 16, 13] use eBPF for network observability, by storing network metrics in different eBPF maps, which is a fundamental data structure to store state information in eBPF datapath. However, so far there has been no systematic study on how the type of eBPF maps used affect the performance of the observability tool and end-application traffic.

In this paper, we propose *netobserv-ebpf-agent*, an eBPF-based traffic monitoring tool that we developed to enable Network observability in orchestrators such as Kubernetes. *Netobserv-ebpf-agent* hooks onto the Traffic Control (TC) of the interfaces at the end-hosts (worker nodes) to monitor the network traffic at the flow-level and exports aggregate statistics using gRPC/Kafka/IPFIX (refer Figure 1). The main primitive of network observability is the monitoring of the necessary packet/flow-level information of the ongoing network packets in the data-path. The type of eBPF maps which maintains the records in the data-path is quite important, since it is accessed on a per-packet basis. Using benchmarks, we study how the eBPF maps used in the eBPF monitoring datapath impacts the performance of the observed traffic as well as affect the overheads. We apply

the learnings from the benchmarks to design *netobserv-ebpf-agent* based on a novel architecture using Per-CPU hash maps where the data-structure is maintained per-core. The monitoring data is scattered across different hash buckets in the data-path and is aggregated in the user space.

We implement the *netobserv-ebpf-agent* using Cilium Go [4] eBPF library and evaluate our system with realistic traffic at line-rate and observe that *netobserv-ebpf-agent* can perform better by a magnitude compared against state-of-the-art monitoring systems used in production, while consuming very little overhead in terms of CPU and memory. Finally, we present the deployment model of *netobserv-ebpf-agent* in Kubernetes with enrichment of network metrics with contextual information like namespace, pod names and topology. *netobserv-ebpf-agent* is available open-source and also released along with *Red Hat OpenShift* and its compatible with Kubernetes regardless of the underlying networking datapaths.

2 Background & Related Work

In this section, we give a brief background on network observability with eBPF and enumerate the existing open-source tools available for observability.

eBPF [8] provides a way to run sandboxed programs in the OS kernel and safely extend the capabilities of the kernel. eBPF programs can be attached to several hook points like system calls, socket APIs, Traffic Control (TC), XDP, etc to implement features in security, networking, profiling and observability. eBPF uses datastructure called maps to store & exchange information with the userspace. Several types of eBPF maps exists while the popular types used for network traffic analysis being hash map and ring buffer (as explained in Section 3). In the context of network observability, maps are popularly used to store network metrics (flow-id, packet counts, etc) which the userspace program would pick up and export it to a collector in the necessary format. Over the recent years, driven by adoption of cloud (predominantly running linux [23]), several open source tools have emerged to achieve network observability.

Libebpfflow [16] is an eBPF-based network visibility library. It hooks on to various points in the host stack like kernel probes (e.g. inet_csk_accept) and tracepoints (e.g. net:net_dev_queue) to analyze TCP/UDP traffic states. Its eBPF implementation uses perf event buffer (or ringbuffer) to notify TCP state change events to the user-space.

Cloudflare's ebpf exporter [7] provides APIs for plugging in custom ebpf code to record custom metrics of interest. Pixie [20] uses bpftrace to trace syscalls for application profiling and protocol tracing (TLS). It listens to TCP/UDP state messages to collect necessary information and stores them for querying. Inspektor [12] provides a collection of tools for Kubernetes cluster debugging, added as a daemonset on each node of the cluster to collect traces. These events are written to a ring buffer, which is consumed retrospectively when a fault occurs (for e.g. upon a pod crash).

L3AF [15] provides a set of eBPF packages which can be packaged and chained together using tail-calls. It provides network observability by storing flow records on a hash map in the eBPF datapath to be exported as IPFIX.

Host-INT [13] extends the in-band Network Telemetry [11] to support telemetry for host network stack. Fundamentally, INT embeds the switching delay incurred for each packet into an INT header in the packet. Host-INT does the same for the host network stack and uses a ring buffer to send statistics to the userspace. Skydive [21]. Skydive is a network topology and flow analyzer. It attaches probes to the nodes to collect flow-level information. The probes are attached using PCAP, AF_Packet, OpenVSwitch, etc. Recently, Skydive uses eBPF-based hash maps to capture the flow metrics.

Cilium Tetragon [5] is a recently released extensible framework for security and observability in Cilium. Tetragon stores the recorded metrics in user space using ring buffers. Additionally, eBPF is leveraged to enforce policy spanning various kernel components such as virtual file system (VFS), namespace, system call.

While different tools adopted different data structures to collect and export per-packet metrics, there are no existing performance measurements to study the impact of these data-structure used in the eBPF data-path. We start by first bridging that gap to study how the choice of eBPF maps in the monitoring data-path affects the performance. *netobserv-ebpf-agent* uses the insights from the measurements to use the optimal data-structure to maximize performance of the eBPF data-path.

3 Design & Implementation

3.1 Revisiting the monitoring data-path

In this section, we benchmark the basic primitive of a monitoring data-path which extracts flow-level information per packet (standard 5-tuple {source IP address, destination IP address, source port, destination port, protocol}) and maintains per-flow counters for number of bytes and packets. We implement template eBPF programs (attachable to TC hook-point) using different eBPF maps [24] to collect the same metrics from host traffic. The artefacts of our measurements are open source[4]. We use the following eBPF maps, which are popularly used by existing open source tools to collect and store packet-level and flow-level metrics:

- **Ring Buffer** : Ring buffer is a shared queue between the eBPF datapath and the userspace, where eBPF datapath is the producer and the userspace program is the consumers.
- **Array & Per-CPU Array** : (Per-CPU) Array-based map could be used to store per-packet metrics temporarily in the eBPF data-path before sending to user space.
- **Hash & Per-CPU Hash** : (Per-CPU) Hash map could be used in the eBPF datapath by first performing a hash of the 5-tuple flow-id which would point to a specific bucket in the array. Thus, it is possible to aggregate per-flow metrics since a flow-id maps to the same bucket.

[4] https://github.com/netobserv/ebpf-research/tree/main/ebpf-measurements

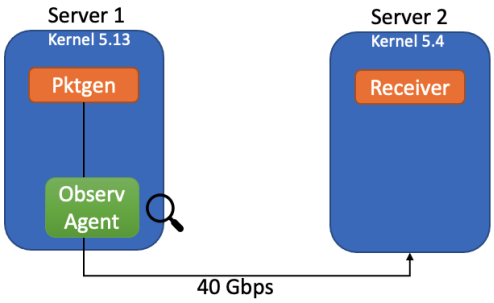


Fig. 2: Testbed Setup used for measurements and evaluation of *netobserv-ebpf-agent*

Measurement studies. Given the above eBPF map options for implementing the observability data-path, its very imperative to study the empirical performance of a data-path using each of the above choices. To do that, we implemented representative eBPF data-paths which collect flow metrics using each of the above mentioned data structures and study the performance by sending traffic using a custom-built UDP-based packet generator built on top of PcapPlusPlus [19]. The testbed setup for benchmarking is shown in Figure 2. The observe agent is the eBPF datapath performing flow metric collection, attached to the TC (traffic control) hook-point of the sender. We use two bare-metal servers connected over a 40G link. Packet generation is done using custom-built UDP-based packet generator over PcapPlusPlus [19] using 40 separate cores. To bring these measurements in perspective, libpcap-based Tcpdump (collecting similar flow information) is also used for comparison.

We run the tests by generating bursts of UDP packets belonging to a *single-flow* and *multi-flow*. While *single-flow* generated packets belonging to a single UDP flow using the 40 cores, *multi-flow* generates 40 different UDP flows (1 flow per core). Traffic was generated in bursts to generate maximum packets per seconds, while saturating the cores. As shown in the Figure 3, native performance without any observe agent is about 4.7 Mpps (Million Packets Per Second), and with tcpdump running, the throughput falls to about 1.8 Mpps. With eBPF-based observe agents, we observed that the performance varies from 1.6 Mpps to 4.7 Mpps based on the eBPF maps used to store the flow metrics. Using a shared data structure such as a single hash map, we observed the most significant drop in performance for a single-flow, because each packet writes to the same hash bucket in the map regardless of the CPU it originated from. However, performance improves when multiple-flows are introduced since the contention reduces for a single bucket. Ring buffer (RB) performs almost equally worse as single Hash Map for burst of packets from a single flow, however fails to improve with multiple flows, since the contention for the memory of the ring buffer remains a bottleneck.

Using a Per-CPU hash map (CpuHash), we observed an increase in throughput for a single flow, because packets arriving from multiple CPUs no longer contend for the same map bucket, and with multiple flows, the performance in-

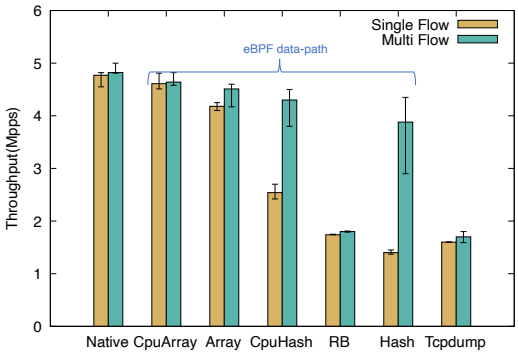


Fig. 3: Throughput observed comparison of different eBPF data-paths built using various data structures

creases significantly as the contention reduces further. With (Per-CPU) arrays, we see a significant increase in the throughput. We attribute this to the fact there is no contention between packets since each packet takes up a different entry in the array incrementally upon arrival. However, the major drawback is we do not handle the eviction of entries in the array upon full, while it performs writes in a circular fashion. Hence, it stores only the last few packet records observed at any time. Array eviction would require using redundant arrays which are periodically flushed (to userspace) to handle large packet rates and hence complicating the eBPF datapath, which makes it hard to maintain. Nevertheless, it provides us the spectrum of performance gains we can achieve by appropriately using the maps in the eBPF datapath.

Key Takeways:

- Ringbuffer and hash maps which are popularly used by several tools [12, 16, 13, 5, 2] have a significant effect on the underlying workload's performance.
- Per-CPU hash maps perform much better in handling multiple flows by aggregating flow metrics in multiple per-CPU buckets in the eBPF data-path.
- Arrays perform best due to absence of contention/locking, however it is hard to be implemented to handle millions of packet records at line-rate.

3.2 Design of *netobserv-ebpf-agent*

Flow Capture Agent Based on the insights gathered in the measurement studies, we design *netobserv-ebpf-agent* using Per-CPU hash maps to aggregate flow metrics in the eBPF datapath, since it offers good performance and is easier to implement and maintain from userspace. *netobserv-ebpf-agent* attaches to the TC hook-point of the individual network interfaces to monitor traffic going through. The architecture of the *netobserv-ebpf-agent* is illustrated in Figure 1, where *netobserv-ebpf-agent* is composed of an eBPF data-path and the userspace daemon. In the following, we explain the components of *netobserv-ebpf-agent* in detail.

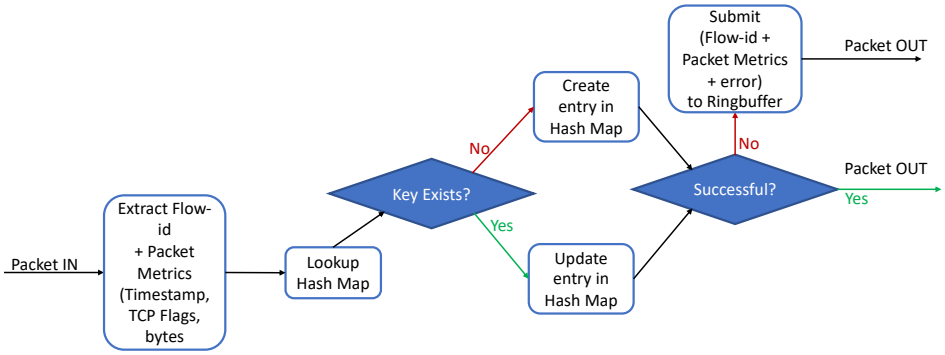


Fig. 4: Logic of the eBPF datapath

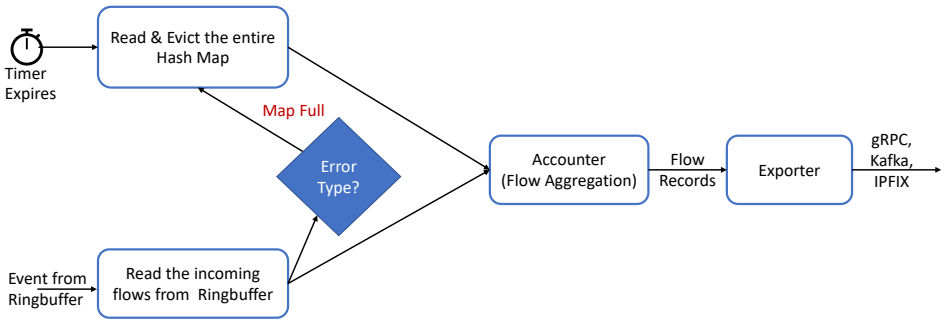


Fig. 5: Logic of the userspace daemon

eBPF Datapath. The eBPF Datapath comprises of a Per-CPU hash map which is used to map a flow-id from every incoming packet to a hash bucket in the map, which maintains metrics like bytes/packets counter, and start and last-seen timestamps. Upon a packet arrival, a lookup is performed on the hash map using the flow-id. If the lookup is successful, the corresponding packet/byte counters and last-seen timestamp are updated. The start timestamp is initialized when the bucket gets created or TCP SYN bit is set in the packet. However, the above logic could fail to map to a hash bucket upon certain scenarios : 1) The hash map is full, and 2) The hash map is busy (due to another operation/eviction). To tackle this scenario, we additionally use a ringbuffer map. When we encounter such scenarios, we send the flow-id with counters to user space using the ringbuffer map along with the error-type for the user space to take appropriate action. Hence, under a normal scenario, the flow counters are aggregated in the eBPF datapath using the Per-CPU hash map. Hash collisions are possible in the Per-CPU hash map, and they are handled by libbpf by maintaining an array of key/value pairs per hash bucket and verifying the key upon insertion/retrieval. We represent this logic in Figure 4.

User space. The user space daemon configures the eBPF datapath with settings such as the interfaces to be monitored, hash map size, eviction timeout, etc. As show in Figure 5, the user space daemon is responsible to periodically evict

the flow records in Per-CPU hash maps based on the configured eviction timeout. Upon eviction, the user space daemon also needs to aggregate the entries per-flow which could be scattered across different CPU buckets of the Per-CPU hash map, which uses different contiguous buckets for each CPU. The aggregated entries from the hash map are sent to the *accounter*. During eviction, the map bucket might be temporarily unavailable to write in the datapath. In such scenarios, the flow information is written to the ring buffer.

The user space daemon also listens to the ring buffer to receive individual flow entries from the eBPF datapath directly. Upon receiving a flow entry, the user space checks the error which the flow encountered in the eBPF datapath. If the error is due to *map being full*, then the user space daemon, immediately triggers a bulk eviction of the map to minimize the usage of ringbuffer (due to performance). However, if the error is due to the map being busy, then the flow entries are sent to the accounter. The user space daemon maintains mutex to ensure only a single eviction of the hash map is happening at a given time. The accounter performs another aggregation of flow entries from the hash map and ring buffer and temporarily stores the flow entries in a buffer before sending to the exporter. The exporter is pre-configured using configuration to export the flow records as gRPC, Kafka, IPFIX records.

The challenge with using Per-CPU hashmap in the eBPF datapath is that memory is not zeroed when an entry is evicted/removed by the user space daemon. Hence, if an entry is removed and a new flow maps to the same hash bucket from other CPUs, the older entries pertaining to the deleted flow would still be present in the hash bucket. This would produce inaccurate results when aggregating the individual flow entries across per-CPU hash buckets. To solve this in a lightweight manner, we additionally, maintain a timestamp of last eviction in the user space. During aggregation, we discard old flow entries whose last-seen timestamp is before the last eviction time. These old entries would eventually be written over by new flows.

In the following, we show the code snippets of the flow and map declarations. *flow_id* encodes both v4 and v6 addresses in *src_ip* and *dst_ip* to avoid maintaining a separate declaration for v6.

```
1  struct flow_metrics {
2      u32 packets;
3      u64 bytes;
4      u64 start_mono_time_ts; // Flow start monotomic timestamp in ns
5      u64 end_mono_time_ts; // Flow end monotomic timestamp in ns
6      u8 errno; // Positive errno of a failed map insertion
7      u16 flags; // TCP Flags encountered by the flow
8  };
9
10 // Attributes that uniquely identify a flow(flow-id)
11 struct flow_id {
12     u16 eth_protocol;
13     u8 direction;
14     u8 src_mac[ETH_ALEN];
```

```
    u8 dst_mac[ETH_ALEN];
    struct in6_addr src_ip; // IPv4 addresses are encoded as IPv6
    struct in6_addr dst_ip; // with prefix ::ffff/96
    u16 src_port, dst_port;
    u8 transport_protocol;
    u32 if_index;
};

// Flow record
struct flow_record_t {
    flow_id id;
    flow_metrics metrics;
};
```

```
// eBPF Map declarations for aggregating the flow
// Common Ringbuffer as a conduit for flows to userspace
struct {
    __uint(type, BPF_MAP_TYPE_RINGBUF);
    __uint(max_entries, 1 << 24);
} direct_flows SEC(".maps");

// Key: the flow identifier.
// Value: the flow metrics for that identifier.
struct {
    __uint(type, BPF_MAP_TYPE_PERCPU_HASH);
    __type(key, flow_id);
    __type(value, flow_metrics);
} aggregated_flows SEC(".maps");

```

Packet Capture Agent We additionally design a Packet Capture Agent (PCA) to selectively capture the entire packet payload for further analysis (for e.g. DNS inspection). The user space daemon also configures the eBPF datapath with PCA settings such enabling packet capture, specifying the filters for capture and the target where a collector will listen and receive a PCAP stream.

A Per-CPU Perf buffer is allotted to capture packets and write them to the userspace. The userspace daemon listens on the perf buffers and the packets are streamed as soon as a collector (for e.g. a wireshark client, zeek [25], etc.) connect to the agent on the specified PCA port.

```
//PerfEvent Array for Capturing full Packet Payloads
// Key: the packet idenitfier
// Value: the packet payload metadata
struct {
    __uint(type, BPF_MAP_TYPE_PERF_EVENT_ARRAY);
    __type(key, u32);
    __type(value, u32);
```

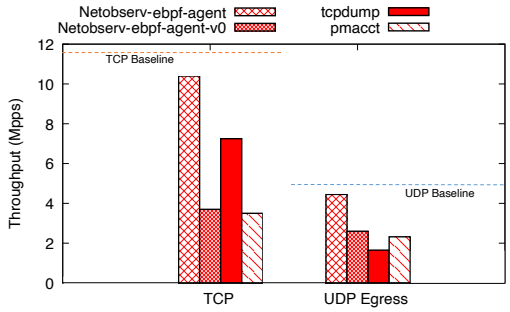


Fig. 6: Throughput observed by *netobserv-ebpf-agent* variants in comparison to existing tools

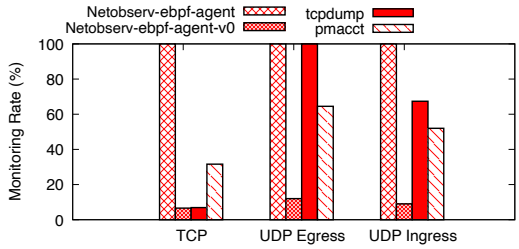


Fig. 7: Monitoring rate observed by *netobserv-ebpf-agent* variants in comparison to existing tools

```
    __uint(max_entries, 256);
} packet_record SEC(".maps");
```

Here, we show the code snippet of the packet metadata structure.

```
struct payload_metadata {
    u32 if_index;    // Interface Index
    u32 pkt_len;     // Length of the packet (headers+payload)
    u64 timestamp;  // Timestamp when packet is received by eBPF
};

```

Packet capture being resource intensive is turned off by default and could be turned on by setting $ENABLE\_PCA = true$. This is followed by setting up the filters and the port that packet capture server would listen to. The capture starts only when a client (receiver) connects to it. The filter is set using environment variable PCA_FILTER and port using PCA_SERVER_PORT. A filter is specified as a combination of protocol and port number e.g. `tcp,80`.

4 Evaluation

We perform the evaluation of *netobserv-ebpf-agent* using the setup shown in Figure 2 with two 80-core Intel(R) Xeon(R) Gold 5218R CPU @ 2.10GHz bare-

metals connected using 40 G Mellanox Connect-X5 NICs. We evaluate *netobserv-ebpf-agent* against other Observ agents : 1) pmacct (a popularly deployed libcap based filtering tool), 2) tcpdump, and 3) *netobserv-ebpf-agent*-v0 : An initial version of *netobserv-ebpf-agent* using only ringbuffer in the eBPF datapath, and entire flow aggregation being done in the user space. This version doesn't have a per-CPU hash map which performs flow-aggregation in the eBPF datapath. Hence, each packet's flow-id and packet size is submitted to a ring buffer. Many monitoring tools [16, 12, 13, 5, 2] use ring-buffer to submit critical events in the datapath

We evaluate the above *Observ agent*(s) under the three traffic scenarios :

- *TCP* : The *pktgen* in Figure 2 is located in Server 2 (ingress direction), and it generates multiple short TCP flows with 100-byte packets for a duration of 100 seconds.
- *UDP-Egress* : The *pktgen* is located in Server 1 along with the *Observ Agent* and generates multiple UDP flows of 75-byte packets. Hence, the packets pass through the *Observ Agent* upon egress.
- *UDP-Ingress* : The *pktgen* is located in Server 2 and generates multiple UDP flows of 75-byte packets. Hence, the packets pass through the *Observ Agent* upon ingress.

We use ingress setup to evaluate the monitoring rate which is calculated by Num(Packets Observed in Observ agent) /Num(Actual Packets) and the egress setup to evaluate the overhead of the *Observ agent* on the throughput generated. For TCP we do not have an egress setup, since ingress setup is sufficient to capture both throughput and monitoring rate. While we use iperf3 to generate 128 different TCP flows (-P option), we use PcapPlusPlus to generate 40 UDP flows (1 flow per core). The intention behind the packet generation is to evaluate the performance of *Observ agent* under stress conditions. Performing this evaluation in bare-metal environment provides more control to perform traffic generation, and fine-grained measurements. We initially plot the throughput observed by *pktgen*, when the *Observ agent* is running in the same node (scenarios *TCP*, and *UDP-Egress*) in Figure 6. *Baseline* refers to the native throughput observed by the *pktgen* without any *Observ agent* running. We observe that native throughput to be about 11.75 Mpps with TCP flows, and about 4.7 Mpps with UDP flows. With *Observ agents* such as pmacct and tcpdump, the throughput degradation is about 50-70%. With *netobserv-ebpf-agent*-v0, we observe a similar degradation in throughput due to usage of only ringbuffer. However with *netobserv-ebpf-agent*, we observe close to only 10% degradation in throughput due to flow aggregation using Per-CPU hash maps in the eBPF datapath.

Next, we measure the monitoring rate of the *Observ agent(s)* , which is calculated by Num(Packets Observed in Observ agent) / Num(Packets sent at line-rate) in Figure 7. While, *netobserv-ebpf-agent* is able to monitor and provide the flow records for all 100% of the traffic sent over the interfaces (with TCP, UDP Egress and UDP Ingress), we observed pmacct, tcpdump and *netobserv-ebpf-agent*-v0 drop significant portion of the traffic as the user space fails to

catch-up. While tcpdump monitors 100% of the traffic in *UDP Egress* scenario, it hampers the throughput of the outgoing traffic (Figure 6).

Finally, we measure the overheads of the *Observ agent(s)* in terms of additional CPU overhead and memory consumed by the process[5] during line-rate traffic. We observe that *netobserv-ebpf-agent* consumes a magnitude lesser in additional CPU consumption % (over 80 Cores) compared to other *Observ agents*. This is mainly attributable to the usage of Per-CPU hash maps, which perform aggregation in the kernel eBPF datapath, thus keeping the userspace daemon idle periodically. We observe the memory footprint of the *netobserv-ebpf-agent* process is only 12 MB, while it being higher for other *Observ agents*. *netobserv-ebpf-agent*-v0 uses a slightly lesser memory due to absence of hash map in the data path.

Table 1: Overheads of the Observ agents

Observ Agent	Extra CPU Overhead (%)			Memory (MB)
	TCP	UDP Egress	UDP Ingress	
netobserv-ebpf-agent	11.19	0.08	20.17	12
netobserv-ebpf-agent-v0	220.4	13.8	233.24	11
tcpdump	51.48	6.18	670.46	27.4
pmacct	234.23	4.68	416.85	89.2

Realistic Deployment Scenario. We deployed *netobserv-ebpf-agent* on a multi-node *Red Hat OpenShift* deployment deployed on AWS m6i.4xlarge EC2 instances (16 vCPU x 64GB memory x 12.5 Gbps network) to capture network metrics from two realistic benchmarking workloads: 1) Node-density-heavy (25 nodes), and 2) cluster-density (120 nodes) [6]. Table 2 captures the average flows processed per minute and the oevrhead incurred in terms of total CPU cores and memory consumed.

Table 2: Performance and Overhead of *netobserv-ebpf-agent* in a 120-node (m6i.4xlarge EC2 instances) *Red Hat OpenShift* deployment

Workload Type	Total Nodes	Flows/Min (Millions)	CPU (Cores)	Mem (GB)
Node-density-heavy	25	0.52	3.32	2.71
Cluster-density	120	1.92	10.14	11.13

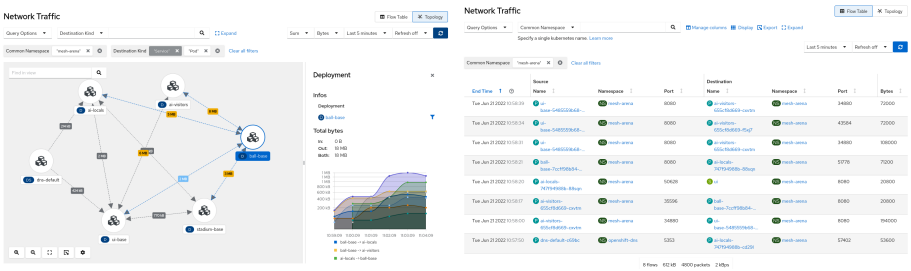
netobserv-ebpf-agent incurs a mere 0.83% CPU and 0.16% memory overhead in node-density-heavy scenarios and 0.52% CPU and 0.14% memory overhead in cluster-density workloads, impacting the deployment cost.

5 Deployment Model

To deploy *netobserv-ebpf-agent* in *Red Hat OpenShift*, we have released an observability operator in OperatorHub[6]. Upon installing and deploying the tool, *netobserv-ebpf-agent* is deployed in all the nodes in the cluster. The flows records exported by *netobserv-ebpf-agent* (gRPC/Kafka/IPFIX) from *netobserv-ebpf-agent* can be consumed by flowlogs-pipeline [9], which are first enriched with

[5] mpstat for CPU consumption and pmap for memory

[6] https://operatorhub.io/operator/netobserv-operator



(a) Network Topology with metrics (b) Network Metrics in dashboard

Fig. 8: Deployment & Visualization of *netobserv-ebpf-agent* in *Red Hat OpenShift*

kubernetes-specific contexts such as pod names, namespace, topology etc, and, in parallel, metrics are derived from the flow records for Prometheus consumption. Alternatively, the enriched metrics could also be exported as IPFIX records. Finally, the metrics are stored in a persistent storage (Loki [10]), and observed using Grafana or our dashboard in the *Red Hat OpenShift* as shown in Figure 8a. Figure 8b shows how the workload network metrics are displayed along with the contextual information (Namespace, app names, etc) in the observability dashboard. *netobserv-ebpf-agent* can also be deployed independently in bare-metal and other environments (OpenStack, VMs, etc) and be configured to export flow records from the host interfaces using IPFIX records.

Use-Cases. *netobserv-ebpf-agent* can be used to auditing network traffic in a cluster, troubleshooting connectivity of applications, cost-planning and provisioning network based on demand and so on.

6 Conclusion

Network observability is gaining more traction with web applications increasingly being refactored as microservices in the cloud. In this paper, we present the design and implementation of *netobserv-ebpf-agent*, an eBPF-based network observability agent that can be readily deployed in existing Kubernetes-based orchestration platforms to gain network insights of the application workloads in a transparent manner with very minimal overhead. In future, we plan to enhance *netobserv-ebpf-agent* with more connection-level performance insights such RTT which would particularly help diagnosing performance issues. We also plan to go beyond the standard headers to derive application-specific insights from the payload like dns analytics, video conferencing, etc. *netobserv-ebpf-agent* is available in opensource and has also been released with *Red Hat OpenShift* [3]

Acknowledgements. We would like to thank the anonymous reviews and our shepherd Ramakrishnan Durairajan for their valuable feedback. We would like to thank the Red Hat development team which lead and foresaw the active development, testing and release of *netobserv-ebpf-agent*. A special shoutout to Nathan Weinberg for benchmarking *netobserv-ebpf-agent* in realistic scenarios, Mario Macias who lead the work during the initial release and Mohammad Mah-

moud for his continuous improvements. Note this is a free version of the paper. Visit Springer site[7] for the published manuscript.

[7] https://link.springer.com/chapter/10.1007/978-3-031-56249-5_11

Bibliography

[1] A Guide to Providing Insight with Network Observability. https://cloud.redhat.com/blog/a-guide-to-providing-insight-with-network-observability.

[2] Aquasecurity Tracee. https://github.com/aquasecurity/tracee.

[3] Check out the new Network observability Support in OpenShift 4.12. https://www.redhat.com/en/blog/check-out-the-new-network-observability-support-in-openshift-4.12.

[4] Cilium eBPF. https://github.com/cilium/ebpf.

[5] Cilium Tetragon. https://github.com/cilium/tetragon.

[6] CloudBullDozer workloads. https://github.com/cloud-bulldozer/e2e-benchmarking/tree/master/workloads/kube-burner.

[7] Cloudflare ebpf_exporter. https://github.com/cloudflare/ebpf_exporter.

[8] eBPF. https://ebpf.io.

[9] Flowlogs-pipeline. https://github.com/netobserv/flowlogs-pipeline.

[10] Grafana Loki. https://grafana.com/oss/loki/.

[11] In-band Network Telemetry. https://github.com/p4lang/p4-applications/blob/master/docs/INT_v2_1.pdf.

[12] Inspektor-gadget. https://github.com/kinvolk/inspektor-gadget.

[13] Intel Host-INT. https://github.com/intel/host-int.

[14] IP Flow Information Export (IPFIX) Implementation Guidelines. https://datatracker.ietf.org/doc/html/rfc5153.

[15] l3af-project eBPF IPFIX Flow exporter. https://github.com/l3af-project/eBPF-Package-Repository/blob/main/ipfix-flow-exporter/bpf_ipfix_egress_kern.c.

[16] Libebpfflow. https://github.com/ntop/libebpfflow.

[17] Monitoring VM Traffic Using sFlow. https://docs.openvswitch.org/en/stable/howto/sflow/.

[18] OVN Kubernetes. https://github.com/ovn-org/ovn-kubernetes.

[19] PcapPlusPlus. https://pcapplusplus.github.io/.

[20] Pixie-IO Pixie. https://github.com/pixie-io/pixie.

[21] Skydive. https://github.com/skydive-project/skydive.

[22] TCPDump and Libpcap. https://www.tcpdump.org.

[23] The state of Linux in the public cloud for enterprises. https://www.redhat.com/en/resources/state-of-linux-in-public-cloud-for-enterprises.

[24] Types of eBPF maps. https://prototype-kernel.readthedocs.io/en/latest/bpf/ebpf_maps_types.html.

[25] Zeek. https://zeek.org .

[26] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller. The express data path: Fast programmable packet processing in the operating system kernel. CoNEXT '18, New York, NY, USA, 2018. Association for Computing Machinery.

[27] O. Michel, S. Sengupta, H. Kim, R. Netravali, and J. Rexford. Enabling passive measurement of zoom performance in production networks. In *Proceedings of the 22nd ACM Internet Measurement Conference*, IMC '22, New York, NY, USA, 2022. Association for Computing Machinery.

[28] A. W. Moore and K. Papagiannaki. Toward the accurate identification of network applications. In *Proceedings of the 6th International Conference on Passive and Active Network Measurement*, PAM'05, 2005.

[29] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado. The design and implementation of open vswitch. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, USA, 2015. USENIX Association.

[30] G. Yu, P. Chen, H. Chen, Z. Guan, Z. Huang, L. Jing, T. Weng, X. Sun, and X. Li. Microrank: End-to-end latency issue localization with extended spectrum analysis in microservice environments. In *Proceedings of the Web Conference 2021*, WWW '21, 2021.

[31] Y. Zhang, G. Kumar, N. Dukkipati, X. Wu, P. Jha, M. Chowdhury, and A. Vahdat. Aequitas: Admission control for performance-critical rpcs in datacenters. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22. Association for Computing Machinery, 2022.