# Evaluating Performance of OMG DDS in Kubernetes Container Deployment (Industry Track)

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

OMG's Data Distribution Service (DDS) is an open, real-time publish/subscribe middleware standard, which has been widely adopted in many latency-sensitive and mission-critical industrial IoT applications. However, deploying and managing large-scale distributed DDS applications is tedious and laborious. As a successful container orchestration platform for distributed applications in the cloud, Kubernetes (k8s) is a promising solution for DDS-based systems. However, the feasibility of running DDS applications in a k8s environment, and the overhead of different k8s virtualization network architectures on DDS application performance has not been systematically studied. To address this, we compare the performance of DDS applications with several k8s network solutions using a comprehensive set of experiments we designed for a DDS benchmark application. Our experimental results reveal that: (1) the overhead of container virtualization is trivial for DDS applications; (2) the overhead imposed by virtual networks is not significant and there is not much performance difference between the experimented virtual networking solutions; (3) WeaveNet is useful for DDS discovery because it supports IP multicast, but its multicast performance is considerably lower than the host network.

***CCS Concepts:*** • **Software and its engineering** → **Publish-subscribe / event-based architectures**; • **Networks** → **Network measurement**; • **General and reference** → **Evaluation**.

***Keywords:*** Kubernetes, Container Networking, Data Distribution Service, Pub/Sub, Performance Evaluation


*Middleware '20, Dec 7-11, 2020, Delft, The Netherlands*


## 1 Introduction

The Industrial Internet of Things (IIoT) is a variant of the IoT for industrial sectors such as manufacturing, healthcare, energy and transportation. IIoT systems collect, store, and analyze fast-moving data streams generated by sensors in cloud/edge service layers for real-time system control and remote monitoring. As the complexity of IIoT tasks increases, completing an operation may require communication between multiple sensors and cloud/edge services, and require (sub)millisecond-level decisions to streaming events.

OMG's Data Distribution Service (DDS) is an open standardized middleware for real-time and distributed IIoT applications. DDS middleware abstracts the underlying logic of data distribution and management to simplify the development of IIoT applications. The event-driven design of DDS enables decoupled communications between processes by transmitting and processing events asynchronously. DDS also supports a fully distributed peer-to-peer communication model and uses a binary wire format protocol. This helps it to meet real-time performance and reliability requirements.

With the rapid growth in the scale of IIoT systems, it has become more challenging to use traditional methods to deploy and manage large-scale distributed applications. Container technologies have been successful in solving scalability challenges in the cloud. Likewise, we believe that container technologies can also be leveraged for IIoT applications to transform the way they have been managed in the past. Containers enable lightweight encapsulation and resource isolation of functional modules, decoupling applications from the underlying platform. Therefore, containers allow applications to be easily tested and deployed across platforms using standardized container image formats.

Kubernetes (k8s) is the de-facto standard for orchestrating containerized workloads. With k8s, it is easy to deploy, update, scale, and self-heal distributed applications. Considering the advantages of DDS and k8s in rapid and scalable deployment of real-time applications, companies are motivated to use k8s to manage their DDS-based applications. However, while k8s has proven to be an effective and successful solution for orchestrating cloud applications, its robustness and performance in

managing distributed real-time embedded applications has not yet been validated. In addition, k8s supports a variety of pluggable virtual network solutions, and they are implemented in different ways. Accordingly, the performance with containers and virtual networks for real-time middleware such as DDS is unknown.

To this end, this paper 1) provides a guide for deploying DDS applications in a k8s cluster; 2) develops an automated benchmark tool for evaluating the performance of DDS applications under various workloads and DDS QoS configurations; and, 3) carries out a systematic set of experiments to quantitatively demonstrate the performance differences of DDS applications with k8s.

## 2 Background on Underlying Technologies

### 2.1 OMG DDS

OMG's DDS standard defines a data-centric, publish/-subscribe (pub/sub) connectivity framework for real-time and embedded systems. DDS is designed to meet the performance, scalability, fault-tolerance, and security requirements of real-time and embedded systems. DDS has been adopted in mission critical applications across a range of vertical domains including healthcare, energy, transportation, aerospace and defense, such as ROS[14], FACE[7], OpenFMB[12], etc.

The DDS pub/sub interaction model promotes loose coupling between applications with respect to time (i.e., the applications need not be present at the same time) and space (i.e., applications may be located anywhere). The core concept of DDS is data-centricity. With a data-centric middleware, the means of interaction is data. The middleware understands the structure of the data it manages and it is aware of the contents (i.e., values). This enables DDS to extensively optimize performance.

DDS also provides configurable Quality of Service (QoS) policies. 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 configurable QoS policies, developers can easily define and update desired behaviors of applications.

### 2.2 Kubernetes

Kubernetes (k8s) is the de-facto standard orchestration platform for containerized applications. An orchestration platform is a set of system services that deploy and manage distributed applications. Specifically, it helps manage the applications by scaling them up and down, performing updates and rollbacks, self-healing, etc.

The deployable unit of k8s is a pod. A pod is a collection of one or more containers with shared storage and network. Containers in a pod share an IP address and they can communicate with each other over shared memory or localhost interface.

### 2.3 DDS in the Context of Kubernetes Networking

k8s uses Docker as a default container engine, but its approach to networking is different from what Docker does by default. In a k8s cluster, every pod gets its own directly accessible IP address, and therefore it does not require users to deal with mapping ports between containers. The k8s networking model[3] creates a clean, backward-compatible model where pods can be treated much like physical hosts. The 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.

The k8s networking model is a better fit for DDS than Docker alone. DDS participants exchange their IP addresses for peer-to-peer communications, and therefore DDS works better over a network without NAT. DDS can help pods discover each other. Pods have unreliable IP addresses, as their addresses are dynamically assigned when created. Because of that, 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. With the DDS discovery service, pods can discover and establish connections with each other by topics, abstracting IP-based communications. This allows DDS pods to discover and communicate without a k8s service, resolving the IP unreliability issue. DDS uses multicast for the discovery service. If a k8s networking plugin supports multicast, DDS discovery can work without an external discovery service such as RTI Cloud Discovery Service (CDS)[10].

### 2.4 Kubernetes Virtual Networking Solutions

k8s provides a plug-in networking interface called Container Network Interface (CNI), which supports a variety of virtual networking technologies. The most popular ones are Calico, Flannel, WeaveNet, and Cilium. Virtual networks provide benefits such as network traffic isolation, dynamic segmenting and routing, and rapid deployment and update. However, this flexibility comes at a cost – added overhead.

Virtual networking solutions can be implemented at Layer 2 (L2) or Layer 3 (L3). Flannel and WeaveNet are realized at L2, Calico is implemented at L3, and Cilium supports both. L2 solutions are usually based on VXLAN tunnel technology that creates virtual bridges between containers and the physical host. The virtual

bridge realizes cross-L3 container network connectivity by encapsulating L2 Ethernet frames in L3 UDP datagrams. Therefore, encapsulation is conceptually the primary overhead of L2 class solutions.

Flannel is the simplest L2 solution. It creates an overlay network in a k8s cluster and assigns a subnet to each physical node. Each pod running in the physical node has a globally unique virtual IP address, which results in a flat network space where pods can directly communicate with each other without needs to map ports between containers and hosts. VXLAN encapsulation is handled by a virtual bridge called flannel0. An essential component of the Flannel network stack is the flanneld process, which uses etcd (a distributed key-value store) to 1) manage available IP address resources, 2) monitor the real IP address of each pod, and 3) establish the pod-to-node routing table in memory. The egress traffic of a pod is forwarded from cni0 to flanneld through flannel0, where cni0 is a bridge maintained by k8s through CNI. Flanneld then encapsulates and propagates packets to target flanneld processes through the physical network.

WeaveNet fully emulates a L2 network whose topology is built by application-level routers located on each host. WeaveNet routers establish TCP connections with each other for protocol handshakes and exchanging topology information. Like Flannel, the routers encapsulate Ethernet frames in L3 UDP datagrams for cross-L3 communication. In contrast to the centralized node discovery strategy adopted by Flannel, spanning tree and gossip protocols are used to share routing information among routers. WeaveNet also creates a bridge on the host to which workload containers and WeaveNet routers connect. WeaveNet supports multicast, and therefore does not require an external discovery service for k8s-based DDS applications. Also, in multi-subscriber scenarios, WeaveNet can provide higher throughput because data can be delivered to multiple subscribers with no additional overhead on the publisher's side.

Calico is a L3 solution that avoids VXLAN encapsulation overhead using Border Gateway Protocol (BGP). A Linux kernel-based virtual router is provided on each physical node, which learns the topology of the container network by exchanging routing and accessibility information with other endpoints in the domain. BGP solves the problem of exponential growth in routing rules with growing cluster size by reducing the size of the routing table (route aggregation) and number of connections among endpoints (route reflection). Calico also provides IP-in-IP tunnel mode for container communication across subnets, which encapsulates the L3 data packet into another IP packet. Encapsulation and decapsulation are performed by a newly created tunl0 interface, which resembles the veth interface in VXLAN.

Cilium supports both L2 and L3 networking modes, and performs packet filtering in kernel space using extensible Berkeley Packet Filter (eBPF) programs, which are generated by the bytecode injected from user space and attached to specific kernel areas. Compared with the iptables-based packet filtering in kube-proxy, BPF eliminates the overhead of (1) copying unexpected traffic into memory and executing high-level protocols, (2) traversing and modifying iptables when pods are created or destroyed, which significantly improves the scalability of the Cilium network. Moreover, using eBPF, Cilium can perform fine-grained layer 7 filtering, which can potentially support DDS topic-based/content-based filtering at the virtual network layer for better performance.

## 3 Evaluating DDS Performance in Kubernetes

In this section, we benchmark DDS applications in a k8s cluster with the purpose to understand the performance overhead introduced by containers and virtual networks. We first compare the performance between DDS applications running directly on bare metal and by running in containers using the host network. This provides a baseline for subsequent experiments and shows the overhead of container virtualization. Then, we measure the throughput and latency of DDS applications with host network, Calico, Flannel, WeaveNet, and Cilium in unicast mode. We also compare the multicast performance of DDS using WeaveNet and the host network.

### 3.1 Experimental Setup and Configurations

Figure 1 represents the deployment setup for our experiments. The k8s cluster for our experiments is composed of one master node and two worker nodes. Each node is equipped with an Intel i7-5557U dual-core 3.10 GHz processor, 16GB RAM, and Intel I128-V Gigabit Ethernet device. All nodes have Ubuntu 16.04 (64bit) installed and are connected via a 1 Gbps LAN. We use the default MTU size (1500 bytes) of each network device.

Our experiment automation apparatus comprises a manager application executing on the k8s master node that interacts with the k8s cluster through k8s API and controls the execution progress of the overall testing plan. At the beginning of an experiment, the manager runs a k8s deployment of RTI's Cloud Discovery Service (CDS) for discovery and creates a corresponding ClusterIP Service to enable the DDS publisher and subscriber to discover each other via a configured DNS name (e.g., rti-clouddiscoveryservice:7400). Once RTI CDS is running, the manager deploys DDS performance testing applications in k8s pods (see Figure 1).

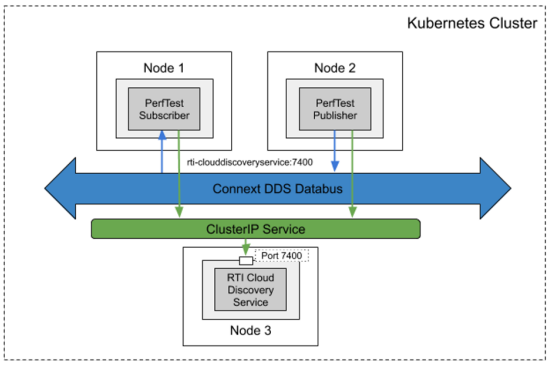



**Figure 1.** Setup for Performance Experiments

We use RTI's benchmarking tool called PerfTest[9], which is a command-line application that can be configured as a publisher or subscriber. It is used to measure the latency and throughput of configurable scenarios using RTI's DDS implementation called Connext DDS.

PerfTest has two operational modes: Latency Test and Throughput Test. Throughput is calculated by the subscriber by counting the number of received bytes and samples. PerfTest behaves differently in the Latency Test mode: all samples are marked as latency samples, and samples are exchanged in a ping-pong manner. The publisher is blocked until being acknowledged by a replied latency sample. Once the publisher receives a replied latency sample, it calculates the one-way latency from the measured round trip time. The primary advantage of this measurement method is that it is not subjected to system clock jitters across different machines.

We optimized the OS network performance based on recommended OS configurations from the RTI community[1]. The DDS reliability protocol automatically applies back-pressure to the publisher when the subscriber cannot keep up. Back-pressure happens when the sendQueueSize is filled with unacknowledged samples. When the sendQueueSize is larger, less back-pressure occurs, which leads the publisher to send samples faster than the subscriber can keep up. We found that the default sendQueueSize adversely impacts the throughput performance of virtual networks as most packets are dropped at the subscriber side's network bridge. To resolve the issue, we reduced the sendQueueSize to 20 to slow down the publisher sooner when the subscriber cannot keep up. In addition, batching is enabled for the throughput test and disabled for the latency test. If the data length is less than 8192 bytes, the data is padded to maintain the batch size at 8192 bytes, otherwise it is equal to the data length.

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

### 3.2 Experimental Results

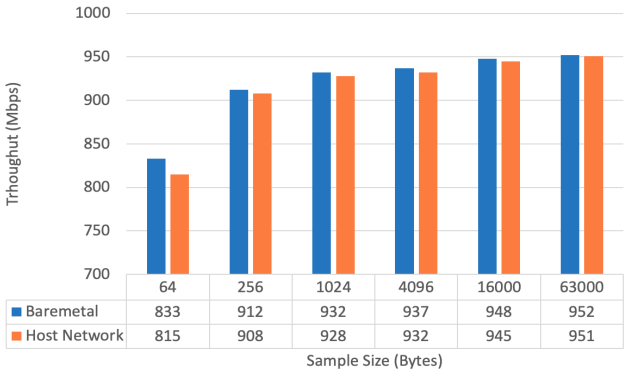
**3.2.1 Container Performance.** We measured throughput and latency on bare metal and containers to understand the potential overhead introduced by container virtualization. For the host network case, we run k8s pods using the host network interface directly to avoid virtualization for the container network. Figure 2 shows that a container introduces some overhead for DDS applications, which is more pronounced for small messages.

Work [11] illustrates that a container consumes more CPU cycles for I/O operations than native Linux by two times due to the interference of the Linux network bridge. The publication rate is unlimited, so it is controlled by the back-pressure incurred by the subscriber side. The test with small messages is more CPU-intensive, which explains the higher performance degradation at 64 bytes compared to other cases. But overall, throughput overhead is less than 3% on average.

On the other hand, there is little or no difference in latency performance. The reason is that PerfTest uses ping-pong to test end-to-end network latency and batch processing is disabled by default. Therefore, there is no data backlog in the DDS waiting queue and socket buffer of the container and bare metal. The time difference on processing a single message is trivial. For 99 percentile latency, it is increased by 8-25 microseconds.

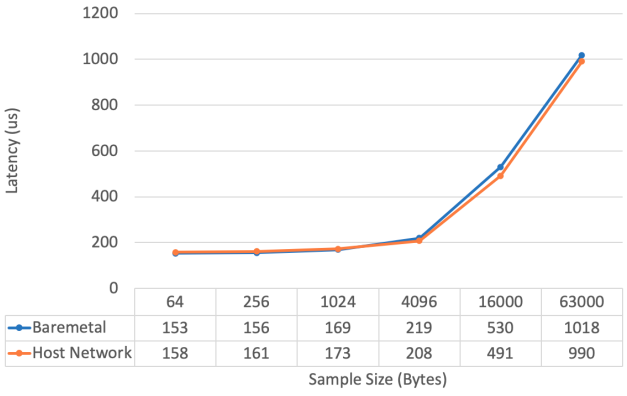
**3.2.2 Virtual Networking Performance.** In the following set of experiments, we measured throughput and latency on the host network and a set of virtual networks to measure the overhead introduced by k8s virtual networking. It should be noted that we use the default settings of each virtual network solution because our purpose is to present performance of each for general use cases. As shown in figure 3, there is not much performance difference among virtual networks. Calico performed slightly better than others in terms of throughput. Throughput overhead ranged from 2% to 7%. Flannel gained the best throughput performance for a small message sizes (3% overhead). Calico performed the best for large message sizes (2% throughput overhead). Cilium did not work reliably for large samples. For small messages, latency increased from 37 to 69 microseconds for median and 47-74 microseconds for 99 percentile. Calico performed the best (increased by only 37 microseconds for median). For large messages, latency increased from 59-255 microseconds for median and 0-235 microseconds for 99 percentile. Cilium performed the best (increased by 59 microseconds for median).

**3.2.3 Multicast Performance.** Multicast is beneficial to pub/sub applications since messages can be delivered to multiple receivers simultaneously, thereby significantly



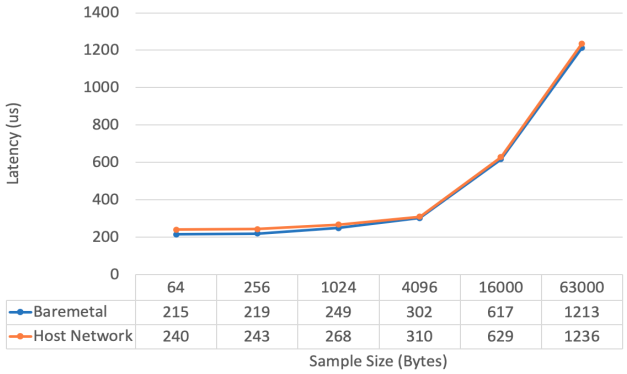

| | 64 | 256 | 1024 | 4096 | 16000 | 63000 |
|---|---|---|---|---|---|---|
| Baremetal | 833 | 912 | 932 | 937 | 948 | 952 |
| Host Network | 815 | 908 | 928 | 932 | 945 | 951 |

**(a)** Throughput (Mbps)




| | 64 | 256 | 1024 | 4096 | 16000 | 63000 |
|---|---|---|---|---|---|---|
| Baremetal | 153 | 156 | 169 | 219 | 530 | 1018 |
| Host Network | 158 | 161 | 173 | 208 | 491 | 990 |

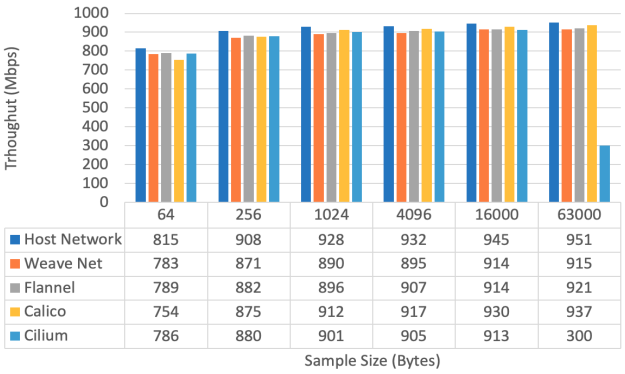
**(b)** Median Latency in Microseconds




| | 64 | 256 | 1024 | 4096 | 16000 | 63000 |
|---|---|---|---|---|---|---|
| Baremetal | 215 | 219 | 249 | 302 | 617 | 1213 |
| Host Network | 240 | 243 | 268 | 310 | 629 | 1236 |

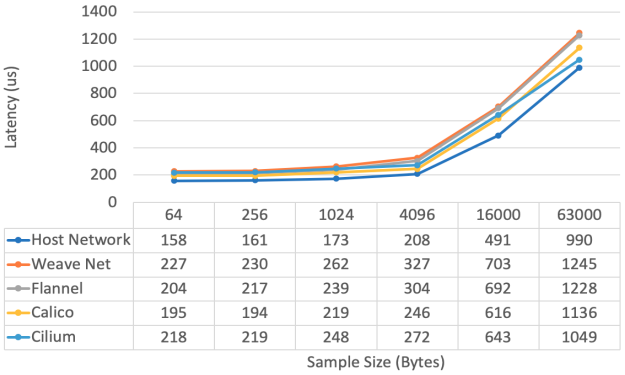
**(c)** 99-percentile Latency in Microseconds

**Figure 2.** Bare Metal vs. Container: One Publisher and One Subscriber with Unlimited Publication Rate. Each test was run for 60 seconds and repeated 3 times.



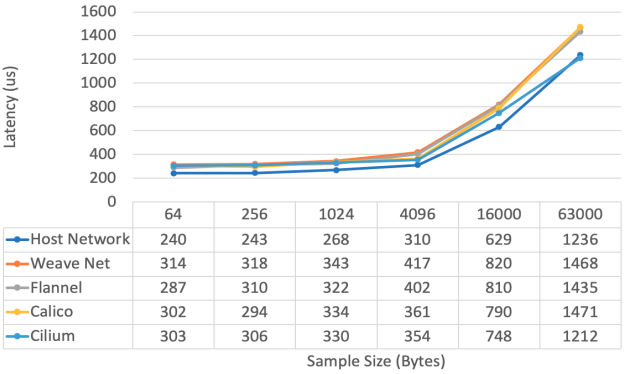

| | 64 | 256 | 1024 | 4096 | 16000 | 63000 |
|---|---|---|---|---|---|---|
| Host Network | 815 | 908 | 928 | 932 | 945 | 951 |
| Weave Net | 783 | 871 | 890 | 895 | 914 | 915 |
| Flannel | 789 | 882 | 896 | 907 | 914 | 921 |
| Calico | 754 | 875 | 912 | 917 | 930 | 937 |
| Cilium | 786 | 880 | 901 | 905 | 913 | 300 |

**(a)** Throughput(mbps)




| | 64 | 256 | 1024 | 4096 | 16000 | 63000 |
|---|---|---|---|---|---|---|
| Host Network | 158 | 161 | 173 | 208 | 491 | 990 |
| Weave Net | 227 | 230 | 262 | 327 | 703 | 1245 |
| Flannel | 204 | 217 | 239 | 304 | 692 | 1228 |
| Calico | 195 | 194 | 219 | 246 | 616 | 1136 |
| Cilium | 218 | 219 | 248 | 272 | 643 | 1049 |

**(b)** Median Latency in Microseconds




| | 64 | 256 | 1024 | 4096 | 16000 | 63000 |
|---|---|---|---|---|---|---|
| Host Network | 240 | 243 | 268 | 310 | 629 | 1236 |
| Weave Net | 314 | 318 | 343 | 417 | 820 | 1468 |
| Flannel | 287 | 310 | 322 | 402 | 810 | 1435 |
| Calico | 302 | 294 | 334 | 361 | 790 | 1471 |
| Cilium | 303 | 306 | 330 | 354 | 748 | 1212 |

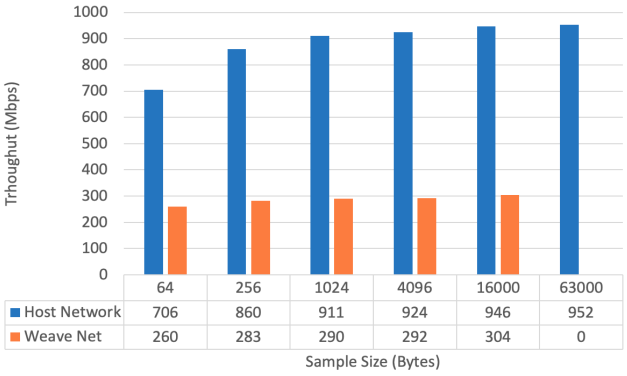
**(c)** 99-percentile Latency in Micoseconds

**Figure 3.** Host Network vs. Virtual Networks: One Publisher and One Subscriber with Unlimited Publication Rate. Each test was run for 60 seconds and repeated 3 times.

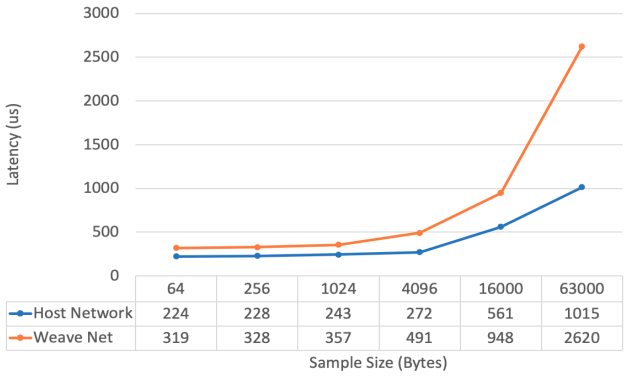
improving throughput and system scalability. Only WeaveNet supports IP multicast. Figure 4 demonstrates multicast performance of a host network vs WeaveNet. For messages smaller than 16KB, compared with unicast, the throughput of host network and WeaveNet in the multicast degrades 0.8%-13.4% and 66.8%-67.5%, respectively, while their median latency increases 30.8%-41.8% and 36.3%-50.2%, respectively. The throughput of WeaveNet multicast climbs from 260 Mbps to 304 Mbps as the message size increases from 64B to 16KB, then suddenly drops to 0 when message size is 63000 bytes. Comparing to host network, WeaveNet multicast suffers severe performance degradation due to VXLAN encapsulation.

### 3.3 Discussion

Based on the results of our experiments, we summarize our findings:




**(a)** Throughput(mbps)




**(b)** Median Latency in Microseconds

**Figure 4.** Multicast Performance Comparison: One Publisher and Two Subscriber with Unlimited Publication Rate. Each test was run for 60 seconds and repeated 3 times.

1. In general, the overhead of container virtualization is minimal for DDS applications, but it is non-negligible when the application is CPU-intensive;
2. In unicast-based pub/sub scenarios, the performance overhead by virtual networks is not significant, and there is not much difference among them. Among virtual networks, we recommend using Flannel for small messages; Calico is a reliable choice when publishing large messages.
3. As WeaveNet supports multicast, it can be useful for DDS discovery. However, its multicast performance is significantly lower than the host network. If the underlying network supports a large MTU, performance can be potentially improved by increasing the MTU size[4]. However, if multicast performance is critical and there are no strict security requirements to isolate container traffic, using the host network for multicast is recommended.

## 4 Related Work

Several efforts have studied the performance of DDS and k8s separately. Related to DDS, existing efforts usually compare DDS with other IIoT middleware horizontally. For instance, [6] compares the latency, bandwidth consumption and packet loss of DDS, MQTT, CoAP and a custom UDP application under a constrained wireless access network. Likewise, [2] includes more middleware protocols, such as MQTTSN, AMQP, and XMPP. This work [8] surveys multiple middleware protocols including DDS based on their primary characteristics and potential performance issues including throughput, latency, and energy consumption. Similarly, [13] 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.

Longitudinal studies, such as [16] investigated three popular DDS implementations comparing their architectures and performance. In [15], authors explored the overhead and side-effects of a variety of VM-based virtualization methods for distributed systems using DDS.

Related to k8s, [1] 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] for the testing application. The authors in [17] measured and 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 [5] is similar to ours; they analyzed the performance overhead of OVN, Flannel, WeaveNet, and Calico on CoAP and FTP applications, while we focused on real-time DDS applications, and conducted multicast testing.

## 5 Conclusion

In this paper, we validated the feasibility of deploying DDS applications with k8s and explained the k8s components and operations through a workflow. Second, we qualitatively analyzed the overhead of four mainstream k8s network plug-ins: Flannel, Calico, WeaveNet and Cilium. Finally, we demonstrated their performance (throughput and latency) differences quantitatively by executing a systematic set of DDS benchmarking tests under a variety of workload patterns.

Our future work includes: (1) extending our experiments to advanced network functions supported by k8s network plugins and more complex DDS QoS configurations, and (2) applying k8s network policies to DDS applications to implement network-level packet filtering for discovery scalability.

[2] https://iperf.fr/

## References


[1] Ducastel Alexis. 2019. 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.

[2] M Anusha, E Suresh Babu, LS Mahesh Reddy, AV Krishna, and B Bhagyasree. 2017. Performance analysis of data protocols of internet of things: a qualitative review. *International Journal of Pure and Applied Mathematics* 115, 6 (2017), 37–47.

[3] The Kubernetes Authors. 2020. Kubernetes Cluster Networking. https://kubernetes.io/docs/concepts/cluster-administration/networking/#the-kubernetes-network-model.

[4] Bryan Borham. 2015. Weave Networking Performance with the New Fast Data Path. https://www.weave.works/blog/weave-docker-networking-performance-fast-data-path/.

[5] Alina Buzachis, Antonino Galletta, Lorenzo Carnevale, Antonio Celesti, Maria Fazio, and Massimo Villari. 2018. 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, 1–10.

[6] Yuang Chen and Thomas Kunz. 2016. Performance evaluation of IoT protocols under a constrained wireless access network. In *2016 International Conference on Selected Topics in Mobile & Wireless Networking (MoWNeT)*. IEEE, 1–7.

[7] The Open Group Future Airborne Capability Environment Consortium. 2020. Future Airborne Capability Environment (FACE™). https://www.opengroup.org/face.

[8] Jasenka Dizdarević, Francisco Carpio, Admela Jukan, and Xavi Masip-Bruin. 2019. A survey of communication protocols for internet of things and related challenges of fog and cloud computing integration. *ACM Computing Surveys (CSUR)* 51, 6 (2019), 1–29.

[9] Real-Time Innovations. 2019. RTI_Perftest 3.0 documentation. https://community.rti.com/static/documentation/perftest/3.0/index.html.

[10] Real-Time Innovations. 2020. DDS Discovery in Cloud-Based Environment. https://www.rti.com/developers/rti-labs/discover-data-in-cloud-services-with-cloud-discovery-service.

[11] Kyungwoon Lee, Youngpil Kim, and Chuck Yoo. 2018. The impact of container virtualization on network performance of IoT devices. *Mobile Information Systems* 2018 (2018).

[12] Open Field Message Bus (OpenFMB). 2020. OpenFMB. https://openfmb.ucaiug.org/.

[13] Stefan Profanter, Ayhun Tekat, Kirill Dorofeev, Markus Rickert, and Alois Knoll. 2019. 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)*.

[14] The Robot Operating System (ROS). 2020. ROS.org. https://www.ros.org/.

[15] Rosbel Serrano-Torres, Marisol García-Valls, and Pablo Basanta-Val. 2014. Performance evaluation of virtualized DDS Middleware. In *Simposio de tiempo real, Madrid*. 18–19.

[16] Ming Xiong, Jeff Parsons, James Edmondson, Hieu Nguyen, and Douglas C Schmidt. 2010. Evaluating the performance of publish/subscribe platforms for information management in distributed real-time and embedded systems. *omgwiki.org/dds* (2010).

[17] Hao Zeng, Baosheng Wang, Wenping Deng, and Weiqi Zhang. 2017. Measurement and evaluation for docker container networking. In *2017 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)*. IEEE, 105–108.