Creating disaggregated network services with eBPF: the Kubernetes network provider use case

Creating Disaggregated Network Services with eBPF: the Kubernetes Network Provider Use Case

Federico Parola, Leonardo Di Giovanna, Giuseppe Ognibene, Fulvio Risso
Dept. of Control and Computer Engineering, Politecnico di Torino
24, Corso Duca degli Abruzzi, Torino, 10129, Italy
Email: {federico.parola, leonardo.digiovanna, giuseppe.ognibene, fulvio.risso}@polito.it

***Abstract*—The eBPF technology enables the creation of custom and highly efficient network services, running in the Linux kernel, tailored to the precise use case under consideration. However, the most prominent examples of such network services in eBPF follow a monolithic approach, in which all required code is created within the same program block. This makes the code hard to maintain, to extend, and difficult to reuse in other use cases. This paper leverages the Polycube framework to demonstrate that a disaggregated approach is feasible also with eBPF, with minimal overhead, introducing a larger degree of code reusability. This paper considers a complex network scenario, such as a complete network provider for Kubernetes, presenting the resulting architecture and a preliminary performance evaluation.**

I. Introduction

The extended Berkeley Packet Filter (eBPF) allows executing arbitrary code in different kernel hooks, which are triggered upon multiple events, such as a syscall or a packet received. This enables to extend the processing done by the kernel without changing its source code or loading kernel modules. eBPF code is sandboxed in order to guarantee that a user provided program can not compromise the functioning of the kernel. Several projects leverage eBPF to bring new functionality to the kernel in the fields of security, monitoring and networking. However, most of the networking-related projects implement new features as monolithic eBPF programs, making the code hard to maintain, to extend, and difficult to reuse in other use cases. In this respect, the Polycube software framework [1] addresses some of the well-known eBPF limitations when creating complex virtual network functions [2], and introduces the capability to split eBPF software in multiple, independent network functions, which can be arbitrarily connected in order to create a more complex service graph. This enables the creation of complex networking services by compositing elementary basic blocks (e.g., bridge, router, firewall, NAT, load balancer, and more), with an high degree of reusability.

Container networking is a perfect example of such scenario, and has gained key importance with the diffusion of the microservices architecture and the decomposition of applications in a collection of small, loosely-coupled services running in containers. Container orchestrators such as Kubernetes (K8s) must provide a flexible and efficient network infrastructure, since containers can be created and destroyed at high frequency, must exchange a lot of data and must be easily accessible from the Internet. However, K8s defines only a functional networking model and it relies on 3rd party plugins for the actual implementation of network services.

This paper shows how a K8s network provider can be created using solely eBPF primitives according to the *disaggregated services* model, i.e., defining a modular architecture based on traditional network components such as routers, load balancers and NATs, and without giving up on performance.

This paper is structured as follows. Section II provides an overview of the K8s networking model and how Polycube achieves service disaggregation. Section III describes the overall node architecture of our solution while Section IV shows a preliminary performance comparison compared to existing solutions. Finally, Section V describes the relevant related work and Section VI draws our conclusions, highlighting the potential future work.

II. Background

A. Kubernetes networking

Kubernetes is an open-source orchestrator for containerized applications and defines a functional network architecture organized in three levels. (*i*) **Pods**, the basic scheduling concept, execute containerized applications that are connected to a default virtual network, whose outreach is limited to a single server (*node*, in the Kubernetes terminology). Pods are ephemeral entities that can be destroyed and re-spawned if needed, even on another node. Each server can host a maximum number of pods, usually organized in a contiguous and private address space (e.g., consecutive /24 networks on different nodes). (*ii*) **Physical nodes**, which inherit their addressing space from the physical datacenter network; for scalability reasons, this usually includes switched and routed portions. (*iii*) **Services**, a higher-level concept that enables the reachability of one or more (homogeneous) pods by means of the same network identifier (e.g., IP address). This primitive guarantees the decoupling between the *service* IP endpoint, which remains stable, from the *actual* pod(s) that provides the service, whose IP can change (e.g., in case the pod is restarted in another location) or it may be present in multiple replicas (hence, multiple IPs could be used). Services leverage a third addressing space, disjoint from pod and datacenter addresses.

Kubernetes foresees three types of services. A **ClusterIP** identifies a service that is reachable only from pods within the cluster, or by an application that runs on the cluster

nodes. A **NodePort** service, instead, is reachable from outside the datacenter: a TCP/UDP port `<nport>` is allocated to the service itself, and all packets directed to any `<node_ip_address:nport>` will be redirected to one of the pods associated to the given service. NodePorts are not widely used because they require (*i*) nodes with reachable (e.g., public) IP addresses; (*ii*) external users to know the IP addresses of the nodes and (even worse) (*iii*) the port that has been allocated. Finally, a **LoadBalancer** service allocates a public IP address associated to the given service, which is achieved by interacting with an external entity in charge of the above public addresses; all packets directed to the LoadBalancer IP address will be delivered to one of the corresponding pods.

Basic connectivity between pods is provided through the cooperation of datacenter networking and plugins implementing the CNI (Container Network Interface) [3], a specification that enables changeable modules that configure network interfaces in Linux containers. The CNI specification is very simple, dealing only with network connectivity of containers and removing allocated resources when the container is deleted. Instead, packets to services are by default handled by a dedicated Kubernetes component, `kube-proxy`, which configures `iptables` with the proper rules to translate *service* IP addresses into *pod* IP addresses, and to implement load-balancing policies.

Most of the so-called CNI providers (e.g., Cilium, Calico, Flannel, etc) implement more than just the base CNI specification and include (*i*) data-center wide networking (either using an *overlay model*, e.g., through `vxlan` interfaces, or *direct routing*, hence interacting with the datacenter physical infrastructure to push the proper routes that satisfy K8s reachability rules) and (*ii*) IP address management (IPAM module). Instead, most of the CNI providers rely on `kube-proxy` for services: a packet coming from a pod and directed to a service is delivered by the CNI to kube-proxy, which operates the proper transformation on IP addresses and ports and returns it again to the network plug-in, which takes care of delivering it to the target pod, possibly traversing the datacenter network.

K8s adopts a functional model for networking, defining a set of behavioral rules for network connectivity[1] but without specifying how those must be implemented by the specific network provider. In addition to rules already mentioned for services, it adds the following ones for basic connectivity: (*i*) all pods can communicate with all other pods without NAT; (*ii*) agents on a node can communicate with all pods on that node; (*iii*) pods in the host network of a node can communicate with all pods on all nodes without NAT. It turns out that network providers have full freedom to choose their own implementation strategy, hence privileging e.g., the easiness of use, performance, scalability, and more.

However, this freedom is widely recognized as a nightmare, being very difficult to understand how each network provider works under the hood, hence severely impairing the capability of a network engineer to debug a problem. This is exacerbated by the complexity of the Kubernetes networking, which overall includes functions such as bridging and routing (for pod-to-pod connectivity), load balancing and NAT (for pod-to-service and Internet-to-service), and masquerading NAT (for pod-to-Internet), not to mention the necessity of security policies (hence, firewalls) to protect both pod-to-everything and external-to-everything communications.

Finally, all the above networking components must be integrated with a control plane, which interact with K8s and detect any change in the status of the cluster (e.g., a node/pod/service is added/removed, a service is scaled up/down, etc.), hence propagating the required configurations in all involved components (e.g., adding a new node requires configuring a new route toward that node in the routing table of all existing nodes).

Given this complexity, it becomes evident that the capability to rely on well-known (disaggregated) functions (e.g., bridging, routing, load balancers), each one running with their configurations and state, would greatly simplify day-by-day monitoring and debugging operations compared to network providers created according to the monolithic model.

B. Service disaggregation with Polycube

Polycube [1] is an open-source software framework based on eBPF that enables the creation of arbitrary network function chains, adopting the same model (boxes connected through wires) currently used in the physical world. Polycube network functions, called *cubes*, are composed by an eBPF-based data plane running in kernel (actually one or more eBPF programs) and a control/management plane running in user space. A user space daemon (*polycubed*) provides a centralized point of control, allowing to access the configuration and state of cubes through a RESTful API. Cubes can be seamlessly connected to each other or to network interfaces through *virtual ports*, an abstraction that is implemented through an eBPF wrapping program that performs some pre- and post- processing in order to receive/send packets from the previous/to the next component in the chain. To implement this redirection mechanism, each port is identified by a unique ID inside the cube, and each cube maintains a forward chain map containing information about the peer associated to each port. This information is used by the post-processor to apply the correct action to forward the packet, that could be either a *tail call* to the eBPF data plane of the next cube or a `bpf_redirect()` to a destination interface. Figure 1 shows an example of this mechanism for a simple topology composed of one router and one bridge. Chaining capabilities of Polycube represent a suitable way to create disaggregated services; however, this solution has never been validated in a complex scenario such as K8s networking.

III. Architecture

Our proposed architecture targets the entire K8s networking, including also services and, potentially, network policies, hence replacing also `kube-proxy`, i.e., the component that provides cluster-wide service-to-pod translation and load balancing. Our network provider supports ClusterIP and NodePort

[1] https://kubernetes.io/docs/concepts/services-networking/.

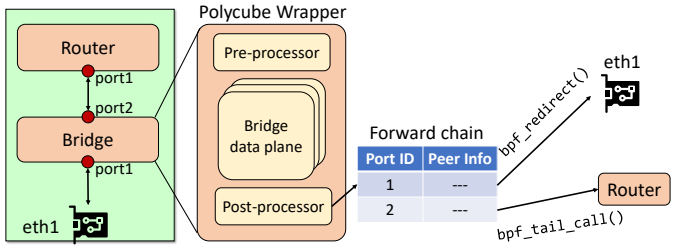

Fig. 1: Chaining and ports in Polycube.

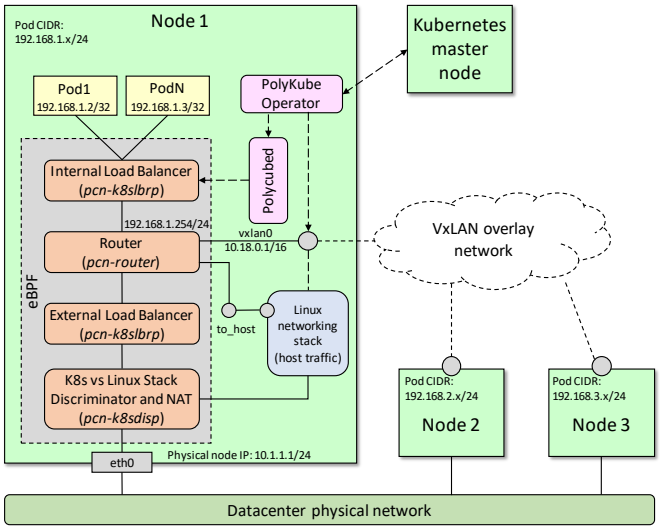

Fig. 2: Overall architecture of the eBPF K8s network provider.

services and relies on a VxLAN overlay network to support inter-node communication. The current version of the prototype does not support security policies but, thanks to the modular approach, this and other functionalities can easily be added without any change in the other components. The architecture (shown in Fig. 2) leverages four Polycube-based independent network services, while it relies on the kernel to handle the lifecycle of VxLAN tunnels.

A. Main components

K8s vs Linux Stack Discriminator and NAT (DISC-NAT). This service performs source NATting for the pod-to-Internet traffic, replacing the address of the pod with the address of the node. For incoming packets, it distinguishes among traffic directed to the host (either directly or because it needs VxLAN processing) and traffic directed to pods (an external host trying to contact a NodePort service or the return traffic of a pod). This is done by checking that the packet (*i*) does not belong to a NATted session (i.e., a lookup in the NAT session table of the service), and (*ii*) that is not directed to a NodePort service (i.e., a check against the TCP/UDP destination port of the packet). In case both lookups fail, the packet is sent to the Linux network stack. Vice versa, in the first case we apply the reverse NAT rule and the packet continues its journey towards the pods. In the second case, different actions can be applied according to the *ExternalTrafficPolicy* of the rule. If the policy is *local*, the traffic is allowed to reach only backend pods located on the current node, hence the packet can proceed towards the pod without modifications. In case the policy is *cluster*, the packet can also reach backend pods located on other nodes. Since later in the chain the packet will be processed by a load balancer and we must guarantee that the return packet will transit through the same load balancer, we apply source NAT replacing the source IP address with the address of a fictitious pod belonging to the PodCIDR of the current node (currently the first address of the range is used).
External Load Balancer (ELB). This element maps new sessions coming from the Internet and directed to a NodePort service to a corresponding backend based on the 5-tuple of the first packet. This load balancing decision is stored in a session table, implemented as a Least Recently Used (LRU) eBPF map, and reused for all subsequent packets. Old sessions are automatically purged by the LRU map. Incoming packets are updated with destination/port of the backend, while source fields are restored to service values for outgoing traffic.

Router. Since K8s requires that (*i*) all the pods in the cluster can communicate with all pods without NAT and (*ii*) different network addresses (PodCIDR) are allocated to pods on each node, a routing component is required. To facilitate the operations of the Internal Load Balancer (see later), we do not use a bridge between pods, hence forcing all pod traffic to be always delivered to the router. In fact, our network plugin assigns a /32 network to each pod and adds an ARP static entry for the gateway; hence all the packets are sent directly to the gateway, with pods never issuing any ARP request. The router is configured with four ports: (1) towards the physical interface of the node; (2) towards local pods, configured with the proper PodCIDR (e.g., /24); (3) to enable the reachability of K8s processes and pods running in the host network (e.g., kubelet); (4) connected to a kernel VxLAN interface, which is used for inter-node pod-to-pod communication.
Internal Load Balancer (ILB). This load balancer operates on traffic coming from local pods and directed to ClusterIP services; all the other traffic is forwarded as is. This module has two types of ports; 'edge' ports are connected to entities that generate new sessions (hence, pods), while the 'server' port is the one used to reach the final servers, hence is connected to the router. Edge ports are configured with the IP address of the pod in order to be able to forward packets coming from the router to the correct destination. The load balancing logic is the same of the ELB, with a service-specific *InternalTrafficPolicy* attribute determining which backends (*local* or *cluster*-wide) are configured in the load balancer.
K8s Control Logic. A K8s operator is in charge of reacting to the cluster events and reconfigure the required network parameters in the controlled cluster. The operator has to react to events related to the following three Kubernetes resources. (1) *Nodes*: when a node joins/leaves the cluster, a route is added/removed in the *Router* to update the reachability of the given PodCIDR through the VxLAN overlay network, and the node address is added to the VxLAN configuration.

(2) *Services*: the operator watches events regarding ClusterIP and NodePort services. ClusterIP services trigger an update of the ILB, while a NodePort triggers an update of the ELB and DISC-NAT module for the obvious reasons, as well as the ILB because of the creation of the ClusterIP address that is associated with the NodePort service.
(3) *Endpoints*: the operator watches any event that refers to Endpoints associated to Services. For each pair (address, port) extracted from the endpoints object, the corresponding service backend is updated on the proper Load Balancer: the ILB for ClusterIP services and both for NodePort services.

This component is deployed as a K8s DaemonSet, which ensures it runs on any node; K8s adopts a distributed configuration, in which each node has its own agent in charge of the node network configuration. This DaemonSet runs as privileged pod, and it includes the `polykube-operator` container (running the actual K8s control plane) and the `polycubed` container (running the Polycube daemon). The two communicate through the node loopback interface.

B. Communication scenarios

The main communication scenarios of a typical K8s cluster, as shown in Fig. 3, are implemented as follows.

Pod-to-pod. The originating pod sends its packets to the ILB, which transparently forwards them to the Router. If the destination is on the same node, the traffic is sent back immediately and the ILB forwards it to the destination. If the destination pod is on another node, the router redirects the packets to the VxLAN interface, where they are encapsulated by the kernel and forwarded to the destination node. On the remote node, the DISC-NAT passes the traffic to the kernel, which decapsulates it and sends it to the router and then to destination pod.

Pod-to-Internet. The traffic traverses the ILB, then the router forwards it towards the physical interface of the node, hence transparently crossing also the ELB. The NAT, instead, applies a source NATting rule, hence replacing the address of the pod with the one of the node, as well as the source port with a new available one. This allows the return traffic to reach the correct node without the necessity to advertise the PodCIDR on the external network. On the return path, the NAT checks if the packet belongs to a translated session; if so, it replaces the destination address and port of incoming packets with the ones of the original pod.

Pod-to-service. The pod traffic toward a ClusterIP service is first processed by the ILB, which selects a proper backend pod and updates the destination address and port of packets. Traffic is then handled by the router in the same way as with the pod-to-pod communication. For return traffic, the ILB checks if the packet belongs to a translated session; if so, it restores the original source service address and port.

Internet-to-service. When a remote host wants to contact a NodePort service, the packet is first processed by the DISC-NAT, that identifies it as targeting a NodePort service (based on the destination port). If the *ExternalTrafficPolicy* of the service is *cluster*, the DISC-NAT updates the source address of the packet with the one of the fictitious pod, to guarantee that the return packet will come to the same node. The packet is then forwarded to the ELB, which (*i*) selects a proper backend pod, (*ii*) updates the destination address and port with the ones of the backend, and (*iii*) forwards the packet to the router, that can handle it such as in normal pod-to-pod communications.

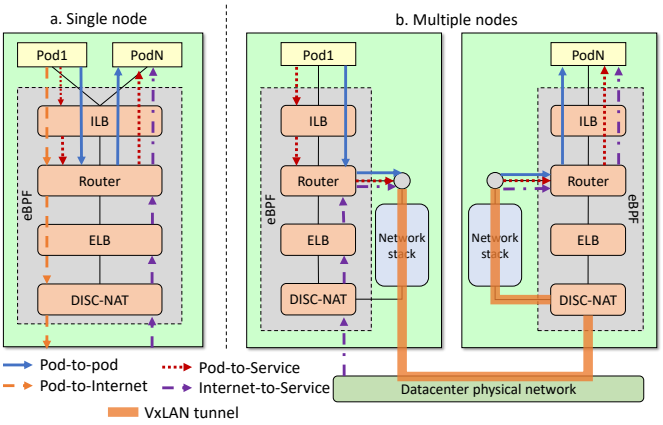

Fig. 3: Modules involved in single (a) and multi-node (b) communications.

IV. Evaluation

This section presents an assessment of the performance provided by our network provider under different circumstances, to determine whether the disaggregated approach would introduce any noticeable performance penalty. We run the tests using the `iperf3` tool, configured with the default parameters; the server was always running in a pod, while the client was either in a physical machine or in another pod depending on the test. Tests were carried out on a cluster of 2 nodes, each one featuring a CPU Intel® Xeon® CPU E3-1245v5@3.50GHz (4 cores plus HyperThreading), 64GB RAM, and a dualport 40 GbE Ethernet XL710 QSFP+ card, all running Linux kernel v5.4.0 and K8s v1.23.5. Tests involve other two eBPF-based solutions (namely, Cilium and Calico) and a widely used 'traditional' approach such as **Flannel** [4]. All providers were deployed using the VxLAN overlay model; Cilium and Calico were configured with their eBPF kube-proxy replacement, hence enabling a complete eBPF data plane such as in our solution.

We considered the following communication scenarios: **pod-to-pod**: a pod client connects to the actual IP address of a pod server, showing the performance of the base networking without load balancing; **pod-to-service**: a pod client connects to a pod server using its ClusterIP service, to evaluate the performance of the load balancer as well as the L3 routing; **internet-to-service**: the client is executed in an external host and the pod server is accessed through its NodePort service. Tests were performed with pods running both on a single node and on multiple nodes, with the latter adding the overhead of the VxLAN encapsulation and the limitation of the physical network (link speed, PCI bus) and requiring to cross the physical network twice ((*i*) client to receiving node; (*ii*) receiving

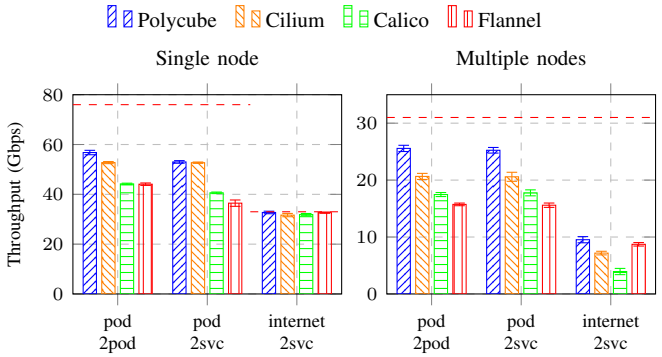

Fig. 4: Throughput comparison (TCP).

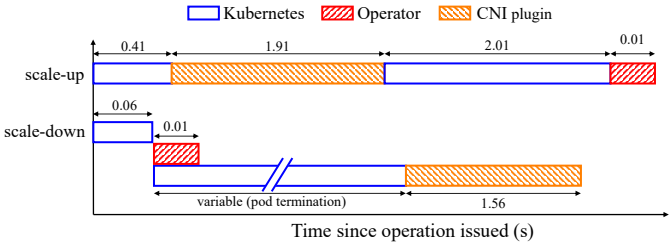

Fig. 5: Reaction time in case of scale up/down events.

node to backend node) for the *internet-to-service* tests. Results are depicted in Fig. 4, with the red dashed lines representing the baseline achieved running bare metal `iperf3` on localhost (in case of single node) or between two nodes. As expected, the throughput decreases when the traffic traverses a larger number of network components. However, despite the disaggregated architecture, our solution provides always better performance compared to other solutions, with even higher margins when considering multiple nodes. While this result may be impacted by other providers supporting more features than our PoC code, such as network policies, these have not been used in the tests, hence providing the ground for a fair comparison. Overall, this suggests how disaggregation does not introduce performance penalties compared to a traditional monolithic approach.

Figure 5 shows the reaction time of our operator and CNI plugin when requiring to scale up/down the pods of a service, compared with time required by other Kubernetes components until connectivity to the target pod is available/disabled. Results show that the time taken by our components is negligible compared to the overall time required by K8s to react.

V. Related work

Among the many eBPF-based network services, we cite here only the ones that are most representative in this space.

Katran [5] represents a software solution to offer scalable network load balancing to layer 4 that leverages eBPF/XDP to provide fast packet processing. While being very sophisticated, it has been engineered to be the sole (monolithic) network function active on the network path, hence preventing the deployment of other functions operating on the same traffic.

Cilium [6] provides networking, security and observability for cloud-native environments such as Kubernetes clusters. Cilium is based on eBPF, which allows for the dynamic insertion of powerful network security, visibility and control logic into the Linux kernel. In Cilium, eBPF is used to provide high-performance networking, multi-cluster and multi-cloud capabilities, advanced load balancing, transparent encryption, extended network security features, and much more. While providing observability primitives through its *Hubble* module, its internals are rather complex and made with a monolithic approach. Cilium defines a set of six logical objects (Prefilter, Endpoint Policy, Service, etc.), based on six different features offered by the provider (DoS mitigation, network policies, load balancing, etc.). However, these objects are not mapped into clearly separated modules, neither for the data plane (their logic is scattered among different intertwined eBPF programs), nor from a topological point of view (cannot track the path of a packet or capture the traffic flowing from one object to another), nor from a control plane perspective (cannot configure and inspect these objects independently). Similar characteristics can be found in **Calico** [7], which recently adopted the eBPF/XDP technology as well.

VI. Conclusions

We presented a network provider for Kubernetes based on disaggregated eBPF services, which improves monitoring and debugging as well as how code can be maintained, extended and reused. Our open-source solution[2] demonstrates the feasibility of the disaggregated approach in eBPF and our preliminary evaluation shows no particular overhead introduced by our model with respect to another state-of-the art monolithic solution. As a future work we plan to introduce support for (*i*) direct routing and (*ii*) network policies.

Acknowledgment

Authors thank Hamza Rhaouati for his initial work on this topic, all the people who contributed to the Polycube project, and Roberto Procopio and Yunsong Lu for their support. Finally, Federico Parola acknowledges the support from TIM S.p.A. through the PhD scholarship.

References

[1] S. Miano, F. Risso, M. V. Bernal, M. Bertrone, and Y. Lu, "A framework for ebpf-based network functions in an era of microservices," *IEEE Transactions on Network and Service Management*, vol. 18, no. 1, pp. 133–151, 2021.

[2] S. Miano, M. Bertrone, F. Risso, M. Vásquez Bernal, and M. Tumolo, "Creating complex network service with ebpf: Experience and lessons learned," in *High Performance Switching and Routing (HPSR)*. IEEE, 2018.

[3] "Cni – the container network interface," https://github.com/containernetworking/cni, (Accessed on: Jan. 30, 2022).

[4] "Flannel," https://github.com/flannel-io/flannel, (Accessed on: Apr. 12, 2022).

[5] "Katran," https://github.com/facebookincubator/katran, (Accessed on: Feb. 8, 2022).

[6] "Cilium," https://github.com/cilium/cilium, (Accessed on: Feb. 8, 2022).

[7] "Calico," https://github.com/projectcalico/calico, (Accessed on: Feb. 8, 2022).

[2]Code and docs available at https://github.com/polycube-network/polykube.