A Comprehensive Performance Evaluation of Different Kubernetes CNI Plugins for Edge-based and Containerized Publish/Subscribe Applications

Zhuangwei Kang
Vanderbilt University
Nashville, Tennessee
zhuangwei.kang@vanderbilt.edu

Kyoungho An
Real-Time Innovations
Sunnyvale, California
kyoungho@rti.com

Aniruddha Gokhale
Vanderbilt University
Nashville, Tennessee
a.gokhale@vanderbilt.edu

Paul Pazandak
Real-Time Innovations
Sunnyvale, California
paul@rti.com

***Abstract*—The growing number of data- and latency-sensitive Internet of Things (IoT) applications is posing significant challenges for the edge and cloud deployment of publish/subscribe services, which are required by these applications. Two independently developed technologies show promise in addressing these challenges. First, Kubernetes (K8s) provides a de-facto standard for container orchestration that can manage and scale distributed applications in the cloud. Second, OMG's Data Distribution Service (DDS), a standardized real-time, data-centric and peer-to-peer publish/subscribe middleware, is being used in thousands of critical systems around the world. However, the feasibility of running DDS applications within K8s for latency-sensitive edge computing, and specifically the performance overhead of K8s' network virtualization on DDS applications is not yet well-understood. To address this, in this paper we evaluate the performance overhead of several container network interface (CNI) plugins including Flannel, WeaveNet and Kube-Router installed on a hybrid (ARM+AMD) edge/cloud K8s cluster. The paper reports results from a comprehensive set of experiments conducted to measure and analyze the performance (throughput, latency, and CPU/memory usage) of containerized DDS applications from the perspectives of virtualization overhead, reliability (DDS Reliable and BestEffort QoS), transport mechanisms (UDP unicast and multicast), and security. The insights derived from this study provide concrete guidance to developers of DDS-based applications in choosing the right virtual network plugin and configurations when hosting their real-time IoT applications in real-world containerized environments.**

***Index Terms*—Kubernetes, Container Networking, Data Distribution Service, Pub/Sub, Performance Evaluation**

I. Introduction

There is a growing trend towards deploying large-scale, data- and latency-sensitive Internet of Things (IoT) applications in number of industrial sectors, such as manufacturing, healthcare, energy and transportation. These applications must collect, store, and analyze fast-moving data streams generated by sensors in the cloud/edge service layers for real-time control and monitoring. Moreover, as the scale and complexity of such industrial IoT applications increases, completing an operation may require communication between multiple (potentially heterogeneous) sensors and cloud/edge services, and require (sub)millisecond-level decisions to streaming events.

The Object Management Group (OMG) Data Distribution Service (DDS) is an open standards-based middleware designed to support the data dissemination needs of real-time applications. DDS abstracts the underlying logic of data distribution and management to simplify the development of real-time publish/subscribe (pub/sub) applications. The publish-subscribe design of DDS enables decoupled communications between applications in time and space, so applications can dynamically join and leave any time from anywhere without breaking the system. DDS also supports a fully distributed peer-to-peer communication model and uses a binary wire format protocol. This helps it to meet performance and reliability requirements for mission-critical real-time applications.

Although DDS addresses the need for real-time and reliable information dissemination, the large scale of IoT applications and the heterogeneity of the deployment environment comprising a continuum from edge to cloud resources makes this deployment complicated. It is in this context that containerization technologies, such as Docker and Kubernetes, show promise. Containers enable lightweight encapsulation of functional modules, such as DDS' pub/sub capabilities, and offer resource isolation thereby allowing distributed applications and services to be easily deployed, scaled, and operated across distributed and heterogeneous platforms.

Managing a cluster of containers across distributed resources is supported by technologies, such as Kubernetes (K8s). K8s is the de-facto standard for orchestrating containerized workloads. With K8s, it is easy to deploy, update, scale, and self-heal distributed applications. For these reasons, there is growing demand in industry to use K8s to manage distributed and real-time applications.

Despite the promise of K8s, two fundamental gaps remain unresolved in deploying industrial strength IoT pub/sub applications across the edge/cloud continuum. First, the feasibility of deploying K8s-managed, containerized DDS pub/sub services for large-scale IoT applications that are deployed across the edge-to-cloud resources remains to be investigated. Second, K8s supports several container network interface (CNI) plugins that provide a uniform network namespace and DNS capabilities across the entire container cluster. However, since each CNI plugin is implemented in different ways, their performance when used with IoT applications based on

real-time pub/sub middleware, such as DDS, is likely to be different. Until now, no study has investigated this.

The research presented in this paper addresses these key gaps. To the best of our knowledge, this is the first comprehensive study of DDS performance in a K8s cluster deployment at the edge with different CNI plugins. Specifically, this paper makes the following contributions:

1) It validates the feasibility of operating DDS applications in a K8s cluster;
2) It explores and analyzes a set of K8s CNIs deployed on a K8s cluster with heterogeneous platforms;
3) It describes the automated benchmark framework that we developed to evaluate the performance of containerized DDS applications under various CNIs, workloads, and DDS quality of service (QoS) configurations; and
4) It presents a systematic set of experiments that quantitatively show the performance impact of K8s CNIs on DDS applications with different QoS settings and payload lengths.

The rest of the paper is organized as follows: Section II provides an overview of DDS and K8s, it explains the relevance of the K8s network model to DDS, and it compares the most commonly used CNI plugins for K8s; Section III introduces our evaluation environment, system configurations, benchmark framework, and reports on results of performance experiments; Section IV compares our work to related work; and finally, Section V provides insights, concluding remarks and future work.

II. Background on Underlying Technologies

To make the paper self-contained, this section provides the requisite background on the technologies used in this paper.

A. OMG Data Distribution Service (DDS)

The OMG DDS standard defines a data-centric, publish/subscribe (pub/sub) communication framework for real-time applications. DDS is designed to meet the performance, scalability, fault-tolerance, and security requirements of real-time and mission critical systems. The DDS pub/sub interaction model promotes loose coupling between applications with respect to time (i.e., the publisher and subscribers need not be present at the same time) and space (i.e., publishers and subscribers may be located anywhere). A core concept in DDS is data-centricity wherein the data moved between publisher and subscriber is treated as a first-class citizen in the system. This requires programmers to define a data model of named topics with specific data types, and to provide this information to the middleware. This allows DDS to understand the structure and values of the data it manages, which consequently allows it to perform a wide range of fine-grained data-centric operations and optimizations, such as validating data liveliness, content-based data filtering, etc.

Figure 1 presents the primary entities involved in a DDS application. A DomainParticipant is the starting point in the pipeline of creating a DDS application. It is responsible for (1) joining a DDS Domain (only participants using the same domain ID can communicate with each other); (2) creating DDS Publishers and Subscribers; (3) creating topics, data types, and QoS policies; (4) creating DDS Writers and Readers that are associated with created topics, data types, and QoS policies. A Publisher or Subscriber can have multiple DataWriters or DataReaders, respectively. Each DataWriter/-DataReader is associated with a single topic. When sending messages, samples written by a DataWriter are forwarded to matching DataReaders in terms of topics, data types, and QoS policies.

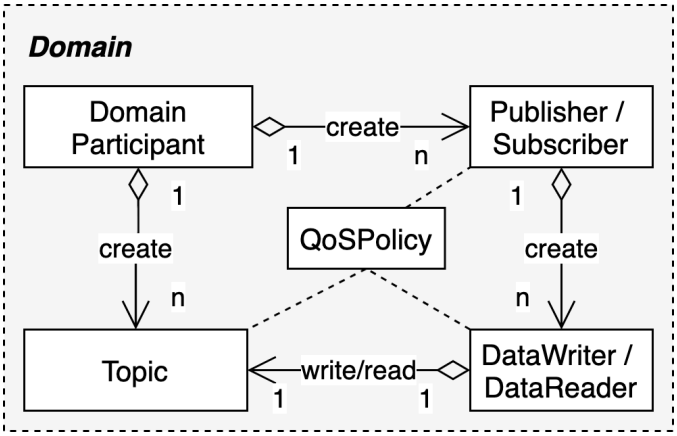

Fig. 1: DDS Architecture

One of the most compelling features of DDS is its support for a rich set of configurable QoS policies that can be applied on DDS objects at different levels of the architecture. DDS QoS is a concept that is used to specify the non-functional behavior of an application (e.g. reliable transmissions and persisting historical data). With XML-based configurable QoS policies, developers can define and update desired behaviors and resource distribution strategies for applications. For example, the Reliable QoS permits reliable communication based on a series of delivery confirmation mechanisms at the middleware layer even when using the unreliable UDP transport protocol. Likewise, Batching QoS allows publishers to buffer data samples and send in bulk for a high throughput use case.

B. Kubernetes and its CNI Plugin Support

Kubernetes (K8s) is an orchestration platform for deploying and managing containerized workloads and services. It helps manage the applications by scaling up and down, performing updates and rollbacks, self-healing, etc. The deployable unit of K8s is a pod that is essentially a collection of one or more containers with shared storage and network. Containers in a pod share an IP address and can communicate with each other over shared memory or localhost interface.

In a K8s cluster, every pod gets its own directly accessible IP address, and therefore does not require users to deal with mapping ports between containers. The K8s networking model [1] creates a clean, backward-compatible model where pods can be treated much like physical hosts. When they are

created, K8s pods get unreliable (i.e., non static) IP addresses as they are dynamically assigned. Therefore, pods are typically stitched to a K8s service that has a reliable IP address and a DNS name. Then, a K8s service load balances network traffic for the stitched backend pods. This model imposes the following fundamental requirements: (1) all containers can communicate with all other containers without Network Address Translation (NAT); (2) all nodes can communicate with all containers (and vice-versa) without NAT; (3) the IP that a container sees itself is the same IP that others see.

Kubernetes provides a plugin model for network connectivity called the Container Network Interface (CNI), which essentially constructs a flat network address space where each node owns an individual network segment (a CIDR block). A CNI plugin enables every pod running on a physical node to have a globally unique virtual IP address and eliminates the need for network address translation for pod-to-pod communications. The most popular contemporary CNIs include Flannel, Kube-Router, Calico, WeaveNet, and Cilium. However, the latest implementations of Calico and Cilium do not support the ARM architecture, which we need for our edge deployments; hence, this paper focuses on Kube-Router, Flannel, and WeaveNet only.

CNI plugins can be implemented at the data link layer (L2) or network layer (L3). Flannel enables both L2 (VXLAN) and L3 (Hostgw) backends, while Kube-Router and WeaveNet are implemented at L3 and L2, respectively. Figure 2 provides an architectural comparison between these technologies described below. We first introduce the *Host-mode*, which is a native networking solution in K8s and is used as the baseline setting for understanding the overhead of CNI network plugins reported in Section III and then delve into the details of each CNI.

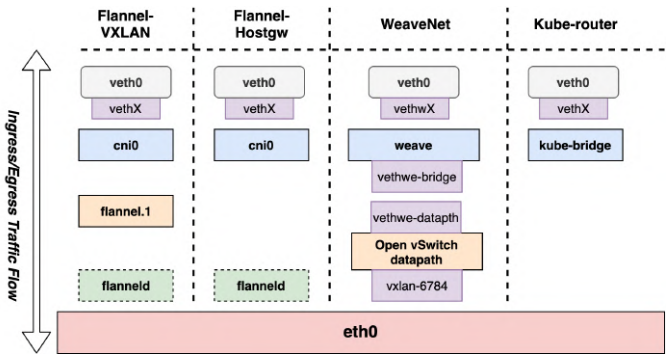

Fig. 2: Architecture Comparison of K8s CNIs

Host-mode is a native networking option in K8s allowing pods to share the host's network space. Therefore, it does not require installing any virtual network plugins; rather it establishes connections between containers using host IP addresses. This implies that users have to deal with port conflicts when running multiple containers on the same host. Moreover, since the Host-mode completely exposes the host network space to containers, any pod can sniff and manipulate the network traffic of other containers. Therefore, it is regarded as an insecure solution alone, but the security issue can be compensated by leveraging DDS Security that can protect applications from unauthorized access. . Host-mode is the best option for performance [2] [3] because it directly uses the host's network interface and hence its performance is adopted as the baseline in this paper.

Flannel [4] is the most straightforward and mature K8s CNI [5]. The Flannel network fabric is established and controlled by a daemon process called *flanneld,* which is responsible for 1) installing tunnel devices (interface and bridge) on the host machine and maintaining the network configuration with *etcd* (a distributed key-value store running on K8s master node); 2) allocating and managing IP address resources for pods; 3) emulating the ARP protocol and populating the host ARP table; and, 4) querying routing information (MAC, IP, VNI) when forwarding packets. Flannel can be paired with other backends (e.g., VXLAN and Host-gw) to meet the deployment demands on heterogeneous platforms and network environments. In terms of security support, Flannel provides an experimental backend called Flannel-IPSec that encapsulates and encrypts packets based on the Internet Protocol Security (IPSec) protocol.

Figure 2 shows the Flannel networking architecture on a single host when VXLAN and Host-gw backends are used. Specifically, the VXLAN backend creates an overlay on top of the underlying network based on the VXLAN tunneling technique. It installs a network bridge *cni0* and a virtual interface *flannel.1* on the host machine, where cni0 bridges traffic between the host-end virtual interface of VETH pair and flannel.1. When a sending packet arrives at the local flannel.1 interface, flanneld queries etcd to obtain the destination IP (physical) and encapsulates the packet into another UDP packet with the source and destination container IP (virtual) in kernel space. The Host-gw backend uses peer hosts as gateways, writing routing rules between source and target pods directly into the host's iptable ensuring that there are no in-kernel packet wrapping and unwrapping operations during communication which is required by the VXLAN option. The Host-gw backend incurs less overhead, but the L2 interconnection between nodes is required.

WeaveNet [6] fully emulates a L2 network whose topology is built by application-level virtual routers located on the user space of each host. WeaveNet routers set up full mesh TCP connections and synchronize routing information based on spanning tree and gossip protocols. WeaveNet supports two encapsulation modes: sleeve mode and fastpath mode. In the sleeve mode, it first captures packets from the weave bridge (see Figure 2), then forwards them to the virtual router in the user space to perform UDP encapsulation, and finally sends them back to the kernel space for post routing. In contrast, the fastpath mode adopts open-datapath (odp) of Open vSwitch (OVS) to wrap UDP packets with a VXLAN header in the kernel space, and then routes packets based on the odp routing tables issued by the virtual router. Since the fastpath mode has less context switches than the sleeve mode, it has been proven to earn better performance [7], and hence is

used in our analysis. Moreover, a distinctive feature supported by WeaveNet is unicast-based multicast, which means that containers are able to send multicast traffic which is then propagated by the physical network devices using unicast. Similar to Flannel-IPSec, the WeaveNet fastpath mode also supports data encryption based on IPSec; the difference being that Flannel realizes this in the kernel, while WeaveNet does so in the OVS VXLAN tunnel.

Kube-Router [8] is a lean yet powerful K8s CNI designed for the sake of simplifying the K8s network fabric, which can be seen from its architecture in Figure 2. Kube-Router constructs L3 virtual networks by deploying an agent on each node. When operating in a single subnet, Kube-Router employs the internal Border Gateway Protocol (iBGP) for exchanging routing and reachability information between pods. In the context of BGP, a K8s cluster is regarded as an autonomous system (AS), and nodes within the AS act like routers that form a full node-to-node mesh by advertising pod subnet CIDR to the rest of peers. To enable cross-subnet pod-to-pod communication, Kube-Router offers both overlay (IP-in-IP tunneling) and underlay (BGP) options, where the IP-in-IP tunnel encapsulates a pod IP packet as the payload of host IP packet. Kube-Router seamlessly integrates with out-of-the-box K8s functionalities, such as IP address management (IPAM), pod networking, Network Policy, etc. However, data encryption is not yet supported and so security features need to be implemented at the application or transport layer.

C. Kubernetes-managed Distributed DDS Application

DDS is an application-layer networking framework that relies on the underlying transport layer protocols (e.g. UDP, TCP). DDS participants rely on an internal DDS-specified discovery service to exchange their IP addresses for peer-to-peer communications. Therefore, DDS works better over a network without a NAT requirement. Since K8s does not require NAT, its container orchestration capabilities and networking model makes it better suited than Docker alone for realizing distributed DDS applications across heterogeneous resources.

With the DDS discovery service, containerized DDS participants belonging to the same DDS Domain can discover and establish connections with each other using topics, abstracting away IP-based communications. This allows DDS pods to discover and communicate without a K8s service thereby overcoming the unreliable IP address issue with pods. DDS typically employs multicast for the discovery service. However, if a K8s networking addon or a transport protocol does not support multicast, a complementary means to enable DDS participants to set up machine-to-machine connections automatically is an implementation provided by the RTI Cloud Discovery Service (CDS) [9], which plays the role of bridging and maintaining reachability information between participants. However, given that RTI CDS is a centralized standalone application in the system, a K8s StatefulSet can offer a set of replicated services to avoid this single point of failure.

III. Evaluating DDS Performance in Kubernetes

In this section we evaluate the performance of DDS applications deployed in a K8s cluster along the dimensions of reliability, scalability, and security. Accordingly, we aim to study the impact of K8s CNIs on DDS performance from different angles and propose deployment recommendations when different DDS QoS policies are used. It is worth noting that DDS uses UDP as the default transport protocol because it meets the design concept of real-time systems; hence, the following experiments employ UDP as the underlying transport protocol. The source code for the benchmarking capabilities presented in this paper are available at https://github.com/doc-vu/K8sDDS, which provides the experimental apparatur for examining the validity of our experimental approaches and results. Note our experiments were conducted on a resource-limited edge environment and performance values may vary to some extent under other hardware profiling and cluster scales.

A. Experimental Setup and Benchmark Framework

1) Hardware: The K8s cluster for our experiments is composed of one master node and 10 worker nodes. The master node is equipped with quad-core Intel i7-2600 3.40 GHz processors, 4GB RAM. Worker nodes are composed of 10 Raspberry Pi 3 Model B boards that each has four ARMv7l 1.20 GHz processors and 1GB RAM. All worker nodes are connected to a 100 Mbps LAN and we use the default MTU size (1500 bytes). This setup is representative of an indoor edge-fog deployment, e.g., a warehouse, factory floor, hospital or convention center where networks are stable but still use IoT devices to deploy applications. We chose this setup for two reasons: first, it is representative of real-time IoT applications deployed at the edge, and second, we wanted to maintain stable underlying network conditions to accurately pinpoint the effects of K8s CNI plugins on DDS performance.

2) Software: We installed Ubuntu 20.04 LTS (64bit) and Raspbian 9 (32bit) OS on the master and worker nodes, respectively. We used Docker 19.03 and Kubernetes 1.19 for container runtime and orchestration, respectively. The versions of CNI network addons used were: Flannel 0.11.0, WeaveNet 2.6.5 and Kube-Router 0.3.0. We leveraged the RTI Connext DDS v6.0.1, and its test harness called PerfTest v3.1 [10] for benchmarking applications and to measure latency and throughput with various configurations. To obtain optimal performance results, the OS network performance was tuned based on recommended configurations for RTI Connext DDS.[1].

3) Benchmark Framework: To reduce manual efforts of setting up the cluster and evaluating heterogeneous CNIs and DDS configurations, we developed an automated benchmark framework as shown in Figure 3, which is an out-of-the-box tool that can help DDS users to make CNI selection and configuration decisions under specific use cases. The K8s auto-deployer in the figure accepts a custom deployment profile,

[1] https://github.com/rticommunity/rtiperftest/blob/master/srcDoc/tuning_os.rst

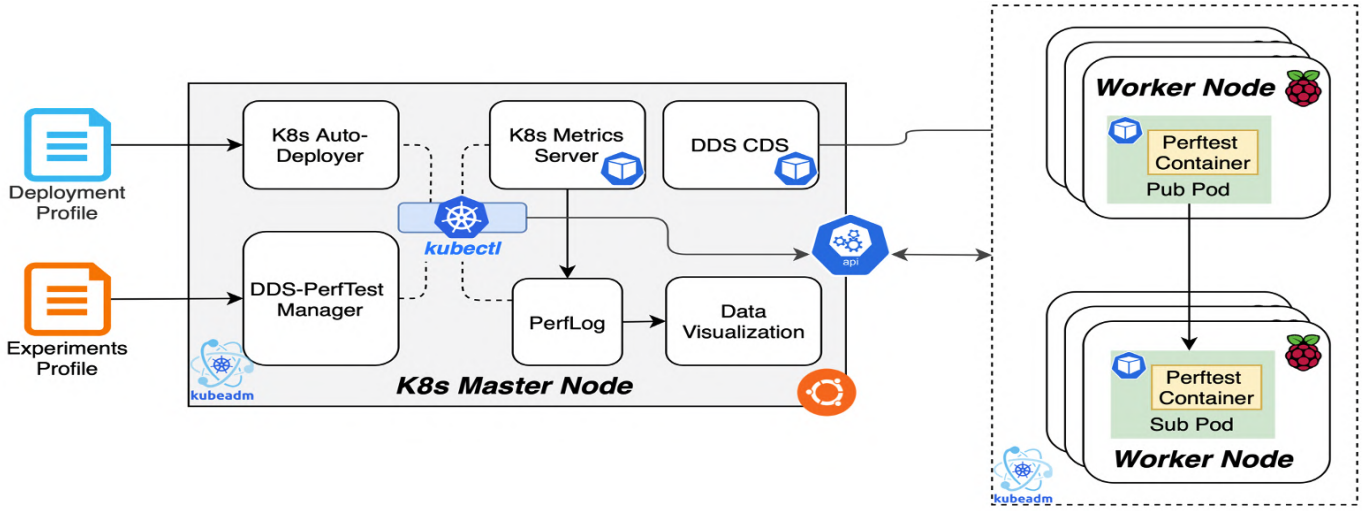

Fig. 3: Architecture of K8sDDS Benchmark Framework

then automatically carries out the following tasks in sequence: (1) initialize the K8s cluster; (2) install CNI plugins; (3) deploy Perftest pods, DDS-CDS Deployment, and Resource Metrics Monitor Deployment on the cluster. The functionality of each component is described below.

- A Perftest pod is a containerized Perftest benchmark application compiled with Connext DDS, which can be profiled as either a publisher or subscriber. It exposes a rich set of configurable DDS QoS policies and run-time experimental options allowing users to emulate heterogeneous use cases. Unlike broker-based service discovery mechanism in centralized messaging middleware, Connext DDS does so with multicast.
- The Cloud Discovery Service (CDS) is an external DDS infrastructure service used for discovery when multicast is not supported by the network infrastructure, such as K8s CNIs.
- Resource Metrics Monitor is a K8s built-in metrics server that is deployed as a K8s Deployment. It collects container CPU/memory usage information from a Kubelet (a K8s agent running on each node) and exposes them to K8s API server, which are then accessible through Metrics API or the K8s command-line tool, kubectl.

Although our experiments were conducted in an edge environment, the framework can be applied to a cloud-based cluster as well.

Once the cluster is created, the DDS-PerfTest Manager generates performance test commands with a given experiment profile and executes them on pods using the kubectl tool. The manager then monitors the experiment's progress and periodically gathers node resource usage from the K8s Metrics Server using the "kubectl top" command. For DDS performance measurements, we deploy the containerized PerfTest [10] pub/sub application pods on worker nodes. For single pub/sub tests, a PerfTest container was deployed on each Raspberry Pi board without any resource constraints. For multi-subscriber scenarios, we deployed multiple containers on a Raspberry Pi and limited each container to use a single core. A PerfTest application measures latency, throughput, and CPU usage internally, and reports execution summary through standard I/O when completed. Each machine used in our testbed accommodates a single publisher or subscriber pod in our experiments unless otherwise specified. By integrating the kubectl tool with the PerfTest application, we can easily invoke and trace a performance test. For instance, the following commands show an example of 1-pub/1-sub latency test:

```
nohup kubectl exec -t ./perftest_cpp -pub
    ↪ -latencyTest -dataLen 1024 -
    ↪ executionTime 120 -domain 1 -
    ↪ transport UDPv4 -nic eth0 -cpu -
    ↪ noPrint -- perftest-pub0 > logs/
    ↪ perftest-pub0.log 2>&1 &

nohup kubectl exec -t ./perftest_cpp -sub
    ↪ -latencyTest -dataLen 1024 -
    ↪ executionTime 120 -domain 1 -
    ↪ transport UDPv4 -nic eth0 -cpu -
    ↪ noPrint -- perftest-sub0 > logs/
    ↪ perftest-sub0.log 2>&1 &
```

PerfTest has two operational modes: Latency Test and Throughput Test. In the throughput test mode, a publisher sends data samples at a configurable rate, and a subscriber calculates throughput by counting the volume of received bytes and samples. PerfTest behaves differently in the Latency Test mode, where all samples are marked as latency samples, and they are exchanged in a stop-and-wait manner between publisher and subscriber. In each trip, specifically the publisher enters into the wait state after sending a "ping" sample until a responding "pong" is returned by subscriber. The Publisher computes the one-way latency from the measured round trip time. This measurement method gets rids of the impact of sys-

tem clock difference across machines on latency accuracy and the interference of network congestion. We used the default configuration of PerfTest unless specified. The following is an example command that we used for experiments:

B. Experimental Results

1) Virtualization Overhead: To understand the feasibility of encapsulating DDS-based real-time applications as containers in K8s clusters, it is worth investigating the performance overhead introduced by container virtualization. To this end, we design two sets of controlled experiments once running a 1pub-1sub DDS application directly on baremetal and once on containers with the Host-mode network.

Figure 4 shows experimental results with 1 publisher-1 subscriber over UDP unicast. The throughput of containerized DDS application is about 6% lower than the performance on baremetal when sending payloads that are smaller than 1KB. Previous studies [11], [12] indicate that container virtualization can lead to slight computation overhead for CPU-intensive applications, which is the case when writing small messages at unlimited rate. For large samples, with network bandwidth becoming saturated, the impact of container virtualization is negligible. Latency results indicate the end-to-end response time for the two types of deployments in a zero-load network condition. The performance difference is smaller than 1.5% over 5 payload lengths. Overall, the impact of container virtualization is not detrimental to the real-time property of DDS applications.

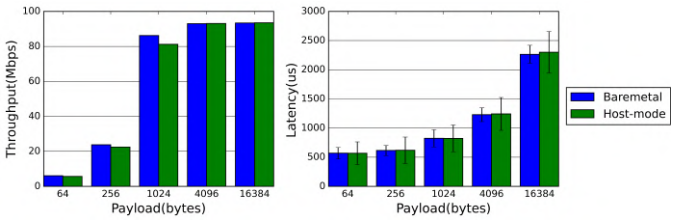

Fig. 4: Performance overhead of container virtualization in Throughput and Latency Tests. Figure shows mean throughput and 90th latency. Communication Pattern: 1pub-1sub. Publication rate: unlimited for throughput test and ping-pong mode for latency test. Reliability : enabled, Batching: disabled. Test period: 120 seconds.

2) CNI-based Network Virtualization Overhead: Due to the drawbacks of Host-mode (security, portability, and resource isolation), CNI-based network solutions are recommended for most K8s use cases. To quantify the performance overhead imposed by virtual networks on DDS applications, we conducted throughput and latency tests with different CNI plugins.

DDS accomplishes reliable data transmission over unreliable (UDP) transport relying on application-level heartbeat and acknowledgment/negative-acknowledgment (ACK/-NACK) mechanisms. For the throughput tests, the publisher distributes data samples at an unlimited publication rate. For a publisher that requests strict reliability, its DataWriter thread will block until receipt of an ACK signal from the subscriber that confirms a message has been delivered successfully. Then the confirmed sample is deleted from the queue, and DataWriter recovers to the writable state. In this experiment, the Reliable QoS is enabled the batching is disabled.

Although the default WeaveNet MTU is 1376 bytes, we manually configured it to 1438 bytes as it is the maximum effective MTU that allows WeaveNet to work in its fastpath mode on our testbed. Flannel uses the same MTU (1500 bytes) as the physical network interface when porting with the Host-gw backend, while using 1450 bytes MTU in VXLAN mode. The MTU allowed by Kube-Router is 1480 bytes. The rest of the experiments in this paper also adopt the same MTU configurations. The overhead ratios of each CNI on throughput and latency compared to Host-mode are summarized in Tables I and II, respectively.

$\Delta T(\%)$ Payload(B) / CNI	64	256	1024	4096	16384
Flannel-VXLAN	45.6	44.4	41.3	8.4	2.9
Flannel-Hostgw	29.8	28.4	24.6	0.3	-0.1
Kube-Router	28.1	28.4	24.5	0.3	-0.1
WeaveNet	52.6	51.6	49.5	15.5	2.8

TABLE I: The throughput overhead of each CNI relative to Host-mode in percentage.

$\Delta L(\%)$ Payload(B) / CNI	64	256	1024	4096	16384
Flannel-VXLAN	39.3	37.8	28.7	25.6	35.9
Flannel-Hostgw	16.5	15.2	13.1	18.8	27.7
Kube-Router	13.5	12.6	9.8	19.1	30.3
WeaveNet	52.2	48.7	38.6	33.1	41.4

TABLE II: The 90th latency overhead of each CNI relative to Host-mode in percentage.

Throughput results shown in Figure 5 and Table I reveal that CNIs that rely on VXLAN tunneling, such as Flannel-VXLAN and WeaveNet, present notable throughput degradation over all payload lengths due to the overhead of L2 encapsulation. WeaveNet's throughput is comparably lower than Flannel-VXLAN suggesting the OVS VXLAN encapsulation in WeaveNet yields more overhead than the native VXLAN implementation adopted by Flannel. Direct routing approaches like Flannel-Hostgw and Kube-Router obtain similar throughput. Although the MTU of Flannel-Hostgw is 20 bytes smaller than Kube-Router, the impact is negligible when the data frequency is unlimited. The overall trend indicates that the performance gap between Host-mode and CNIs gradually narrows with the increase in payload size. The performance of L3 CNIs is on par with the Host-mode when the bandwidth is saturated. Therefore, the extent of performance degradation when running DDS applications in containers depends on the CNI used, payload size, and network load condition.

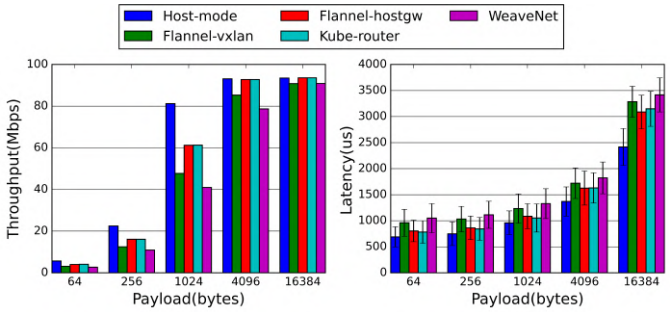

Fig. 5: Performance comparison of different K8s CNI plugins in Throughput and Latency Tests. Figure shows mean throughput and 90th latency. Communication Pattern: 1pub-1sub. Publication rate: unlimited for throughput test and ping-pong mode for latency test. Reliability: enabled, Batching: disabled. Test period: 120 seconds.

For the latency test (see Figure 5 and Table II), we notice that Kube-Router has around 3% better latency than Flannel-Hostgw for messages that are smaller than 1KB indicating that Kube-Router spends less time in encapsulating and routing packets. However, when the payload length is greater than 1KB, Flannel-Hostgw performs better than Kube-Router, which is caused by the difference in MTU. For instance, a 16KB message is chopped into $\lceil \frac{16K}{1480B} \rceil = 12$ IP segments when using Kube-Router for pod networking, while Flannel-Hostgw needs $\lceil \frac{16K}{1500B} \rceil = 11$ instead. WeaveNet is slower than Flannel-VXLAN by 4.3%-12.9% over five payload lengths, which further testifies WeaveNet's data encapsulation strategy incurs more overhead.

3) BestEffort QoS: Apart from the stringent reliability QoS employed in the previous experiment, DDS offers a weak reliability choice, called BestEffort, which meets the demand of applications that desire superior end-to-end latency but tolerable packet dropout and out-of-order delivery. When BestEffort QoS is enabled, every sample is written immediately and disseminated *at most once* and no extra resources are consumed for maintaining reliable connections between publishers and subscribers. If the send queue is full, DataWriter replaces the last sample with the incoming one. With BestEffort, applications are expected to obtain lower latency due to lower application-level overhead while the throughput performance correlates closely with the bandwidth utilization rate.

Publishers with BestEffort QoS can use bandwidth more effectively in low load conditions but may aggravate congestion if network is overwhelmed. Therefore, it is unreliable to observe the maximum throughput with BestEffort. In this case, this experiment concentrates on the latency of unreliable DDS applications on each CNI and compares the performance changes relative to Reliable mode. In the latency test, we configured a 200 millisecond timeout period for the publisher to wait for a "ping" response.

Figure 6 shows the latency numbers when enabling BestEffort and the performance gain compared to the Reliable mode. Similar to the previous experiment, L2 CNIs have higher latency than L3 ones. However, their latency dramatically reduces in the BestEffort mode.

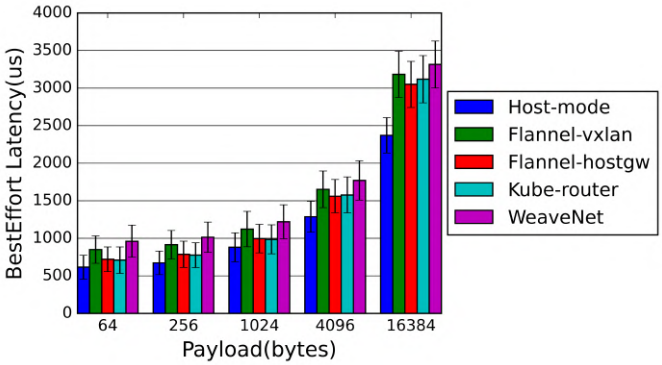

Fig. 6: 90th Latency with BestEffort QoS. Communication Pattern: 1pub-1sub. Transport: UDP. Batching: disabled. Test period: 120 seconds.

4) Multicast vs. Unicast: DDS supports reliable middleware-level multicast with both underlying unicast and multicast transports in which case application performance and scalability is closely influenced by the underlying transport mechanism. As such, this experiment investigates performance with unicast and multicast of each CNI in multi-subscriber scenarios. DDS can leverage multicast for scalability with multiple subscribers. However, WeaveNet is the only K8s CNI plugin that emulates L2 multicast. With that, a user application sends multicast packets to WeaveNet, and then WeaveNet disseminates the packets to multiple receivers using unicast. Other CNI plugins only support unicast, in which case, DDS takes care of sending data samples to multiple subscribers.

In general, multicast is more efficient when the same data needs to be sent to multiple subscribers. By using multicast, the network bandwidth and CPU usage of publisher pod will be constant, regardless of the number of subscribers. Therefore, a publisher with WeaveNet is expected to use less resources and performs better than others when there are multiple subscribers. According to our previous experiments, Kube-Router and Flannel-Hostgw are preferred over WeaveNet in reliable unicast communication. This raises the question whether WeaveNet multicast can make up for the overhead yield from OVS VXLAN encapsulation, and thus become a more promising solution in multi-subscriber use cases. For this experiment, we deployed each pub/sub container on different nodes. In the throughput test, a single publisher continuously sends 1KB workloads to 4/8/12 subscribers at an unlimited publication rate.

WeaveNet fully emulates layer-2 network supporting multicast, but incurs nontrivial overheads as shown in Figure 7. The performance of WeaveNet-Multicast lags behind all other unicast solutions when there are four subscribers, then exceeds WeaveNet-Unicast and Flannel-VXLAN in the 1pub/8sub use case, eventually becoming the optimal CNI as the number of

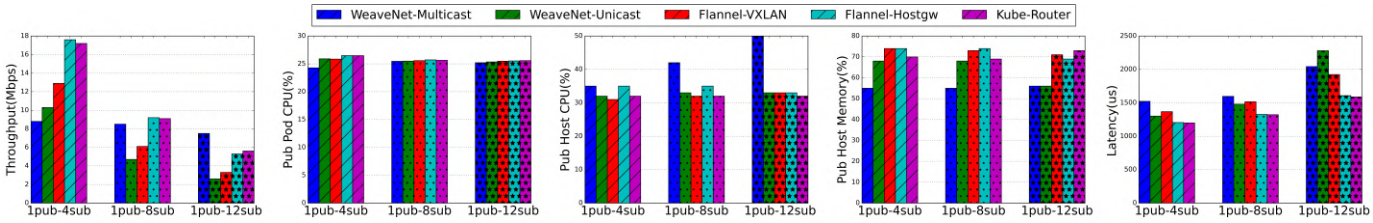

Fig. 7: Performance comparison of multicast/unicast-enabled CNIs in aspects of throughput, pub pod CPU usage, pub host CPU usage, pub host memory usage, and 90th ping-pong latency. The left four plots are produced by the same throughput tests. QoS settings: Reliable QoS: enabled, Batching QoS: disabled, Transport: UDP. Payload length: 1KB. Test period: 120 seconds.

subscribers increases to 12. The throughput results in Figure 7 confirms that unicast throughput linearly decreases by the number of subscribers, while performance with emulated multicast degrades at a relatively slower pace. In the throughput test, since the publication rate is unlimited, the DataWriter is blocked once the writing buffer at the middleware layer overflows. Therefore, the CPU usage of perftest process and publisher host are restricted to around 25% and 32%, respectively. However, for WeaveNet-Multicast, the process of wrapping samples to individual subscribers takes place in the Weave container, leading to incrementally increasing CPU utilization on the host. WeaveNet-Multicast effectively saves host memory due to its multicast functionality on the virtual network stack. The relatively low memory usage and throughput of WeaveNet-Unicast are due to the overhead of OVS VXLAN encapsulation.

Recalling the description from the previous 1pub-1sub experiments, we compute one-way latency based on the round-trip time, which prevents the influence of system clock differences on the accuracy of latency measurements at the nanosecond level. In an N-subscriber scenario, however, if all subscribers respond to a latency "ping" sample, the publisher needs to process N "pong" samples, which will artificially increase the round-trip time of the N_{th} "pong" as the previous $N-1$ responses have to be processed in a single-thread, consequently resulting in inaccurate latency observations. Therefore, we measure the average one-way latency over all subscribers by letting them return a single "pong" sample in a round-robin manner, such that

$$L_{avg} = \frac{1}{K} \sum_{i=1}^{K} L_{i\%N} \tag{1}$$

, where $L_{i\%N}$ is the one-way latency of the (i%N)-th subscriber and K is the total number of samples emitted by publisher during the test. Also, since the underlying layer of WeaveNet-Multicast is unicast, it is reasonable to compare its latency results with other unicast-based CNIs. According to Figure 7, the average latency over all subscribers for WeaveNet-Multicast is greater than others, which means the delay introduced by emulating multicast is non-negligible. In contrast, since considerable duplicated messages are cached in the fix-sized send queue in high traffic scenarios, unicast-based approaches obtain higher latency.

Overall, we conclude that performance in multi-subscriber scenarios is coupled with the number of subscribers and the data frequency. For small-scale systems where a publisher has lower data frequency, unicast-based CNIs are recommended; otherwise, WeaveNet-Multicast is more scalable and and therefore a better option for high throughput and low latency use cases.

5) Evaluating Security Support: Communication between containerized DDS applications in the K8s cluster can be protected at the middleware layer (DDS Security extension), transport layer (DTLS), or IP layer (IPSec), where the security addon and DTLS are defined by the OMG DDS security model and IPSec is supported in some K8s CNIs. Given that the differences between DDS Security plugin and DTLS have been well-studied in [13] and DDS Security plugin is more suitable for DDS-based systems, this section delves into the comparison of DDS Security extension and IPSec approaches in a K8s deployment. Particularly, Flannel and WeaveNet employ the Encapsulating Security Payload (ESP) protocol to secure IP packets, which is a member of IPSec protocol suite.

Authentication, integrity, data-origin authenticity, confidentiality, and access control are core foundations of information security. The first phase of secure communication is to establish mutual trust connection between participants. The DDS Security plugin does so with Public Key Infrastructure (PKI) framework. The OpenSSL library (v1.1.1d for this experiment) is adopted for supporting diverse cipher suites and certificate exchange choices. The IPSec approaches in Flannel and WeaveNet instead rely on the Pre-shared Key (PSK) authentication method and the Internet Key Exchange protocol.

For any received message, the DDS Security plugin and Flannel-IPSec first examine the origin authenticity and integrity of ciphertext by computing a 128 (for this experiment)/192/256-bit message authentication tag using AES-GMAC. The tag length is 128 bits in WeaveNet. This ensures the packet is not damaged or altered since creation time. In terms of data encryption, AES-GCM algorithm with either 128 (for this experiment), 192, or 256 bits keys is supported by the DDS Security plugin and Flannel-IPSec. WeaveNet does so with AES-256-GCM. However, Flannel operates IPSec

ESP in tunnel mode that encapsulates and encrypts whole IP packets, while WeaveNet encapsulates packets using OVS VXLAN and encrypts UDP datagrams in ESP transport mode. Therefore, Flannel-IPSec protects the pod IP in transmission but incurs more encryption overhead. Overall, the effective MTU of WeaveNet on our testbed reduces to 1414 bytes due to 50 bytes VXLAN and 34 bytes ESP cryptographic cost. Likewise, the effective MTU of Flannel-IPsec is verified at 1,423 bytes, including 20 bytes IP header and 57 bytes ESP overhead. Previous sections demonstrate Kube-Router stands out in reliable unicast performance, thus it is used to build the pod network for this experiment when evaluating the DDS Security plugin.

Compared to IPSec ESP, DDS Security extension decouples data integrity and confidentiality in a configurable way and seamlessly knits into the current DDS Core architecture. The DDS payload sent on the wire are encapsulated with the Real-Time Publish Subscribe (RTPS) protocol [14], which contains an RTPS header followed by a set of RTPS submessages. The submessage model defines a variety of DDS built-in signals for enforcing QoS and realizing reliable communication over unreliable channels. Particularly, serialized user data is defined as a class of submessage called DATA. According to the hierarchical structure of RTPS model, the DDS Security standard defines three data protection kinds for user data, RTPS Submessage, and RTPS message: SIGN, ENCRYPT, and WITH_ORIGIN_AUTHENTICATION. They, respectively, correspond to the protection for data integrity, confidentiality, and source authenticity offered by DDS Security. SIGN and ENCRYPT can be applied on all levels, and are compatible with WITH_ORIGIN_AUTHENTICATION when securing RTPS Submessage or RTPS message. Moreover, DDS Security allows users to define access control permission for every domain and topic, which are not available in IPSec methods.

DDS Security provides a comparable data protection level as IPSec ESP when encrypting RTPS submessages, signing RTPS, and authenticating the origin of RTPS together, which corresponds to the SignWithOrigAuthRTPSEncryptSM test in Figure 8. With that, Kube-Router with DDS Security performs better than Flannel-IPSec when sending messages that are smaller than 1KB but earns lower throughput and higher latency for larger ones. However, the fine-grained data protection functions that are unique to DDS Security plugin enable applications to plug tailored security options for a specific domain or topic to avoid performance overhead induced by unnecessary security operations. Although the ESP transport mode of WeaveNet IPSec incurs lower encapsulation overhead than the tunnel mode in Flannel, the impact of VXLAN encapsulation offsets the overall performance.

For multi-subscriber use cases, Figure 9 indicates DDS Security produces better throughput performance than security provided by virtual networks. The reason lies in DDS Security ciphers only once when transmitting samples to multiple recipients; in contrast, IPSec ESP has to repeat this process for each destination session, which is not scalable for one-to-many pub/sub communications.

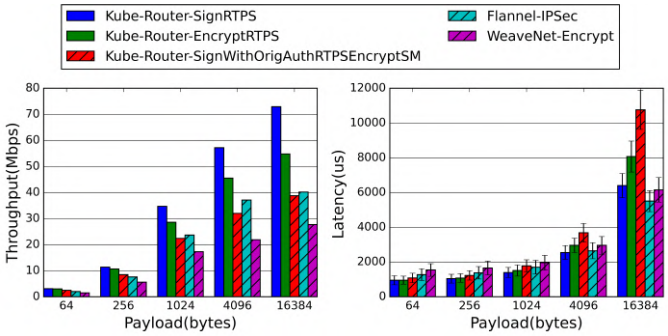

Fig. 8: Performance comparison of security implementations on application-layer(DDS Security plugin) and network-layer(Flannel-IPSec, WeaveNet-Encrypt). Communication pattern: 1pub-1sub. Reliability: enabled, Batching: disabled. Test period: 120 seconds.

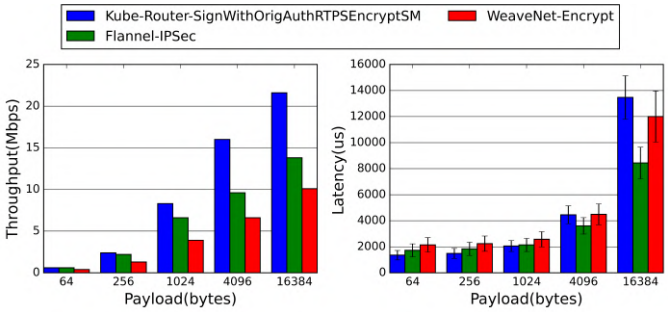

Fig. 9: Performance comparison of security implementations on application-layer(DDS Security plugin) and network-layer(Flannel-IPSec, WeaveNet-Encrypt) in multi-subscriber use case. Communication pattern: 1pub-4sub. Reliability: enabled, Batching: disabled. Test period: 120 seconds.

IV. Related Work

Several efforts have studied the performance of DDS and K8s separately but we have not found any prior work that studies DDS performance in the context of K8s-managed deployment. Related to DDS, existing efforts usually compare DDS with other IoT middleware horizontally. For instance, [15] compares the latency, bandwidth consumption and packet loss of DDS, MQTT, CoAP and a custom UDP application under a constrained wireless access network. Likewise, [16] includes more middleware protocols, such as MQTTSN, AMQP, and XMPP. The work in [17] surveys multiple middleware protocols including DDS based on their primary characteristics and potential performance issues on throughput, latency, and energy consumption. Similarly, [18] focused on DDS, ROS, OPC UA, and MQTT, and measured the round trip time of messages in different system states: idle, high CPU load, and high network load. Compared to these efforts, we concentrate on containerized DDS applications and analyze the performance from multiple perspectives: reliability, scalability, and

security. Longitudinal studies, such as [19] investigated three popular DDS implementations comparing their architectures and performance. Our work uses the RTI Connext DDS, which is one of the best-known DDS implementations and provides the most mature and thorough support for the OMG DDS standards. In [20], authors explored the overhead and side-effects of a variety of VM-based virtualization methods for distributed systems using DDS. Compared to virtual machines, containers are more lightweight and easier to be profiled, and thus better suited to resource-limited edge environments. As such, our study presents detailed analyses of the impact of various network virtualization techniques on containerized DDS applications.

Related to K8s, [21] evaluated the performance of K8s virtual network plugins with multiple transport protocols (TCP, UDP, FTP, HTTP, and SCP) in terms of MTU auto-detection, throughput, memory and CPU utilization. They used iperf3[2] as the benchmarking application. Similarly, Suo et al. [2] conducted qualitative and quantitative analyses of container networks including Docker Overlay, Flannel, WeaveNet and Calico. Their benchmarks comprised four domain-specific networking applications but did not include DDS. Although scalability and security of the involved networking solutions were discussed qualitatively, corresponding measurements of virtual network addons were not presented. There are edge-based K8s distributions, such as K3s [22] and KubeEdge [23]. Previous studies [24], [25] compared the resource utilization of these distributions over multiple use cases. However, the performance of DDS containers on K3s and KubeEdge remains to be investigated, which will be one of our future works. In addition, many popular distributed messaging systems have been deployed on K8s. Javadekar [26], Donca [27] and the study in [28] illustrate the practice of deploying Kafka, RabbitMQ, MQTT, and HiveMQ on K8s.

The authors in [5] evaluated the performance of Flannel, Swarm Overlay and Calico. Their latency, and TCP and UDP throughput results reveal that Calico has the highest performance, and its TCP throughput is close to host network. The purpose of [29] is similar to ours; they analyzed the performance overhead of OVN, Flannel, WeaveNet, and Calico on CoAP and FTP applications. In comparison, our work addresses several gaps in the aforementioned efforts: (1) we took application-level QoS properties into account when designing experiments, such as DDS Reliable and BestEffort QoS policies shown in Section III; (2) we compared unicast and multicast performance of CNIs in multi-recipient scenarios; and (3) we investigated the performance overhead of security operations implemented at the application layer and network layer.

V. Conclusions

This paper validates the feasibility of deploying DDS-based IoT applications with K8s specifically in a hybrid platform setup where the K8s master runs on a relatively powerful AMD machine compared to the remaining edge nodes, which are ARM devices. It qualitatively analyzes the overhead of three mainstream K8s CNI plugins that support hybrid-platform deployment. Lastly, it evaluates their performance by executing a systematic set of DDS benchmarking tests under a variety of workload patterns and QoS configurations. The automated benchmark framework presented in this paper provides users with an out-of-the-box tool to evaluate DDS performance on K8s clusters.

Our experimental results reveal that there is no one-size-fits-all CNI recommendation when deploying DDS applications on K8s clusters. The performance of DDS applications on different CNIs is affected by system topology, QoS policies, and CNI configurations. Specifically, for deploying K8s-managed DDS applications, we provide the following take-home messages on choosing the appropriate CNIs.

1) From the point of view of performance only, the K8s Host-mode achieves the best performance across multiple scenarios, but it cannot isolate the container network space, which may lead to security issues. However, the security issues can be resolved if applications can leverage DDS Security. Also, it may require additional configuration efforts to avoid port conflicts. DDS automatically allocates unused ports to avoid conflicts, and therefore the port conflict issue can be mitigated with using DDS.
2) This paper analyzed containerized DDS application performance on a hybrid-platform (ARM+AMD) K8s cluster when integrating with Flannel, WeaveNet, and Kube-Router. DDS containers earn better performance when using Kube-Router or Flannel-Hostgw CNI in reliable unicast communication.
3) Compared to the reliable mode, enabling DDS Best-Effort QoS is a good practice to reduce latency and L2 CNIs show better performance improvement than Flannel-Hostgw and Kube-Router.
4) Although WeaveNet is the only CNI that emulates L2 multicast, the overhead induced by VXLAN encapsulation may offset the multicast's benefit when operating in small-scale clusters. But it promises better scalability in large-scale multi-subscriber use cases compared to Flannel-Hostgw and Kube-Router.
5) To protect DDS packets from malicious attacks, the DDS security extension offers more flexible and fine-grained protection than IPSec approaches supported by Flannel and WeaveNet. In the 1-pub/1-sub communication, DDS Security outperforms Flannel-IPSec and WeaveNet-Encrypt when the payload is smaller than MTU, while incurs more overhead when sending large samples.
6) In multi-subscriber use cases, DDS Security is the best practice for achieving high performance and scalability. Our observation also indicates that Flannel-IPsec performs better than WeaveNet-Encrypt in unicast communication. In addition, the DDS Security plugin prevails

[2] https://iperf.fr/

over IPSec solutions in multi-subscriber use cases, thus delivering better scalability.

A. Scalability Discussion

The primary factors that affect scalability of K8s-based DDS applications are twofold: (1) the overheads of K8s CNI on network protocol stack and (2) DDS communication pattern (unicast/multicast). Although our experimental cluster was limited to 10 nodes, the empirical results indicate that L3 CNIs incur lower performance overhead than L2 ones. Thus, it is reasonable to infer that the benefits persist in even larger clusters, which is also shown in [30]. As multicast is naturally more scalable than unicast in multi-subscriber scenarios, the benefit of WeaveNet multicast is extensible to larger clusters.

B. Limitations and Future Work

The experiments reported in this paper were limited to understanding the maximum throughput and minimum latency that DDS can achieve when integrated with different K8s CNIs. The performance of containerized DDS applications with dynamic data flows has not been evaluated. In addition, considering the heterogeneity of workload patterns, deployment environments, and system typologies in real-world DDS use cases, tuning DDS and K8s CNI knobs under particular QoS and hardware restrictions intelligently and adaptively is a significant step for performance management of containerized DDS applications, which becomes one of our future work. Kube-router and WeaveNet support K8s Network Policies that allow users to control traffic flow at the IP address or port level. It could impact the performance of DDS applications. Our future work will focus on these evaluations.

This paper is also limited to use cases of operating DDS applications on a single K8s cluster, but the feasibility, performance, and scalability of cross-cluster communication of containerized DDS applications has not been systematically studied. Leveraging eBPF-based [31] packet filtering features provided by Calico [32] CNI plugin to improve DDS performance and discovery scalability is an interesting future direction to explore. Load balancing with session stickiness would be also interesting to scale out stateful DDS services. Last but not least, this paper, as a foundation work, will guide our future research to depict a blueprint to achieving high-performance and automated deployment of containerized DDS applications on K8s platforms.

ACKNOWLEDGMENTS

We thank Dr. Gerardo Pardo-Castellote from RTI for valuable guidance and feedback on this work. The authors are also grateful for insightful comments and suggestions from anonymous reviewers.

REFERENCES

[1] T. K. Authors, "Kubernetes cluster networking," https://kubernetes.io/docs/concepts/cluster-administration/networking/#the-kubernetes-network-model, 2020.

[2] K. Suo, Y. Zhao, W. Chen, and J. Rao, "An analysis and empirical study of container networks," in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE, 2018, pp. 189–197.

[3] T. Yu, S. A. Noghabi, S. Raindel, H. Liu, J. Padhye, and V. Sekar, "Freeflow: High performance container networking," in *Proceedings of the 15th ACM workshop on hot topics in networks*, 2016, pp. 43–49.

[4] Flannel-Io, "flannel-io/flannel." [Online]. Available: https://github.com/flannel-io/flannel

[5] H. Zeng, B. Wang, W. Deng, and W. Zhang, "Measurement and evaluation for docker container networking," in *2017 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)*. IEEE, 2017, pp. 105–108.

[6] Weaveworks, "weaveworks/weave." [Online]. Available: https://github.com/weaveworks/weave

[7] "Weave networking performance with the new fast data path," https://www.weave.works/blog/weave-docker-networking-performance-fast-data-path/.

[8] C. Labs, "cloudnativelabs/kube-router." [Online]. Available: https://github.com/cloudnativelabs/kube-router

[9] R.-T. Innovations, "Dds discovery in cloud-based environment," https://www.rti.com/developers/rti-labs/discover-data-in-cloud-services-with-cloud-discovery-service, 2020.

[10] ——, "Rti_perftest 3.0 documentation," https://community.rti.com/static/documentation/perftest/3.0/index.html, 2019.

[11] N. Huber, M. von Quast, M. Hauck, and S. Kounev, "Evaluating and modeling virtualization performance overhead for cloud environments." *CLOSER*, vol. 11, pp. 563–573, 2011.

[12] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. De Rose, "Performance evaluation of container-based virtualization for high performance computing environments," in *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. IEEE, 2013, pp. 233–240.

[13] M. Friesen, G. Karthikeyan, S. Heiss, L. Wisniewski, and H. Trsek, "A comparative evaluation of security mechanisms in dds, tls and dtls," in *Kommunikation und Bildverarbeitung in der Automation*. Springer Vieweg, Berlin, Heidelberg, 2020, pp. 201–216.

[14] O. M. Group, "Dds interoperability wire protocol," https://www.omg.org/spec/DDSI-RTPS/.

[15] Y. Chen and T. Kunz, "Performance evaluation of iot protocols under a constrained wireless access network," in *2016 International Conference on Selected Topics in Mobile & Wireless Networking (MoWNeT)*. IEEE, 2016, pp. 1–7.

[16] M. Anusha, E. S. Babu, L. M. Reddy, A. Krishna, and B. Bhagyasree, "Performance analysis of data protocols of internet of things: a qualitative review," *International Journal of Pure and Applied Mathematics*, vol. 115, no. 6, pp. 37–47, 2017.

[17] J. Dizdarević, F. Carpio, A. Jukan, and X. Masip-Bruin, "A survey of communication protocols for internet of things and related challenges of fog and cloud computing integration," *ACM Computing Surveys (CSUR)*, vol. 51, no. 6, pp. 1–29, 2019.

[18] S. Profanter, A. Tekat, K. Dorofeev, M. Rickert, and A. Knoll, "Opc ua versus ros, dds, and mqtt: performance evaluation of industry 4.0 protocols," in *Proceedings of the IEEE International Conference on Industrial Technology (ICIT)*, 2019.

[19] M. Xiong, J. Parsons, J. Edmondson, H. Nguyen, and D. C. Schmidt, "Evaluating the performance of publish/subscribe platforms for information management in distributed real-time and embedded systems," *omgwiki. org/dds*, 2010.

[20] R. Serrano-Torres, M. García-Valls, and P. Basanta-Val, "Performance evaluation of virtualized dds middleware," in *Simposio de tiempo real, Madrid*, 2014, pp. 18–19.

[21] D. Alexis, "Benchmark results of kubernetes network plugins (cni) over 10gbit/s network," https://itnext.io/benchmark-results-of-kubernetes-network-plugins-cni-over-10gbit-s-network-updated-april-2019-4a9886efe9c4, 2019.

[22] "K3s: Lightweight kubernetes," https://k3s.io/.

[23] "Kubeedge," https://kubeedge.io/en/.

[24] H. Fathoni, C.-T. Yang, C.-H. Chang, and C.-Y. Huang, "Performance comparison of lightweight kubernetes in edge devices," in *International Symposium on Pervasive Systems, Algorithms and Networks*. Springer, 2019, pp. 304–309.

[25] A. C. Beltrão, B. B. N. de França, and G. H. Travassos, "Performance evaluation of kubernetes as deployment platform for iot devices."

[26] S. Javadekar, "Kafka on kubernetes: From evaluation to production at intuit," 2018.

[27] I.-C. Donca, C. Corches, O. Stan, and L. Miclea, "Autoscaled rabbitmq kubernetes cluster on single-board computers," in *2020 IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR)*. IEEE, 2020, pp. 1–6.

[28] "Best practices for operating hivemq and mqtt on kubernetes," https://www.hivemq.com/best-practices-for-operating-hivemq-and-mqtt-on-kubernetes/.

[29] A. Buzachis, A. Galletta, L. Carnevale, A. Celesti, M. Fazio, and M. Villari, "Towards osmotic computing: Analyzing overlay network solutions to optimize the deployment of container-based microservices in fog, edge and iot environments," in *2018 IEEE 2nd International Conference on Fog and Edge Computing (ICFEC)*. IEEE, 2018, pp. 1–10.

[30] S. Qi, S. G. Kulkarni, and K. Ramakrishnan, "Assessing container network interface plugins: Functionality, performance, and scalability," *IEEE Transactions on Network and Service Management*, vol. 18, no. 1, pp. 656–671, 2020.

[31] M. A. Vieira, M. S. Castanho, R. D. Pacífico, E. R. Santos, E. P. C. Júnior, and L. F. Vieira, "Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications," *ACM Computing Surveys (CSUR)*, vol. 53, no. 1, pp. 1–36, 2020.

[32] "Project calico — tigera," https://www.tigera.io/project-calico/.