ALFA BK UNIVERSITY
FACULTY OF INFORMATION TECHNOLOGIES
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
International Scientific Conference

UDK:621.39:004.7
DOI: 10.46793/ALFATECHproc25.165S
Original scientific paper

Implementing Network Policies in Kubernetes

Sava STANISIC[1], Vladimir MLADENOVIC[2]

Abstract: Managing network traffic and ensuring security has become critical for maintaining robust Kubernetes environments. This paper aims to investigate the implementation of Kubernetes Network Policies to enhance network security and operational efficacy. The objectives of these research include exploring the configuration of NetworkPolicy resources, employing label selectors, and managing namespaces. The methodology presented in this paper involves leveraging advanced tools like Flannel and Calico to enforce network policies effectively. The results indicate significant improvements in traffic control and workload security, offering a comprehensive guide for optimizing Kubernetes clusters with refined network policy management techniques.

Keywords: Egress Traffic; Ingress Traffic; Kubernetes; Network Policies; Security

1 INTRODUCTION

Managing network traffic and ensuring security has become critical for maintaining robust Kubernetes environments in the era of cloud-native applications. Kubernetes, a powerful container orchestration platform, provides a flexible and scalable way to manage applications across diverse environments. However, with this flexibility comes the challenge of securing network communication within the clusters, making network policies indispensable [1].

Network policies in Kubernetes play a pivotal role in controlling the flow of traffic between different pods and ensuring that only authorized communication is allowed. By defining these policies, administrators can enhance the cluster's security and manage traffic more efficiently.

Network policies in Kubernetes are akin to firewall rules for cloud environments. They enable administrators to define rules and constraints on how pods communicate with each other within a cluster, and with external networks.

This paper aims to delve into the configuration of NetworkPolicy resources, employing label selectors, and managing namespaces to achieve a finely-tuned network policy framework.

2 KUBERNETES

Kubernetes, often referred to as "K8s", is an open-source platform used for automating the deployment, scaling, and management of containerized applications. Developed originally by Google and now maintained by the Cloud Native Computing Foundation (CNCF), Kubernetes has revolutionized the way applications are managed and deployed in modern IT environments.

In this section, the basic Kubernetes components will be presented, alongsid some key features of this container orchestration tool.

At the heart of Kubernetes lie two main components: Master Node and the Worker Nodes [2].

The Master Node serves as the control plane, orchestrating the entire cluster's operation. The Master Node consists of an API Server that processes user and application requests, a Scheduler that distributes workloads based on resource availability, and a Controller Manager that ensures the cluster maintains its desired state. Also, the Master Node has a data store called etcd, that basically holds the cluster configuration and state information.

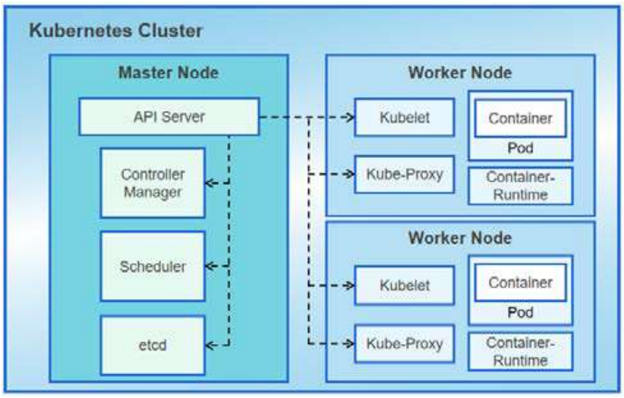

Figure 1 Basic Kubernetes Architecture

The Worker Nodes are the nodes that run containerized applications. Each Worker Node includes a kubelet, which guarantees that container are running within a pod, and a Kube-Proxy, which manages networking both within the cluster and with external networks. The Container Runtime Interface (CRI), such as Docker or containerd, manages the container lifecycle, ensuring applications run smoothly.

The basic architectural components of Kubernetes are presented on the Figure 1.

A fundamental concept in Kubernetes is the Pod, the smallest deployable unit in the Kubernetes ecosystem. A pod represents a single instance of a running process and can consist of one or more containers that share the same network namespace. This design simplifies the management of containers that are closely coupled and need to share resources [3].

Services in Kubernetes create a stable interface to a set of Pods. They abstract complex networking details, ensuring consistent access to Pods regardless of their lifecycle changes or the physical nodes they run on. Deployments lay the foundation for managing application updates declaratively. By defining the desired state and the number of replicas for a set of Pods, Deployments ensure applications are always running as specified.

The architecture of a simple Kubernets cluster is presented on the Figure 2.

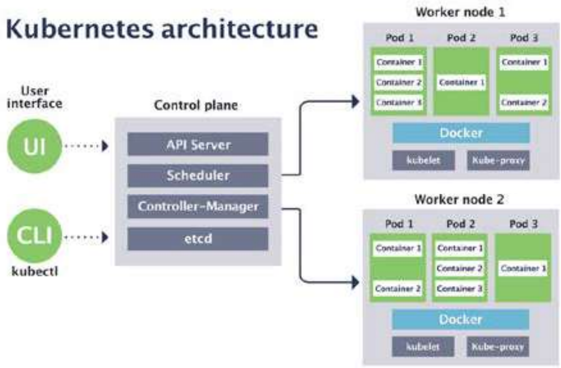

Figure 2 Basic Architecture of a simple Kubernetes Cluster

One of the standout features of Kubernetes is its scalability. The platform can seamlessly scale applications across clusters of machines, adapting to changing workloads. Additionally, Kubernetes ensures high availability with self-healing and load-balancing capabilities, which maintain application uptime and reliability [4]. Automated rollouts and rollbacks allow for smooth application updates and reverting to previous versions if issues arise. Kubernetes also excels in resource management, efficiently balancing loads across nodes to optimize the use of computing resources.

3 NETWORKING IN KUBERNETES

Kubernetes networking is crucial for the functionality and security of applications running within the cluster. Understanding how Kubernetes handles networking is essential for effectively deploying and managing containerized applications.

Kubernetes abstracts the complexities of networking, enabling developers to focus on application logic rather than the underlying infrastructure.

It also employs a flat networking model where each Pod is assigned a unique IP address, ensuring that Pods can communicate with each other directly without network address translation (NAT). This networking model allows applications to behave as if they were running on a single machine.

Three primary concepts form the foundation of Kubernetes networking:

1. **Pod-to-Pod Communication**: All Pods can freely communicate with each other by default, which facilitates seamless inter-pod communication.
2. **Pod-to-Service Communication**: Services in Kubernetes provide a stable IP address and DNS name for a set of Pods, ensuring consistent and reliable access.
3. **External-to-Service Communication**: Services can be exposed to external traffic, allowing users and external systems to interact with the applications running in the cluster [5].

This tool relies on the Container Network Interface (CNI) plugin architecture to manage network resources for containers. CNI provides a standardized mechanism for configuring and managing network interfaces in Linux containers. When a Pod is created, the CNI plugin is responsible for allocating an IP address and setting up the necessary routing, thus ensuring each Pod can communicate within the cluster.

Services abstract the underlying network details and provide a stable IP address and a DNS name for accessing a set of Pods. This abstraction ensures that applications can reliably communicate with each other using consistent addresses. Endpoints are objects that track the IP addresses of the Pods associated with a Service, allowing the network proxy to route traffic to appropriate Pods.

There are several types of Services:

1. **ClusterIP**: The default type, which exposes the Service on a cluster-internal IP. This type is accessible only within the cluster.
2. **NodePort**: Exposes the Service on each Node's IP at a static port.
3. **LoadBalancer**: Exposes the Service externally using a cloud provider's load balancer.
4. **ExternalName**: Maps a Service to the contents of the externalName field (e.g., myservice.example.com) by returning a CNAME record with its value.

Ingress controllers manage external access to services within a cluster, typically via HTTP and HTTPS. They provide features such as load balancing, SSL termination, and name-based virtual hosting. The most used Ingress controllers, at the moment of writing, are:

1. **NGINX Ingress Controller**: Widely used for its reliability and extensive feature set.
2. **Traefik**: Known for its simplicity and integration with multiple backend technologies.
3. **HAProxy**: Offers high performance and advanced routing capabilities.

Kube-Proxy is a network component that runs on each node in a Kubernetes cluster. It maintains network rules on nodes, allowing network communication to Pods from inside or outside the cluster. Kube-Proxy uses iptables or IPVS to manage the network rules and route traffic to the appropriate Pods based on the defined Services.

4 NETWORK POLICIES

Kubernetes Network Policies are essential for securing and controlling the communication between the different components of a Kubernetes cluster. By defining and enforcing network rules, administrators can ensure that only authorized traffic is allowed, enhancing the security and reliability of the applications [6].

At its core, a Network Policy is a set of rules that define how Pods can communicate with each other and with other network endpoints. These rules are defined using YAML or JSON configurations and provide fine-grained control over network traffic within a Kubernetes cluster. Network Policies work at the OSI layer 3 and layer 4, allowing administrators to specify rules based on IP addresses, ports, and protocols.

Network Policies consist of several key components:

1. **Pod Selector**: Specifies the group of Pods to which the policy applies. This is done using labels, which are key-value pairs attached to the Pods.
2. **Ingress and Egress Rules**: Defines the traffic allowed into and out of the selected Pods. Ingress rules control incoming traffic, while egress rules control outgoing traffic.

3. **Policy Types**: Indicate whether the policy applies to ingress traffic, egress traffic, or both. Specifying Policy Types helps clarify the scope of the policy.

The example of a Network Policy that allows ingress traffic from a specific Namespace is presented here:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-frontend
  namespace: default
spec:
  podSelector:
    matchLabels:
      role: backend
  policyTypes:
  - Ingress
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          role: frontend
    ports:
    - protocol: TCP
      port: 80
```

This YAML file structure is specific to Kubernetes. YAML (YAML Ain't Markup Language) files are used to define the desired state of various resources within the cluster. These files provide a declarative way to manage Kubernetes objects, such as Pods, Services, Deployments, etc [7].

The **apiVersion** field specifies the version of the Kubernetes API that the resource uses. Different resources might have different API versions, like: v1, apps/v1, networking.k8s.io/v1.

The **kind** field defines the type of Kubernetes object that is being described (Pod, Service, Deployment, NetworkPolicy…).

The **metadata** section contains information about the resource, such as its name, namespace, and labels. This section helps in organizin and identifying different resources within the cluster.

The **spec** section defines the desired state of the resource. This is where the resource's configuration details are specified. For example, in a Pod YAML file, the spec section would define the containers to run, their images and other relevant settings

The YAML file presented in the example is explained in the following section:

1. apiVersion

The **apiVersion** is set to networking.k8s.io/v1, indicating that this resource uses the networking API version 1.

2. kind

The **kind** is NetworkPolicy, specifying that this YAML file defines a Network Policy resource.

3. metadata

The **metadata** section includes the name and namespace of the Network Policy. Here, the policy is named allow-frontend and is applied to the default namespace.

4. spec

The spec section outlines the details of the Network Policy.

podSelector: This selector targets Pods with the label role: backend. Only these Pods will have the defined network policy rules applied.

policyTypes: Specifies that this policy applies to ingress traffic (incoming traffic). The other type, egress (outgoing traffic), is not defined here.

ingress: Defines the rules for incoming traffic.

from: This block specifies the sources allowed to send traffic to the selected Pods. In this case, it allows traffic from Pods in namespaces with the label role: frontend.

ports: Specifies the protocol and port that the traffic is allowed on. Here, it allows TCP traffic on port 80 [8].

This Network Policy allows Pods labeled role: frontend in any namespace to communicate with Pods labeled role: backend in the default namespace on port 80 using the TCP protocol.

5 ADVANCED CONFIGURATION

Advanced network policies let administrators define more sophisiticated and granular rules to secure and control traffic within the cluster. These configurations often involve the use of additional tools and integrations for enhanced functionality [9].

Advanced network policies can specify granular rules based on specific labels, namespaces, and external IP addresses. An example Network Policy that incorporates multiple selectors is presented here:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: advanced-policy
  namespace: default
spec:
  podSelector:
    matchLabels:
      role: backend
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          role: frontend
    - podSelector:
        matchLabels:
          role: security
    ports:
    - protocol: TCP
      port: 443
  egress:
  - to:
    - ipBlock:
        cidr: 192.168.1.0/24
    ports:
    - protocol: UDP
      port: 8080
```

The spec section outlines the specifics of the Network Policy:

podSelector: This selector determines the group of Pods to which this policy applies. Here, it targets Pods with the label role: backend. Only these Pods will be subject to the defined rules.

policyTypes: Lists the types of traffic the policy applies to. In this case, the policy controls both ingress (incoming) and egress (outgoing) traffic.

ingress: Defines the rules for incoming traffic to the selected Pods.

from attribute specifies the sources of this traffic:

namespaceSelector: Allows traffic from Pods in namespaces that have the label role: frontend.

podSelector: Allows traffic from Pods with the label role: security.

ports: Specifies that this traffic is only allowed on TCP protocol over port 443, which is typically used for HTTPS traffic.

egress: Defines the rules for outgoing traffic from the selected Pods.

to attribute specifies the destination for this traffic

ipBlock: Allows traffic to the IP range 192.168.1.0/24.

ports: Specifies that this traffic is only allowed on UDP protocol over port 8080 [10].

5.1 MULTI-CLUSTER AND HYBRID CLOUD NETWORKING

While the paper has focused on single-cluster setups, modern enterprises often deploy applications across multiple clusters in hybrid or multi-cloud environments (e.g., AWS EKS, Google Anthos, Azure AKS). Managing network policies in such scenarios introduces challenges like cross-cluster communication, consistent policy enforcement, and integration with cloud-native networking solutions.

Key considerations include:

- Service Mesh Integration: Tools like Istio or Linkerd can enforce policies across clusters by abstracting network layers.
- Global Network Policies: Solutions like Calico's Tigera Secure allow administrators to define policies spanning multiple clusters.
- Cloud-Specific Networking: AWS VPC peering, Google Cloud Interconnect, or Azure Virtual WAN enable secure cross-cluster communication.
- Hybrid Environments: Bridging on-premises Kubernetes clusters with public cloud clusters requires VPNs or SD-WAN solutions.

Example use case: A policy allowing ingress traffic from an on-premises cluster (CIDR: 10.0.0.0/16) to a cloud-based cluster (CIDR: 192.168.0.0/16) can be defined using Calico's GlobalNetworkPolicy:

```
apiVersion: projectcalico.org/v3
kind: GlobalNetworkPolicy
metadata:
  name: cross-cluster-allow
spec:
  selector: role == 'cloud-service'
ingress:
  - action: Allow
    source:
      nets: [10.0.0.0/16]
namespaceSelector: has(environment) && environment in {'prod'}
```

This approach ensures policies scale seamlessly across heterogeneous environments.

6 FLANNEL

Flannel is an open-source virtual network designed specifically for Kubernetes. It provides a simple and easy way to configure layer 3 IPv4 network between multiple nodes in a Kubernetes cluster [11].

Flannel creates an overlay network that abstracts the underlying hardware network. This allows containers to communicate across a cluster with thir own unique IP addresses, and it also integrates with CNI.

Each node in the Flannel network is assigned a subnet, forming the basis of IP address allocation for containers running on that node.

Also, Flannel supports various backend mechanisms for packet forwarding, including VXLAN and host-gw. VXLAN runs a Layer 2 network on top of a Layer 3 infrastructure, while host-gw maps direct routes between the hosts.

Figure 3 shows Kubernetes architecture with Flannel network interfaces.

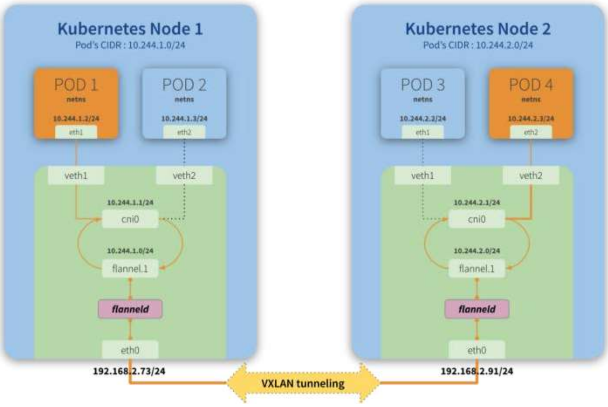

Figure 3 Showcase of Kubernetes architecture with Flannel network interfaces

Flannel is set up in Kubernetes cluster by running the *kubectl apply -f https://github.com/flannel-io/flannel/releases/latest/download/kube-flannel.yml* command from the terminal.

7 CALICO

Calico is a networking and security solution that enables Kubernetes workloads and non-Kubernetes/legacy workloads to communicate seamlessly and securely.

Calico consists of networking to secure communication, and advanced network policy to secure cloud-native microservices or applications at scale.

Calico CNI is a control plane that programs several dataplanes. It is an Layer 3/Layer 4 networking solution that secures containers, Kubernetes clusters, virtual machines, and native host-based workloads [12].

Calico network policy suite is an interface to the Calico CNI that contains rules for the dataplane to execute.

Calico network policy is designed with a zero-trust security model (deny-all, allow only where needed). It integrates with the Kubernetes API server, so that the Kubernetes network policy can still be used,and it also supports legacy systems (bare metal, non-cluster hosts) using the same network policy model.

Calico supports multiple encapsulation methods, includin IP-in-IP and VXLAN, which help in creating efficient overlay networks.

It can also use BGP (Border Gateway Protocol) to distribute routes, integrating easily with on-premises and cloud-based network infrastructure.

Calico can be installed with the following command *kubectl create -f https://raw.githubusercontent.com/projectcalico/calico/v3.29.1/manifests/tigera-operator.yaml.*

Once Calico is properly installed, its advanced features can be used.

The following example shows enabling of IP-in-IP encapsulation in Calico network:

```
apiVersion: projectcalico.org/v3
kind: IPPool
metadata:
```

```
  name: default-ip-pool
spec:
  cidr: 192.168.0.0/16
  ipipMode: CrossSubnet
  natOutgoing: true
```

cidr: Specifies the IP range for the pool.

ipipMode: Set to CrossSubnet, meaning IPIP encapsulation will be used only for traffic between different subnets.

natOutgoing: Enables NAT for outgoing traffic, which is useful for masquerading pod IPs behind node IPs.

The next example shows setting up BGP with Calico. Calico uses BGP to distribute routing information between networks, enabling high-performance networking without the need for overlays like IPIP or VXLAN in certain environments.

The following YAML file shows how to create BGPPeer resource for establishing BGP peering between Calico and an external router.

```
apiVersion: projectcalico.org/v3
kind: BGPPeer
metadata:
  name: node-to-router
spec:
  peerIP: 203.0.113.1
  asNumber: 64512
```

peerIP: The IP address of the external BGP peer (e.g., a top-of-rack router).

asNumber: The Autonomous System (AS) number assigned to the BGP peer.

After defining the BGPPeer resource, ASN needs to be defined for each Calico node. Also, BGP peering needs to be enabled.

```
apiVersion: projectcalico.org/v3
kind: Node
metadata:
  name: node-hostname
spec:
  bgp:
    asNumber: 64512
```

BGP peering status can be checked by running the command *calicoctl node status*.

8 PERFORMANCE EVALUATION

To quantify the impact of network policies and CNI choices, benchmarks measuring latency, throughput, and resource overhead were conducted.

A 10-node Kuberentes cluster (AWS EC2 t3.medium instances) was deployed and both CNIs were tested using iperf3 and kubebench.

Table 2: The results of Flannel and Calico testing

Metric	Flannel (VXLAN)	Calico (BGP)
Latency (avg)	2.3ms	1.8ms
Throughput	4.2 Gbps	5.1 Gbps
CPU Overhead	12%	18%

Calico’s BGP mode offers lower latency and higher throughput but consumes more CPU.

Flannel’s simplicity suits small clusters, while Calico excels in performance-critical environments.

The impact of Network Policies on performance was tested on the following way: Strict egress policies were applied to a pod handling 1000 requests per second.

Table 3: The results of Network Policy impact testing

Policy Strictness	Latency Increase	Throughput Drop
No Policies	0%	0%
10 Egress Rules	8%	5%
50 Egress Rules	22%	18%

The usage of label selectors consolidates rules and enables avoiding excessive granularity when defining Network Policies.

9 CONCLUSION

The implementation of Kubernetes network policies is paramount for securing and managing network traffic within a Kubernetes cluster. Through the detailed exploration of concepts and tools, this paper demonstrates the importance and effectiveness of network policies in maintaining a robust, secure, and efficient cloud-native environment.

By leveraging Kubernetes' native capabilities and integrating advanced networking solutions like Calico and Flannel, organizations can achieve fine-grained control over network communications. These tools enable administrators to define and enforce policies that safeguard sensitive data, optimize performance, and ensure regulatory compliance.

Network policies restrict unauthorized access, protect sensitive data and applications, and reduce latency to improve overall application performance. Properly implemented network policies help organizations meet industry standards, instilling customer trust and confidence.

Looking ahead, the ongoing evolution of Kubernetes and its ecosystem promises even more sophisticated tools and techniques for network management and security. As organizations continue to adopt and expand their use of Kubernetes, staying informed about these developments will be crucial for maintaining secure and performant environments.

In summary, the strategic use of Kubernetes network policies is essential for any organization aiming to harness the full potential of cloud-native applications. By understanding and implementing these policies, administrators can create resilient, secure, and efficient Kubernetes clusters that support the dynamic needs of modern enterprises.

REFERENCES

[1] K. Li, X. Xiao, C. Gao, S. Yu, X. Tang and G. Tan, "Implementation of High-Performance Automated Monitoring Collection Based on Kubernetes,"2024 3rd International Conference on Cloud Computing, Big Data Application and Software Engineering (CBASE), Hangzhou, China, 2024, pp. 838-843, doi: 10.1109/CBASE64041.2024.10824649.

[2] H. Jeong and S. Pack, "An Implementation Study of 3GPP Network Data Analytics Function on Kubernetes," 2024 15th International Conference on Information and Communication Technology Convergence (ICTC), Jeju Island, Korea, Republic of, 2024, pp. 1931-1933, doi: 10.1109/ICTC62082.2024.10827460.

[3] M. Usman, S. Ferlin, and A. Brunstrom, "Performance Analysis of Lightweight Container Orchestration Platforms for Edge-Based IoT Applications," 2024 IEEE/ACM Symposium on Edge Computing (SEC), Rome, Italy, 2024, pp. 321-332, doi: 10.1109/SEC62691.2024.00032.

[4] J. Yin, Y. Zhao and H. Wang, "A Static Task Allocation and Scheduling Algorithm for Kubernetes Cluster," 2024 IEEE 7th International Conference on Information Systems and Computer Aided Education (ICISCAE), Dalian, China, 2024, pp. 175-179, doi: 10.1109/ICISCAE62304.2024.10761792.

[5] H. Zhou and C. H. Yong, "Implement HPA for Nginx Service Using Custom Metrics Under Kubernetes Framework," IEEE Access, vol. 12, pp. 189722-189734, 2024, doi: 10.1109/ACCESS.2024.3509876.

[6] K. Islam, S. F. Hassan and A. Orel, "An Architecture for Edge Driven Networks," 2024 International Symposium on Networks, Computers and Communications (ISNCC), Washington DC, DC, USA, 2024, pp. 1-5, doi: 10.1109/ISNCC62547.2024.10758977.

[7] S. Koksal, F. O. Catak, and Y. Dalveren, "Flexible and Lightweight Mitigation Framework for Distributed Denial-of-Service Attacks in Container-Based Edge Networks Using Kubernetes,"IEEE Access, vol. 12, pp. 172980-172991, 2024, doi: 10.1109/ACCESS.2024.3501192.

[8] K. P. Sah, N. Jain, P. Jha, J. Hawari and B. M. Beena, "Advancing of Microservices Architecture with Dockers," 2024 15th International Conference on Computing Communication and Networking Technologies (ICCCNT), Kamand, India, 2024, pp. 1-6, doi: 10.1109/ICCCNT61001.2024.10724035.

[9] G. Koukis, S. Skaperas, I. A. Kapetanidou, L. Mamatas and V. Tsaoussidis, "Performance Evaluation of Kubernetes Networking Approaches across Constraint Edge Environments," 2024 IEEE Symposium on Computers and Communications (ISCC), Paris, France, 2024, pp. 1-6, doi: 10.1109/ISCC61673.2024.10733726.

[10] N. T. Nguyen and Y. Kim, "A Design of Resource Allocation Structure for Multi-Tenant Services in Kubernetes Cluster," 2022 27th Asia Pacific Conference on Communications (APCC), Jeju Island, Korea, Republic of, 2022, pp. 651-654, doi: 10.1109/APCC55198.2022.9943782.

[11] flannel-io, 'flannel,' GitHub Repository, [Online]. Available: https://github.com/flannel-io/flannel

[12] Tigera, 'About Calico,' Calico Documentation, [Online]. Available: https://docs.tigera.io/calico/latest/about/

Contact information:

Sava STANISIC 1, MSc
(Corresponding author)
2000.
Faculty of Technical Sciences, Cacak
Svetog Save 65, Cacak
sava.stanisic@vs.rs
https://orcid.org/0009-0002-3118-0537

Vladimir MLADENOVIC 2, PhD
1975.
Faculty of Technical Sciences, Cacak
Svetog Save 65, Cacak
vladimir.mladenovic@ftn.kg.ac.rs
https://orcid.org/0000-0001-8530-2312