Implementing a Network-Aware Kubernetes Scheduler on top of a Mesh Network

Jerico Moeyersons, Brecht Stamper, Bruno Volckaert and Filip De Turck
IDLab, Department of Information Technology
Ghent University - imec, Ghent, Belgium
Email: jerico.moeyersons@ugent.be

***Abstract*—A recent trend observed in the Kubernetes world is trying to deploy a Kubernetes cluster entirely or partially to the edge. When further looking into these edge environments, it is often noted that wireless technologies are used. Kubernetes is not designed to support these wireless network setups and will have to be extended to run smoothly on these networks. One of the most used network setups in edge environments is a wireless mesh network. Therefore the main focus will be on running Kubernetes on top of a wireless mesh network. Other mesh networks will also be supported with a minimal set of changes needed. The main goal of this paper is to list all the problems encountered when running Kubernetes on top of a mesh network and provide a framework of components and extensions to solve these problems. The majority of these components will be implemented, and demonstrations show that these components are able to solve most of the listed problems when used in a setup with some predefined restrictions. When deploying a demo application with the proposed solution, the required bitrate is observed 97 percent of the time compared to 42 percent with the native Kubernetes scheduler.**

***Index Terms*—Kubernetes, network, mesh network**

I. Introduction

With the recent trend of deploying Kubernetes to the edge, a lot of challenges of these deployments are discovered. Some of them are for example running Kubernetes on low-end hardware, network setup and how to manage a highly distributed cluster. This paper will focus on the network setup of a wireless mesh network. Running Kubernetes on top of a wireless mesh network can be done without any modifications but will have a lot of problems. The goal of this paper is to give an overview of these problems and propose a framework of components to solve these problems.

A Wireless Mesh Network (WMN) is a communications network made up of radio nodes organized in a mesh topology [1]. In a traditional WMN, mobility of the nodes is expected to be relatively infrequent. It consists of three types of nodes: mesh routers, mesh gateways, and mesh clients. The mesh routers are responsible for routing and forwarding network packets to their destination. Mesh clients connect to mesh routers and can use the network without knowing the underlying mesh topology. Mesh gateways are mesh routers that connect the WMN to other networks, such as the public internet.

The main benefits of using a wireless mesh network are (1) all devices can communicate with each other, (2) self-organization and configuration of the network, (3) the ability to still work if one or some nodes fail, and (4) gateways support external connectivity. The main downsides are (1) latency/bandwidth dependent on distance and hops, which is dynamic, and (2) shared medium access problems.

In order to extend Kubernetes, some native extensions points were used, together with a modification to the internal routing of a service resource. Kubernetes services are routed with iptables rules, these are configured by the kube-proxy (more options are available but iptables is used as the default one). The used extensions points are: replacing or adding Kubernetes controllers, adding custom resources through Custom Resource Definitions (CRDs) and extending the scheduler controller through the scheduler framework. Extensions that use this framework are called scheduler plugins. This allows the scheduler to be easily extended without having to write a new one and also supports multiple running plugins at the same time.

The remainder of the paper is organized as follows. In Section II, related work is discussed. Section III will give an overview of the encountered problems and propose new components to solve these problems. In Section IV, the previously discussed components will be implemented and discussed in detail. Then, in Section V, the proposed component setup will be evaluated, followed by the discussion on which additions are still required the support a fully dynamic mesh network. Finally, some conclusions are made in Section VI.

II. Related Work

In this section, the related work for this article is discussed in two parts. In part one, network-aware routing extensions in Kubernetes are discussed, followed by some of the more promising papers regarding the end-to-end available throughput in a mesh network in part two. The research was split into these two parts since no papers were published concerning running Kubernetes on a wireless mesh network at the time of writing.

A. Network aware routing in Kubernetes

In towards network-aware resource provisioning [2] the Kubernetes scheduler gets extended to take network bandwidth and latency into account. The supported environment is a static network with predefined available bandwidth. The available bandwidth is only dependent on this predefined value after subtracting the required bandwidth used by the pods already

Final article can be found here: 10.1109/CloudSummit54781.2022.00018

deployed on that node. It only takes required bandwidth into account when scheduling and does not enforce this later on. The main focus of this paper is reducing the network latency observed by deployed workloads. In this article, the main focus will be on providing the requested bitrate and handling this in a mesh network. Because of this, the proposed solution of this paper was not used. However, the idea of abstracting most logic out of the scheduler was used, this is to make the scheduler as fast and straightforward as possible. This also has the added benefit that separate components can be replaced in the future without having to remake the scheduler.

In a network-aware scheduler plugin [3], a plugin is proposed to extend the behavior of the scheduler to take network information into account when scheduling. The environment supported here is more extensive than the previous static one. It adds support to store network weights between multiple regions and zones by implementing a network topology CRD. Bandwidth is also supported through the use of extended resources, a native Kubernetes implementation of exposing node-specific resources. Once pods are scheduled, the decisions made by the scheduler are not persisted and enforced. This could lead to unforeseen problems in the future. The bandwidth requirements are also only node ingress and egress specific and not more detailed such as node-to-node or node-to-zone. This paper was used as the starting point of the proposed component framework. Therefore a more detailed look at what was used and what was not will be given next. The first main similarity is the use of the scheduler framework to make a scheduler plugin, this is currently the recommended way to extend the scheduler, and the other approaches are discouraged. The second similarity is the use of a CRD to abstract the network topology. In this paper, the network topology CRD is used to store network info between multiple zones and regions. A similar approach will be used in this article and will store network metrics between nodes within the same zone. This allows the proposed solution to be partially merged in the future and support both static networks combined with mesh networks while still using a different optimized approach for both use cases. The paper also introduces an appgroup CRD to specify all the network requirements between nodes. This CRD was not used since it requires modeling the entire application data flow between all components. While we see the benefits in this approach for more traditional server deployments, we also think this approach is too complicated and restrictive for edge deployments. Therefore, a new CRD will be proposed that is easier to use and supports mesh routing.

Wojciechowski et al. [4] propose a network metrics-aware Kubernetes scheduler (NetMARKS) powered by the Istio service mesh. The underlying network is a static one connected with cables. Instead of predefined bitrate requirements between multiple components, the information gathered by the Istio proxy is used. All pod traffic passes through at least one of these proxies and allows the scheduler to calculate the bitrate requirements dynamically. The biggest downside of this approach is that pods need to be deployed before the bitrate requirements are known. This introduces a problem in a mesh network where there is possibly a high chance that the initial deployment of pods will not offer the required bitrate. In this case, the metrics gathered by Istio will not reflect the required bitrate but the available bitrate. This approach, however, looks very promising when combined with the proposed solution. An initial bitrate requirement needs to be set in the proposed solution, but once the initial deployment is done, a shift could be made to use these dynamic metrics. This would allow the user to set safe initial bitrate requirements, which would then change over time to reflect the actual bitrate requirements.

B. End-to-end available throughput in a mesh network

Sarr et al. [5] propose a passive approach to estimate available bandwidth between neighboring nodes has been proposed. It does this by monitoring channel usage, probabilistically combining these values to take synchronization error into account, and finally estimating collision probability between each pair of nodes. This information is then spread out in the network through to use of hello packets in the AODV routing algorithm. However, it could easily be implemented similarly in a different routing algorithm. This approach is then later extended to support multi-hop bandwidth estimation. The greatest difficulty in this approach is the need for channel usage metrics. These metrics are not naively exposed by the Linux kernel, and driver support will need to be added. More research is also required to see if this approach actually delivers the promised results and in which scenarios it does not.

Venkatesh et al. [6] propose a modification to low-overhead packet probing bandwidth estimation techniques to accurately estimate end-to-end throughput while still maintaining the original low-overhead and fast convergence of the original algorithms. The results look promising but are obtained by running tests in a simulated and controlled environment. Further analysis would be needed before fully committing to this algorithm.

This is not only the case for this paper but for most papers published in this research topic [7]–[9]. They all suggest a new bandwidth estimation technique or modification and then continue to show some promising results in a simulated controlled environment. However, most of them fail to deliver real-world results and prove that their approach also works in an uncontrolled environment. Therefore non of these approaches were used in this article.

One of the more recent trends observed is estimating throughput through the use of artificial models. Seeing how accurately these models are able to predict values in all kinds of research fields, especially in very complex ones, their usage seems ideal here. Munaye et al. [10] propose a deep learning-based model to predict throughput in a UAV-assisted network. Their model manages to estimate the actual throughput with high accuracy and could potentially do the same in other similar problems, such as estimating the throughput in a mesh network. Samba et al. [11] propose a predictor that can instantaneously predict the achievable bitrate (with reasonable accuracy) in a cellular network between the operator and a

mobile node. This environment shares many problems with a mesh network, such as a shared medium and dynamic moving nodes. The main one it does not share is having to support multi-hop paths. However, it is reasonable to say that this approach could also work in a mesh network. Overall this new trend seems very promising and could potentially solve this problem in the near future.

III. System Overview

This section is split into two parts. The first part will give an overview of the problems generated by the setup and explain the origin of these problems. The second part will then list the new components needed to solve these problems. The details and actual implementation of these components are explained in Section IV.

A. Problems

This section explains the most important problems encountered in this article. Some of these problems are inherent to mesh networks, while others are only encountered when trying to run Kubernetes on a mesh network. The goal of this section is to give a clear understanding of what problems the following sections try to solve and why these problems are encountered in the first place.

1) Bandwidth and latency metrics: Gathering latency metrics is very easy and has almost no overhead on the network, but estimating throughput metrics on the other side is the complete opposite [5]–[7], [12]–[16]. This problem is then made more difficult by specifying the underlying network to be a multi-hop wireless mesh network.

2) Bandwidth requirements: When taking into account the available bandwidth, the amount of this bandwidth that will be used by our application needs to be known as well. In its most simple form, pods will send data to each other at a constant bitrate that never changes. In a real world deployment however, this bitrate would be dynamic and would depend on a combination of all inputs.

3) Dynamic environment: When nodes have the ability to move, it is required to check if all requirements are still met periodically. In case they are not, appropriate action has to be taken. In almost every real-world environment, every static environment becomes a dynamic one because of interference. What this means when looking at statistics is that a static environment can almost immediately be skipped. So from now on, when looking at a static environment, we mean a stationary setup with no moving nodes. The exact definition of this static environment will be specified in more detail later.

4) Service load-balancing: In Kubernetes, services are wildly used and one of the essential building blocks in a micro-service architecture. One of the main goals of this article is to be as hidden as possible for the end-user. Therefore it was a requirement not to change the outer working of services too much. The inner workings, however, have been modified. This was done to disable load-balancing where needed. Load-balancing at first seems like a useful feature to enable, but setting static pod-to-pod routes is required to maintain the bitrates needed between these pods. If the outgoing requests are load-balanced, they will be routed to a random node every time from a network point of view instead of to the same node every time. Routing is required to be predictable and static between nodes to calculate the required bandwidth and check if enough bandwidth is available.

B. Required components to solve these problems

Four new components will be introduced in Section IV, and the native scheduler is extended to solve the above described problems. The components can be split into two groups, one group responsible for gathering metrics and one group responsible for setup and enforcing network routing. In the first group, there is a mesh topology controller responsible for requesting and updating bandwidth and latency metrics. And a mesh agent which runs on every node and is controlled by the topology controller. This component will be responsible for getting the actual metric and returning this metric to the controller. Dividing the agent from the controller allows us to center all the logic in the controller and have minimal agents. This clear separation between logic and the way metrics are gathered, allows us to easily swap this agent out for a different one and support multiple types of mesh networks and bandwidth test implementations. In the second group, there is a mesh connection controller responsible for setting up and maintaining all routes within the mesh network we are able to control. And a mesh proxy to enforce these routes in the background while still appearing to work as a normal service to the end-user. Lastly, the scheduler will use the information provided from both the mesh topology controller and mesh connection controller to decide on which node a pod can be scheduled, taking into account the network requirements.

IV. Implementation

This section explains the implementation of the components introduced in the previous section in depth. The hardware setup and test environment used are explained first. Then every component is discussed in detail. An overview of all components can be seen in Figure 1.

A. Hardware setup and test environment

Four Raspberry Pi's 4 model b were used to make a mesh network. The specifications for these RPis can be found in Table I. The built-in wireless module will be used to set up the mesh network. The RPis are set up with the default Raspberry Os image running Debian Bullseye 64 bit. A Better Approach to Mobile Ad-hoc Networking Advanced version IV (BATMAN advanced IV or batman-adv IV) was used to interconnect these RPIs. BATMAN [17] is a routing protocol for multi-hop mobile ad-hoc networks operating on network layer 2. Batman-adv [18] is the Linux kernel implementation of this routing protocol. This routing protocol was chosen for ease of implementation while still offering competitive performance compared to other routing protocols [19]. From this point on, the RPis are referred to as nodes, the term used by Kubernetes to refer to a virtual or physical machine running

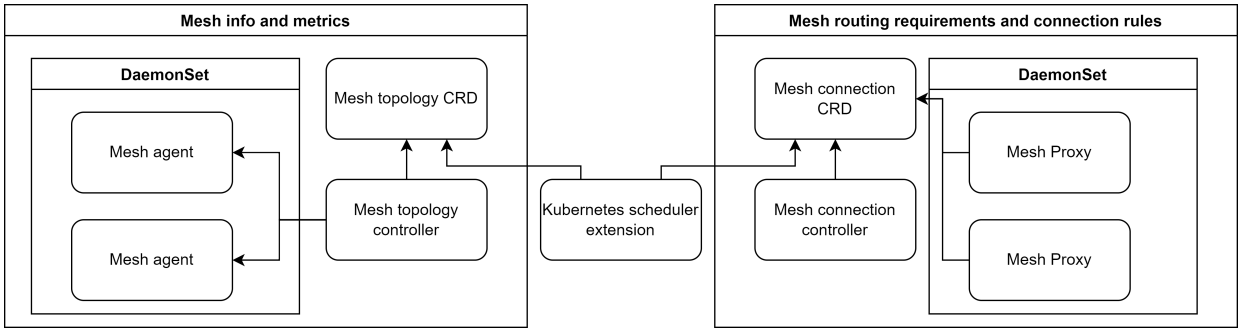

Fig. 1. Overview of all components

TABLE I
SPECIFICATION OF USED RASPBERRY PI

Raspberry Pi 4 model b 2GB specification	
Processor	Broadcom BCM2711, 64-bit SoC @ 1.5GHz
Memory	2GB LPDDR4-3200 SDRAM
Connectivity	2.4 GHz and 5.0 GHz IEEE 802.11ac BCM4345/6, Bluetooth 5.0 BLE, Gigabit Ethernet
Input Power	5V DC via USB-C connector (minimum 3A*)

workloads. Because these nodes are deployed in a home environment and not an isolated test environment, external interference will also play a big role.

B. Components

Four components, an extension to the native scheduler and two custom resource definitions (CRDs) are required to support running Kubernetes on a mesh network. This subsection will look at these components and CRDs, and give a detailed explanation of how they were designed and what their requirements are. Since a CRD is only used to store data and its goals are tightly coupled to its accompanying controller, each CRD will be introduced together with its accompanying controller.

1) ***Mesh topology controller****:* This controller, together with the mesh topology CRD, is responsible for storing, creating, and updating all the needed metrics for every mesh topology. From the start, support was added to have multiple nodes in different mesh networks. Every mesh topology is defined by a corresponding Kubernetes zone. In a typical Kubernetes setup, nodes are grouped in regions and zones. This info might then be used by plugins to offer additional services such as fail-over zones within the same region, ingress routing within the same region, etc. In this article, this functionality is extended by optionally coupling a zone to a mesh network. When a zone is coupled to a mesh network by a mesh topology CRD, the controller will then gather all the required info, such as which nodes are in this zone and node-to-node metrics.

The metrics are not gathered by the controller itself but by a separate agent, instructed with remote procedure calls (RPC). This is done for two reasons. The first one being the current implementation needs an agent running on both the sender and receiver. This could be done by deploying the controller on all required nodes but has the disadvantage of complicating the logic that would now need to be negotiated between multiple of these controllers. The agents are also considerable smaller 14MB versus the 52MB for the controller, because no Kubernetes libraries are needed. The second reason is to implement an extensible solution, one of the goals of Kubernetes itself. By replacing the agents, a different approach to gathering metrics can be used. This can be a more efficient one or even one that supports a different kind of mesh network, such as 5G.

2) ***Mesh agent****:* This component has three functions: *(i)* a TCP ping test, *(ii)* a TCP goodput test, and *(iii)* sending TCP traffic at a certain bitrate. These functions are exposed through remote procedure calls (RPC) and are initiated by the mesh topology controller. The TCP ping test and sending TCP traffic at a certain bitrate are relatively trivial to implement and will not be discussed further. Implementing a simple TCP goodput test was also relatively trivial; however, implementing one that was fast, accurate, and had low overhead on the network was not done successfully.

The one used in this article works as follows: a TCP goodput test is started from the source node and tries to send a configurable amount of data to the destination node. This test was optimized to give fast results while still estimating the true available goodput with good accuracy. A maximum duration of two seconds was also put on this test. In case this duration is exceeded, the test is stopped, and only the received data is taken into account. To limit the negative influence this test has on other network traffic, the test traffic with marked with a type of service (TOS) of background (lowest priority), and priority queuing was enabled on the WiFi modules. Metrics for this solution can be seen in Figure 2 and Figure 3. Both a 1-hop and multi-hop environments are shown. These metrics were taken at different times because they influence each other but are still shown on top of each to show they clearly reflect the same underlying trends.

3) ***Mesh connection controller****:* This controller, together with the mesh connection CRD, is responsible for creating, updating, and deleting network routes. To model network requirements for an application pods are selected with a selector such as a label. When modeling connections, some restrictions were set to make the problem easier to tackle. Firstly data always get pushed from a source to a destination exposed

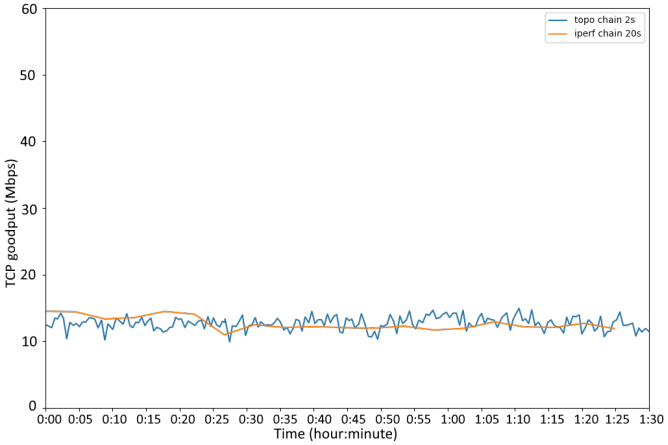

Fig. 2. multi-hop environment 2 hops final iteration (duration 2s) compared to iperf (duration 20s)

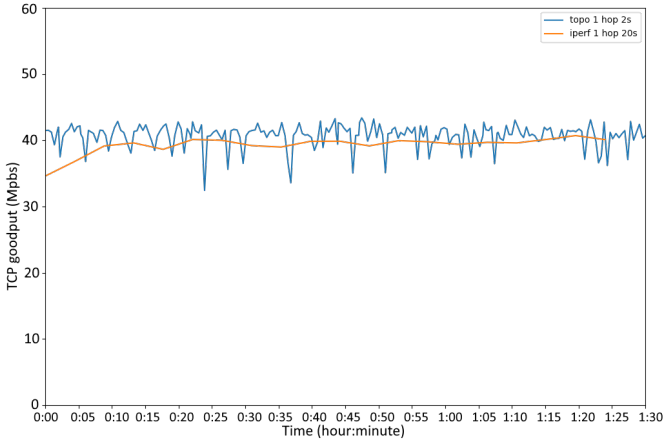

Fig. 3. 1-hop environment final iteration (duration 2s) compared to iperf (duration 20s)

through a service. Secondly, all connections are either push or pull. The second restriction seems unnecessary but is still specified if pull connections get supported in the future. This restriction will be explained in more detail in the subsection of the Kubernetes scheduler plugin.

A distinction was made between a constant bitrate and one needed per incoming connection to model the amount of TCP goodput needed. This currently has no added benefit but could be used to merge multiple incoming connections to a single outgoing connection while still considering that a bigger bitrate would now be required. To be even more flexible, this could also be replaced by an arbitrary function with parameters to support even more use cases and distinctions between different types of incoming connections. Choosing a destination for a waiting connection is not done by the controller but by the Kubernetes scheduler plugin when a new pod gets scheduled, and it is explained in more detail below.

*4) **Mesh proxy**:* The mesh proxy component is responsible for setting up and enforcing all mesh connection routes defined in mesh connection CRDs. It works similarly to the kube-proxy when in iptables mode (the default mode). It looks at all mesh connections, and for each one, a routing chain gets created and appended to the mesh routing chain. This chain is referenced in the prerouting chain. In each mesh chain, rules are created to mirror the behavior defined in the corresponding mesh connection CRD for all waiting and connected connections. If a connection is in the waiting state and thus has no destination, pod traffic gets routing to a black hole IP to prevent the default service routing. Ideally, this traffic should be dropped or refused, but to keep all the logic in the prerouting chain, this approach was used.

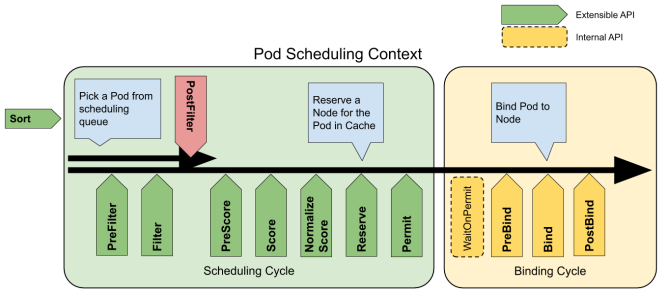

Fig. 4. Kubernetes scheduling framework [20]

*5) **Kubernetes scheduler plugin**:* A plugin was made for the Kubernetes scheduler to support scheduling pods in a mesh network. The Kubernetes scheduler offers extension points for plugins to register. This can be seen in Figure 4. The extension points that are used are sort, prefilter, filter, reserve, and postbind.

The sort extension sorts pods, with the goal to first schedule pods needed in waiting connections. Then handle pods that are not part of a mesh connection or are not needed in a connection currently. In the prefilter extension, all global information (not depending on the node) for this pod gets fetched and checked. If a condition is not met the mesh plugin will be ignored in the following extensions points. In the filter extension, every possible node gets checked separately if it is able to schedule the pod using the information calculated in the prefilter extension. If it does not meet all requirements, the node is dropped from the list of possible nodes and is not used further in the scheduling cycle. In the reserve extension, the node the pod gets scheduled on is known, and all necessary logic to prepare the cluster is done here. First, the waiting connection to changed to a connected connection and filled in with the necessary information. At the same time, fake traffic is enabled until the pod is either ready or canceled. Lastly, the mesh topology metrics are set to invalid, so they get updated, and other iterations of the scheduler know to wait until they are valid again. Finally, the postbind extension is used when the pod is deployed and ready to accept connections. Here only the fake traffic enabled before is disabled again since this will now be replaced by real traffic from the pod.

TABLE II
OVERVIEW OF COMPONENT OVERHEAD DURING TESTING WITH 30 MESH PODS AND FOUR NODES

Component	Size	CPU	Memory	Replica placement
Mesh topology controller	52,6MB	5m	9MB	1 replica no requirements
Mesh agent	13,7MB	30m	10MB	1 replica **each mesh node**
Mesh connection controller	51.9MB	2m	15MB	1 replica no requirements
Mesh proxy	53.7MB	400m	20MB	1 replica **each node**

V. EVALUATION

This section will discuss the overhead introduced delay and scalability of the proposed setup. An overview of the overhead of each component can be found in Table II. This subsection also explains some future work required on the proposed components to have better performance and be more scalable.

A. Overhead

The main overhead added to the cluster is situated in the mesh topology controller, mesh connection controller, and mesh proxy, as seen in Table II. This is because these three components contain most of the logic, and all use the Kubernetes controller libraries. These libraries result in a bigger container size of around 50MB. This could be made smaller by combining the mesh topology and mesh connection controller with a controller-manager. Secondly,a n approach similar to the mesh topology controller and mesh agent workflow could also be used to reduce the mesh proxy's overhead. A central mesh proxy controller would instruct mesh proxy agents to make the required iptables changes. These agents would be minimal and contain almost no logic, resulting in a small container with lower CPU usage. This is especially important since this component has to run on every node.

B. Introduced delay

The delay of the proposed solution will manifest itself when pods are getting scheduled. This is because the plugin will re-queue scheduled pods part of a mesh connection when the required mesh topology metrics are invalid. The delay in scheduling a pod will almost entirely be made up of the time it takes to wait for valid metrics. The main extra delay (D) to schedule one pod part of a mesh connection (mesh pod) is then equal to the time it takes to get a valid metric (T). If a mesh connection destination pod gets deployed to the same node as the source pod, the topology metrics will stay valid, and the time it takes to get a valid metric is zero. The chance of the topology metric still being valid will be denoted as P. To extra delay to schedule a pod at position Q in the queue can then be calculated as:

$$D = Q * (1 - P) * T \tag{1}$$

The amount of pods in the queue is application-dependent and is not dependent on the proposed solution; therefore, this parameter will not be further discussed when looking at the introduced delay. An easy way to decrease the introduced delay would be to give preference to same-node scheduling to prevent waiting for valid metrics. This approach would completely go against the native Kubernetes behavior of balancing the cluster and, as such, was not implemented. The real solution is to minimize the time it takes to get valid metrics. This duration is the main bottleneck and unsolved problem of this research. With the current implementation of getting one metric in a maximum of two seconds. The time it takes to get a valid metric after a mesh pod got scheduled to a different node within a mesh topology consisting of N mesh nodes has an upper bound given by:

$$T = N * (N - 1) * 2s \tag{2}$$

When the environment is not highly dynamic, and the time it takes to schedule pods is unimportant, the proposed solution will work to schedule all pods while keeping goodput requirements. A demo application shown in Figure 5 will be deployed to a Kubernetes cluster to show how the proposed solution works. The camera and microphone components will be deployed as a DaemonSet so that each node will run one replica. The other components are deployed as Deployments with enough replicas to satisfy one mesh connection. The scheduler will decide the node each pod is running on. For this application, the time it took to schedule each component can be seen in Table IV. The number of mesh connections where the source is not on the same node as the destination is also listed. All components in this application send dummy data to each other and log the bitrate at which this data was received. When looking at Table IV, it is possible to experimentally confirm the given equations 1 and 2. The amount of pods in the queue is 32. The part $Q * (1 - P)$ is equal to the number of times new metrics are needed. This value equals the number of mesh connections not on the same node (listed in the table), possibly minus. According to 2 an upper bound for the duration it takes to get valid metrics is 24 seconds (N=4). An approximate upper bound for the case of 17 mesh connections not on the same node, is then $17 * 24 = 408$ when the last connection is on the same node or $16 * 24 = 384$ when the last connection is not on the same node. These values are close to the ones that were experimentally observed, and the difference between them can mostly be assigned to the fact that the scheduling duration is an upper bound and not an absolute value. The bitrates at which data was received are listed in Table III and show that the proposed solution manages to offer the required goodput 97 percent of the time, compared to 42 percent for native Kubernetes. The experiment was repeated five times and was run for 30 minutes each time. The results for native Kubernetes were expected to be around the obtained values since a Kubernetes service routes requests round-robin. This means that all connections compete for network access and will negatively influence each other.

TABLE III
COMPARISON OF NATIVE KUBERNETES COMPARED TO PROPOSED SOLUTION FOR AVERAGE ACHIEVED BITRATE WITH A 95% CONFIDENCE INTERVAL (TEST DURATION OF 30 MINUTES REPEATED FIVE TIMES)

Connection	Required bitrate (Kbps)	Native Kubernetes		Proposed solution	
		Avg achieved bitrate (Kbps)	% of required	Avg achieved bitrate (Kbps)	% of required
Camera - Preprocessor	40 000	15 262 ± 896	38.15 ± 2.24	39 698 ± 46	99.24 ± 0.11
Preprocessor - Model A	20 000	8 271 ± 488	41.35 ± 2.44	19 296 ± 1 022	96.48 ± 5.11
Preprocessor - Model B	20 000	7 999 ± 437	39.99 ± 2.18	18 874 ± 1 175	94.37 ± 5.87
Model A - Storage	5 000	2 867 ± 204	57.35 ± 4.08	4 604 ± 288	92.09 ± 5.76
Model B - Storage	4 000	2 506 ± 168	62.65 ± 4.21	3 911 ± 55	97.77 ± 1.38
Microphone - Storage	1 000	827 ± 41	82.72 ± 4.19	947 ± 37	94.74 ± 3.72
Total	**90 000**	**37 733** ± 1 655	**41.92** ± 1.84	**87 332** ± 1 324	**97.03** ± 1.47

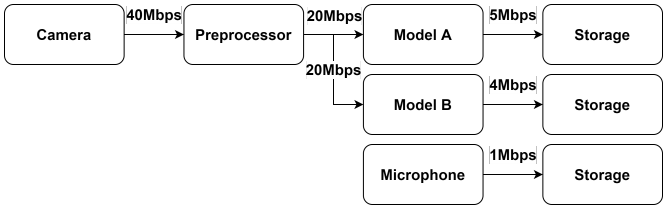

Fig. 5. Demo application

TABLE IV
SCHEDULING DURATION COMPARISON OF DEPLOYING DEMO APPLICATION TO NATIVE KUBERNETES COMPARED TO THE PROPOSED SOLUTION

	Native Kubernetes	Proposed solution	Mesh connections not on same node
Iteration 1	10s	349s	17
Iteration 2	7s	310s	16
Iteration 3	5s	302s	17
Iteration 4	6s	316s	17
Iteration 5	7s	322s	16
Average	**7s** ± 1.64s	**319s** ± 15s	16.6 ± 0.48

C. Scalability

The main change needed to support scalability is a different solution to get available throughput, just like in the previous subsection. This current implementation needs a cycle time that scales $O(N^2)$ and will only be usable in larger mesh networks when the added scheduling time is unimportant. In most cases, however, when there are a lot of nodes present in a mesh network, there will also be many pods scheduled within this mesh network. Combining a large number of pods together with a significant scheduling duration would make the proposed solution work very slowly and poorly in most large mesh networks. One way to combat this difficulty already present in the current solution is to support having multiple separate mesh networks. Instead of one large network, the network could be divided into multiple small networks. Gateway nodes would have to be statically or dynamically defined to take traffic going in and out of a mesh network into account to support these mesh networks. This gateway functionality was not implemented. Once the traffic is outside of a mesh network, a more traditional network routing approach (one not taking mesh characteristics into account) could be used, such as the one proposed in [21].

VI. CONCLUSION

It can be seen in the evaluation section that the proposed component framework manages to offer the required bitrates when running Kubernetes on top of a wireless mesh network. To fully support a dynamic mesh network, a vast amount of future work is still required. The majority of this work would be adding the three features listed next. Firstly, support to have one destination pod for multiple connections at the same time. Secondly, connecting waiting connections without the scheduler. This would allow connections to be made using existing pods and could open new possibilities such as optimizing network requirements as a whole, restoring connections by rescheduling all pods, scaling pod replicas if existing pods cannot fill all connections, etc. Lastly, adding periodic rechecking of all the requirements would be needed to support the dynamic nature of a mesh network. Most of this work is pretty straightforward and extends the proposed solution. However, the main bottleneck of the current approach is getting the available goodput of a node-to-node connection, and as such, this component has to be replaced or modified when a faster or lower-overhead option is found. Seeing how cheap and easy mesh networks are to set up, combined with the increasing trend of deploying Kubernetes to the edge, we conclude that future research is not only essential but the key to creating a universal container orchestration platform that can be run in any environment.

REFERENCES

[1] F. Liu and Y. Bai, "An overview of topology control mechanisms in multi-radio multi-channel wireless mesh networks," *EURASIP Journal on Wireless Communications and Networking*, vol. 2012, no. 1, pp. 1–12, 2012.

[2] J. Santos, T. Wauters, B. Volckaert, and F. De Turck, "Towards network-Aware resource provisioning in kubernetes for fog computing applications," in *Proceedings of the 2019 IEEE Conference on Network Softwarization: Unleashing the Power of Network Softwarization, NetSoft 2019*, 2019, pp. 351–359.

[3] "scheduler-plugins/kep/260-network-aware-scheduling at KepDevWithNTController · jpedro1992/scheduler-plugins · GitHub." [Online]. Available: https://github.com/jpedro1992/scheduler-plugins/tree/KepDevWithNTController/kep/260-network-aware-scheduling

[4] L. Wojciechowski, K. Opasiak, J. Latusek, M. Wereski, V. Morales, T. Kim, and M. Hong, "NetMARKS: Network metrics-AwaRe kubernetes scheduler powered by service mesh," in *Proceedings - IEEE INFOCOM*, vol. 2021-May. Institute of Electrical and Electronics Engineers Inc., may 2021.

[5] C. Sarr, C. Chaudet, G. Chelius, and I. G. Lassous, "Bandwidth estimation for IEEE 802.11-based ad hoc networks," *IEEE Transactions on Mobile Computing*, vol. 7, no. 10, pp. 1228–1241, 2008. [Online]. Available: https://hal.inria.fr/inria-00384832

[6] G. Venkatesh and K. C. Wang, "Estimation of maximum achievable end-to-end throughput in IEEE 802.11 based wireless mesh networks," in *Proceedings - Conference on Local Computer Networks, LCN*, 2009, pp. 1110–1117. [Online]. Available: https://tigerprints.clemson.edu/all_theses

[7] S. K. Khangura and M. Fidler, "Available bandwidth estimation from passive TCP measurements using the probe gap model," in *2017 IFIP Networking Conference, IFIP Networking 2017 and Workshops*, vol. 2018-Janua. Institute of Electrical and Electronics Engineers Inc., jul 2017, pp. 1–9.

[8] T. Goto, A. Tagami, T. Hasegawa, and S. Ano, "TCP throughput estimation by lightweight variable packet size probing in CDMA2000 1x EV-DO network," in *Proceedings - 2009 9th Annual International Symposium on Applications and the Internet, SAINT 2009*, 2009, pp. 1–8.

[9] Y. S. Liaw, A. Dadej, and A. Jayasuriya, "Estimating throughput available to a node in wireless ad-hoc network," in *2004 IEEE International Conference on Mobile Ad-Hoc and Sensor Systems*, 2004, pp. 555–557.

[10] Y. Y. Munaye, A. B. Adege, G. B. Tarekegn, Y. R. Li, H. P. Lin, and S. S. Jeng, "Deep learning-based throughput estimation for UAV-Assisted network," in *IEEE Vehicular Technology Conference*, vol. 2019-Septe. Institute of Electrical and Electronics Engineers Inc., sep 2019.

[11] A. Samba, Y. Busnel, A. Blanc, P. Dooze, and G. Simon, "Instantaneous throughput prediction in cellular networks: Which information is needed?" in *Proceedings of the IM 2017 - 2017 IFIP/IEEE International Symposium on Integrated Network and Service Management*. Institute of Electrical and Electronics Engineers Inc., jul 2017, pp. 624–627.

[12] H. Zhao, E. Garcia-Palacios, J. Wei, and Y. Xi, "Accurate available bandwidth estimation in IEEE 802.11-based ad hoc networks," *Computer Communications*, vol. 32, no. 6, pp. 1050–1057, apr 2009.

[13] Y. S. Liaw, A. Dadej, and A. Jayasuriya, "Estimating throughput available to a node in wireless ad-hoc network," in *2004 IEEE International Conference on Mobile Ad-Hoc and Sensor Systems*, 2004, pp. 555–557.

[14] M. Jain and C. Dovrolis, "End-to-end available bandwidth: Measurement methodology, dynamics, and relation with TCP throughput," *IEEE/ACM Transactions on Networking*, vol. 11, no. 4, pp. 537–549, aug 2003.

[15] V. P. Kemerlis, E. C. Stefanis, G. Xylomenos, and G. C. Polyzos, "Throughput unfairness in TCP over WiFi," in *WONS 2006 - 3rd International Conference on Wireless on Demand Network Systems and Services*, 2006, p. 1.

[16] B. Veal, K. Li, and D. Lowenthal, "New methods for passive estimation of TCP round-trip times," in *Lecture Notes in Computer Science*, vol. 3431. Springer, Berlin, Heidelberg, 2005, pp. 121–134. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-540-31966-5_10

[17] B. A. T. M. A. N, "BATMAN - Open Mesh," p. 29. [Online]. Available: https://www.open-mesh.org/projects/open-mesh/wiki/BATMANConcept

[18] Open-Mesh, "Wiki - batman-adv - Open Mesh." [Online]. Available: https://www.open-mesh.org/projects/batman-adv/wiki/Wiki

[19] L. Liu, J. Liu, H. Qian, and J. Zhu, "Performance evaluation of BATMAN-adv wireless mesh network routing algorithms," in *Proceedings - 5th IEEE International Conference on Cyber Security and Cloud Computing and 4th IEEE International Conference on Edge Computing and Scalable Cloud, CSCloud/EdgeCom 2018*. Institute of Electrical and Electronics Engineers Inc., jul 2018, pp. 122–127.

[20] Kubernetes, "Kubernetes Documentation - Kubernetes," 2019. [Online]. Available: https://kubernetes.io/docs/home/

[21] "scheduler-plugins/kep/260-network-aware-scheduling at KepDevWithNTController · jpedro1992/scheduler-plugins · GitHub." [Online]. Available: https://github.com/jpedro1992/scheduler-plugins/tree/KepDevWithNTController/kep/260-network-aware-scheduling