UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Machine Learning for improved communication between Kubernetes - etcd

Diploma Thesis

Ilias Iliadis

Supervisor: Athanasios Korakis

Month 2022

UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Machine Learning for improved communication between Kubernetes - etcd

Diploma Thesis

Ilias Iliadis

Supervisor: Athanasios Korakis

Month 2022

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Μηχανική Μάθηση για Καλύτερη Επικοινωνία Kubernetes - etcd

Διπλωματική Εργασία

Ηλίας Ηλιάδης

Επιβλέπων/πουσα: Αθανάσιος Κοράκης

Μήνας 2022

Approved by the Examination Committee:

Supervisor **Athanasios Korakis**
Professor, Department of Electrical and Computer Engineering, University of Thessaly

Member **Antonios Argyriou**
Associate professor, Department of Electrical and Computer Engineering, University of Thessaly

Member **Parisis Flegkas**
Assistant Professor, Department of Electrical and Computer Engineering, University of Thessaly

Acknowledgements

Θα ήθελα να εκφράσω την ευγνωμοσύνη μου στους ανθρώπους που συνέβαλαν στο να πραγματοποιηθεί αυτή η διπλωματική, σε όλους όσους στάθηκαν στο πλευρό μου σε αυτό δύσκολο αλλά και συναρπαστικό ταξίδι των σπουδών μου. Εν πρώτοις θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή μου κύριο Αθανάσιο Κοράκη, που μέσα από τα μαθήματά του καλλιέργησε το ενδιαφέρον μου για τον τομέα των Δικτύων και μου έδωσε την ευκαιρία της τριβής με την έρευνα και την τεχνολογία μέσω της συμμετοχής μου σε projects της συγκεκριμένης ειδικότητας. Επιπλέον, θα ήθελα να ευχαριστήσω θερμά τον κύριο Κωσταντίνο Χούμα, για τις συμβουλές και την καθοδήγηση του, όπως και για το ότι ήταν πάντα διαθέσιμος και πρόθυμος να με βοηθήσει σε οποιαδήποτε δυσκολία κι αν αντιμετώπισα κατά την εκπόνηση αυτής της διπλωματικής. Θέλω να ευχαριστήσω από καρδιάς την οικογένεια και τους φίλους μου για την ανεκτίμητη στήριξη, την αγάπη και την βοήθεια που μου πρόσφεραν, ώστε να σταθώ στα πόδια μου και να αντιμετωπίσω όλες τις δυσκολίες κατά τη διάρκεια των σπουδών μου. Χωρίς αυτούς δεν θα έφτανα στο σημείο αυτό.

DISCLAIMER ON ACADEMIC ETHICS AND INTELLECTUAL PROPERTY RIGHTS

«Being fully aware of the implications of copyright laws, I expressly state that this diploma thesis, as well as the electronic files and source codes developed or modified in the course of this thesis, are solely the product of my personal work and do not infringe any rights of intellectual property, personality and personal data of third parties, do not contain work / contributions of third parties for which the permission of the authors / beneficiaries is required and are not a product of partial or complete plagiarism, while the sources used are limited to the bibliographic references only and meet the rules of scientific citing. The points where I have used ideas, text, files and / or sources of other authors are clearly mentioned in the text with the appropriate citation and the relevant complete reference is included in the bibliographic references section. I also declare that the results of the work have not been used to obtain another degree. I fully, individually and personally undertake all legal and administrative consequences that may arise in the event that it is proven, in the course of time, that this thesis or part of it does not belong to me because it is a product of plagiarism».

The declarant

Ilias Iliadis

Diploma Thesis

Machine Learning for improved communication between Kubernetes - etcd

Ilias Iliadis

Abstract

With the rise of microservices, organizations have widely adopted containers for application deployment. Currently, applications can be made up of hundreds of containers that can probably operate in different environments. Due to the increased size and complexity of the production environment, developers require improved tooling to manage the load of containers, as well as to automate application scheduling onto them. These are some of the industry's requirements that resulted in the development of container orchestration technologies, with Kubernetes being the most popular framework to manage containerized applications. In order to function properly, kubernetes is using etcd as it's primary back storage for the cluster's metadata, which is a reliable, consistent, and distributed database with no downtime. However, when the etcd cluster is deployed externally, variables such as TLS encryption, distance between endpoints, and transmission latency caused by excessive load on etcd servers may affect the latency overhead on the communication network between kubernetes and etcd. To address these problems, this project extends the default load balancing policy used for routing client requests to the etcd servers in order to improve the communication link's performance and efficiency by forwarding all client requests directly to the leader of the etcd cluster. We also use machine learning methods to detect high network latencies and excessive workload on the etcd servers to provide reliability on the extended policy. Our designed implementation was deployed and assessed on the NITOS research facility, where we used physical nodes to conduct experiments under real-world environment conditions. Our findings indicate that our approach is faster and more reliable than the default storage method for kubernetes.

Keywords:

Transport Layer Security (TLS)

Διπλωματική Εργασία

Μηχανική Μάθηση για Καλύτερη Επικοινωνία Kubernetes - etcd

Ηλίας Ηλιάδης

Περίληψη

Με την άνοδο των microservices, έχει υιοθετηθεί ευρέως η χρήση των containers για ανάπτυξη εφαρμογών. Τον τελευταίο καιρό εφαρμογές μπορεί να αποτελούνται απο εκατοντάδες containers οι οποίοι πιθανότατα να λειτουργούν σε διαφορετικά περιβάλλοντα εκτέλεσης. Λόγω του αυξανόμενου μεγέθους και πολυπλοκότητας του χώρου παραγωγής, προγραμματιστές απαιτούν αναβαθμισμένα εργαλεία για την διαχείριση του πλήθους των containers όπως και για την αυτοματοποιημένη ανάθεση εκτέλεσης εφαρμογών σε αυτά. Τα προηγούμενα αποτελούν μέρος των συνολικών απαιτήσεων της βιομηχανίας που κατέστησαν αναγκαία την δημιουργία τεχνολογιών για την οργάνωση των containers, εκ των οποίων το kubernetes να αποτελεί την δημοφιλέστερη πλατφόρμα για την διαχείριση εφαρμογών σε containers. Για την σωστή λειτουργία του, το kubernetes χρησιμοποιεί το etcd ώς τον βασικό χώρο αποθήκευσης για τα δεδομένα του cluster, το οποίο είναι μια αξιόπιστη, συνεπής και κατανεμημένη βάση δεδομένων. Παρ' όλα αυτά, στην περίπτωση όπου το etcd εγκαθιστάται εξωτερικά του kubernetes, παράγοντες όπως TLS κρυπτογράφιση, απόσταση μεταξύ των κόμβων και καθυστέρηση μετάδοσης δεδομένων λόγω υπερφόρτωσης των etcd διακομιστών μπορούν να συμβάλλουν στην συνολική καθυστέρηση στο μέσο επικοινωνίας μεταξύ του kubernetes και του etcd. Για την αντιμετώπιση αυτών των προβλημάτων, η παρούσα εργασία επεκτείνει την προεπιλεγμένη πολιτική εξισσορόπησης φορτίου η οποία χρησιμοποιείται για την δρομολόγηση αιτημάτων πρός τους etcd διακομιστές ούτως ώστε να βελτιωθεί η απόδοση και η ταχύτητα επικοινωνίας ανακατευθύνοντας όλα τα αιτήματα στον αρχηγό του etcd cluster. Επίσης γίνεται χρήση μεθόδων μηχανικής μάθησης για τον εντόπισμο υψηλού φόρτου εργασίας των etcd διακομιστών παρέχοντας έτσι αξιοπιστία στην εκτεταμένη πολιτική. Η υλοποίηση της εργασίας αναπτύχθηκε και αξιολογήθηκε χρησιμοποιώντας την ερευνητική εγκατάσταση NITOS, απο την οποία δεσμεύτηκαν πραγματικά μηχανήματα για να διεξαχθούν πειράματα σε πραγματικό περιβάλλον. Τα αποτελέσματα που παράχθηκαν δείχνουν ότι η δική μας προσέγγιση είναι γρηγορότερη και πιο αξιόπιστη απο την προεπιλεγμένη μέ-

θοδο αποθήκευσης του kubernetes.

Λέξεις-κλειδιά:

Table of contents

List of figures

List of tables

Abbreviations

API	Application Programming Interface
IP	Internet Protocol
RPC	Remote Procedure Call
TCP	Transmission Control Protocol
URL	Uniform Resource Locator

Chapter 1

Introduction

1.1 Motivation

Modern application deployment increasingly depends on container technology, which provides an ideal host for small independent applications like microservices. In order to manage and schedule these containers in different production environments on a wide scale, industries rely on container orchestration tools, with Kubernetes being the most preferred one. When Kubernetes is deployed, it generates a cluster of control plane nodes that manage the entire cluster and worker nodes that host the execution of containerized applications. Kubernetes also uses etcd as its primary back store for cluster object configuration data and data providing information about the cluster's status. When it comes to production deployments, a resilient control plane is required in order to operate a highly available (HA) kubernetes cluster and it's achieved by running control plane instances on multiple nodes. An HA cluster's default architecture consists of an odd number of control plane nodes, each of them hosting a member of the etcd cluster that communicates solely with the node's API server component. This stacked topology enables direct and scalable interaction between the etcd cluster and the Control Plane replicas. However, if a master node instance fails, both control plane components and the underlying etcd member are also lost. In this case, it is advisable to consider decoupling the etcd layer from the control plane replicas by setting up the etcd cluster on separate machines. When compared to the stacked topology, the deployment of a standalone etcd cluster achieves higher level of reliability and resiliency since master node failures have less impact on the system's redundancy. Even so, the intermediate network infrastructure that establishes communication between the two cluster architectures may introduce latency

issues that must be addressed.

1.2 Thesis subject

When deploying an etcd cluster on separate architecture from kubernetes, several problems might arise, including security concerns, complex network configuration and communication latency. In this thesis project, we investigate the major factors that contribute to the latency overhead of the communication between these two deployments and address them by extending the default load balancing policy that is used by the kubernetes API server for routing requests to etcd cluster members, which is based on Round Robin. The introduced policy forwards all client requests to the leader node of the external etcd cluster in order to reduce the average response time and improve the performance and efficiency of the communication interface. We also use machine learning methods to detect high network latencies and excessive workload on the etcd servers to provide reliability on the extended policy. The designed implementation was deployed and evaluated in a real-environment topology of physical nodes.

1.2.1 Contribution

To give solutions to the aforementioned issues, we first performed a a thorough analysis of every entry contained in the examined topology, including kubernetes software components and the properties of the etcd datastore. Subsequently, in the NITOS Indoor testbed, we deployed the external etcd cluster and the kubernetes cluster to evaluate the performance of our extended load balancing policy, along with a third-party programm that monitors the status of the etcd leader and elects a new leader if the integrated machine learning model predicts an unstable node status. Finally we assessed the performance of our implementation compared to the default configuration of the examined system using software monitoring tools and metrics extracted from the etcd members.

1.3 Thesis Content Organization

The thesis study is organized into seven Chapters to comprehend various concepts of the examined deployment environment and the relevant problems that this projects aims to re-

solve. In Chapter 2 we describe the kubernetes framework along with all its components and technologies when it is deployed on an containerized environment. In Chapter 3 we analyse the etcd data storage, it's properties and software tools, as well as it's integration with kubernetes through the deployment options to serve as the kubernetes backing store. This Chapter also dives into the etcd client API, which is used by the kubernetes API server to interact with the available etcd servers. Chapter 4 presents the components, properties and functionality of the Raft protocol, which is used by the etcd to guarantee consistency over the data stored in the etcd servers. Chapter 5 details the experimental environment along with all software tools that we utilized to deploy our designed implementation and conduct evaluation experiments. Chapter 6 presents the deployed implementation on the experimental infrastructure, and the experimental results from our extended load balancing policy. The last Chapter reviews our thesis project and suggests future work ideas that could expand the project.

Chapter 2

Kubernetes Ecosystem

Kubernetes is an open-source container orchestration framework [2], which name derives from Greek which translates to “helmsman” or “pilot”. On the foundation it is designed to manage containerized applications. It provides automated and declarative infrastructure management as well as flexible and efficient handling of these applications across different environments including cloud-naive environments or even hybrid ones. It’s vast community constantly contributes to the development and enhancement of the framework. It was developed by Google and released in 2014.

2.1 Chronicle of Application Deployment

Before jumping into the importance of Kubernetes, we must describe how application deployment changed from physical servers to containerization [2]:

- **Traditional deployment era:** Applications were initially deployed by organisations on physical servers. Unfortunately this can initiate resource contention caused by multiple applications deployment on a single physical computer. Some applications would have high resource demands which resulted in degraded performance of the other applications. This scenario made necessary to distribute the applications across different physical servers in terms of performance and scalability. Unfortunately, this lead to scaling issues as servers ran at a small fraction of their capacity making resources underutilized. Also the maintenance of multiple physical servers came at a high cost.
- **Virtualized deployment era:** In order to resolve these issues, server virtualization was introduced. It allows concurrent execution of several virtual servers (VMs) on a single

physical server independently that share the hardware resources of the host system, which is the most important advantage of the technology. A software layer named hypervisor , which lies between the virtual machines and the original hardware, provides each virtual machine a virtualized copy of the resources and manages the virtual machines and controls their resource usage. It also enforces resource usage policies on the virtual machines, and the way they will communicate with each other and with the physical servers system. By doing so the hypervisor guarantees isolation between the virtual machines and the hardware as it allows each virtual machine to run it's own operating system instance and applications, without affecting the rest of the virtual machines in the system. Furthermore, hypervisor ensures that each virtual machine will utilize an amount of resources that will not lead to starvation of the resources. In terms of security, all virtual machines are monitored for malicious activity, unauthorized changes and accesses between them or the host system. If any of the previous actions occurs, the hypervisor will act accordingly by blocking , shutting down or alerting the specific virtual machine. Virtualization improves significantly the performance of the applications by optimizing the resource distribution and efficiency among the virtual machines. On major drawback is the high requirement of each virtual machine for operating system and hardware instances, resulting in fast resource utilization of the system.

- **Container deployment era:** When it comes to resource isolation and allocation, containers and virtual machines have similar functionality. The marked difference between them is that the virtual machines virtualize the entire hardware layer of the host system and containers only virtualize the operating system. Multiple containers are able to run on the same physical machine and work on the same operating system through this technology, making them more portable and efficient across different computing environments. As they are separated from the hardware layer, containers are considered lightweight because they utilize less space than the virtual machines and thus they can handle more applications with lower resource demands.

Some of the significant benefits provided by containers that made them popular are the following [2]:

- Fast Deployment and creation: Containers provide flexible and efficient way of image

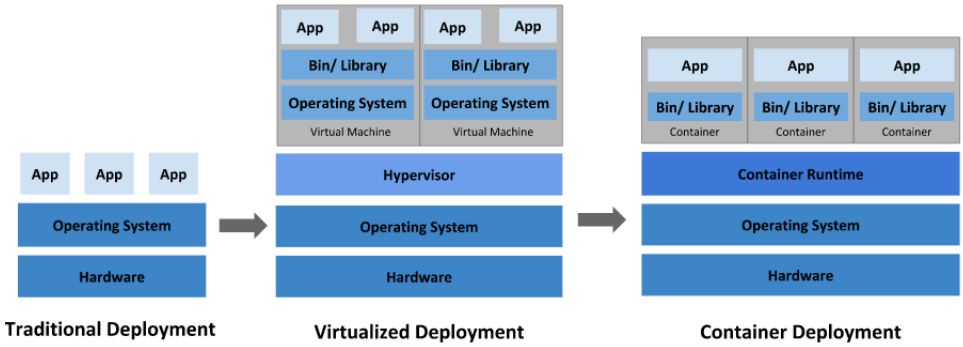


Figure 2.1: Layer architecture of every deployment technology

creation in comparison with virtual machine images.

- Versioning and Rollbacks: container image versioning, enables quick roll back to previous versions in order to track changes. This version control provides reliable and reproductive image build and deployment across different platforms.

- DevOps Enablement: developers can create their applications by building the corresponding container image and then these containers can be deployed with the usage of container orchestration tools.

- Observability: Ability to gain insights of container and host system metrics, health of containerized applications, application data visualisation in dashboard and graphs and much more.

- Environmental consistency across all stages: containers ensure enviromental consistency on different stages of the application lifecycle, such as development, testing and production.

- Portability across different platforms: Container portability allows developers to build an application and run it anywhere. Some of these platforms are Ubuntu, CoreOS, RHEL, on major public clouds and on-premises.

- Easy Management: Rather than focusing on running an operating system instance on virtualized hardware, containerized applications are running on the underlying host operating system using logical resources

- Microservices Architecture: Applications can be decomposed into smaller services,

each of them able to run in its container, making it easier to manage and scale them independently

- Isolation: Each container runs independently of others at application level. It prevents conflicts and enhances security among applications.
- Resource Utilization: Containers efficiently utilize the host system resources with low cost, allowing multiple container execution on a single physical machine without conflicts.

2.2 What are the benefits of Kubernetes

Containers have become quite popular and standard technology for application development due to their exceptional advantages described in the previous section. In production environments, it is important to rely on a container-based infrastructure with robust container load management and high availability. For instance, if a container malfunctions or becomes unavailable, the infrastructure must have an automated mechanism to restore data to the latest state and start a new container from the latest state. The need for automated management, scalability and disaster recovery over containerized applications made the integration of container orchestration technologies essential. Kubernetes guarantees these actions by providing a framework including developer tools (patterns) for ease container-based application build and deployment, flexibility, resilient distributed system execution , and more.

Kubernetes includes several capabilities [2] such as:

- **Service discovery and load balancing:**Kubernetes offers build-in load balancing to distribute the network traffic between containers of an application, ensuring efficient resource usage and preventing container overload. It also enables containers to be accessed using a single DNS name or their IP address.
- **Storage orchestration:** Developers are allowed to automatically mount storage volumes to containers of their choice, including local storages and public cloud providers.
- **Automated rollouts and rollbacks:** Kubernetes provides declarative configuration for the deployed containers. Developers can specify the desired state and the framework will achieve the desired state by creating containers, removing existing ones and resource allocation to the new containers.

- **Automatic bin packing:** When providing a cluster of nodes that can execute containerized task to Kubernetes, the amount of CPU and memory usage is specified for each container. Kubernetes is efficiently placing containers into the inserted nodes in order to reach optimal resource utilization.

- **Self-healing:** Kubernetes self-healing capabilities are reducing downtime and improves reliability of containerized applications. It restarts failed containers automatically, replaces containers that malfunction, and terminates unavailable containers. Client's will not be able to access unhealthy containers unless their state recovers.

- **Secret and configuration management:** Kubernetes provides a secure way to manage and store sensitive information, like API keys, certificates, and passwords. Users can update and deploy configuration data and handle secrets using built-in Kubernetes objects without exposing them to the code or in the container images.

2.3 Kubernetes Basic Architecture

Kubernetes distributes and schedules the application containers automatically across a cluster. It comprised of two kinds of worker machines, called nodes: masters and workers. A node can be either virtual machine or a physical computer . The master node is responsible for the cluster management. This implies that it controls the cluster state, manages worker nodes, schedules and scales applications. Worker nodes serve as hosts for the execution of containerized applications. The group of one or more containers assigned to a worker node is called a Pod. Worker nodes and the corresponding Pods are managed and scheduled by the master node. To maintain fault tolerance and high availability in production environments, multiple masters and nodes are initialized whereas the smallest amount of master nodes for proper functinality of the Kubernetes cluster is one.

The two kinds of nodes host several essential Kubernetes Components that determine the cluster functionality and robustness. Kubernetes Components can are categorized into the following groups [3]: Control Plane components and Node components. The following subsection outlines their functionality and their role in management.

2.3.1 Control Plane Components

Components belonging to the Control Plane manage the overall state and orchestration of the cluster. To be precise they serve as the brain of the cluster handling several tasks such as making decisions about scheduling and scaling of containerized applications, enforcing declarative container orchestration to satisfy given states, controlling network traffic between pods, and respond to various cluster events including pod activation when the given number in the replicas filed of a deployment is unsatisfied. These components are able to run across multiple machines in the cluster but for simplicity sake, the default case suggests to setup the entire control plane on the same machine, and to prevent container execution on this machine. The main control plane components are described below [3]:

- **kube-apiserver:** The API server component forms the primary interface for end users and clients to interact with the cluster. The main implementation of this component is the kube-apiserver which handles incoming requests sent to the exposed Kubernetes REST API. Due to the fact that it relies on etcd storage for saving updated cluster states, kube-apiserver scales horizontally by deploying component copies across multiple nodes. The traffic between these instances can be balanced, ensuring reliability and efficient workload distribution across the instances.

- **etcd:** etcd is a highly-available and consistent key-value storage that is utilized to store both configuration data and data describing the overall cluster state. It is recommended to back up the data of Kubernetes objects that are stored on etcd periodically so as to recover from disaster scenarios.

- **kube-scheduler:** The scheduler detects any newly created pod with no worker node for host and finds an available node for them to run on. In order to implement assignment and scheduling operations, the scheduler takes into consideration various factors based on resource requirements, node affinity and anti affinity, interference among workloads, data locality, and constrains.

- **kube-controller-manager:** This component is running various processes called controllers. A controller is a core component which maintains the desired state of the cluster and continuously monitors the resources state through the kube-apiserver. Despite of being separate processes, controllers run as a single binary for complexity reduction. Kube-controller-manager includes controllers such as:

1. Node Controller: Watches the node status across the cluster and takes specific actions when nodes terminate unexpectedly.
2. Job Controller: manages batch job objects, tracks their status , and creates or removes Pods through the kube-apiserver until the task is carried out.
3. Endpoints Controller: this controller is creating a network link between Pods and Services by populating the Endpoint objects.
4. ServiceAccount controller: Creates ServiceAccounts which authenticate applications running on pods and enables them to access Kubernetes API

- **cloud-controller-manager:** This component forms an interface between the cluster and the selected cloud provider's API, integrating cloud-specific logic into the cluster. It acts as an intermediary between the Kubernetes control plane and the cloud provider's API, and it seperates cloud-specific components from the Control Plane components that operate with the cluster. It starts up exclusively controllers suitable to the connected cloud provider. If Kubernetes is executed on a local infrastructure or in a personal computer, the Control Plane will not include a cloud-controller-manager.

  Similar to the previous component, the cloud-controller-manager compiles and unifies multiple control loops, which are compiled together and run as a single binary. Like kube-apiserver, this component can be scaled horizontally by creating more instances to enhance fault-tolerance and to improve performance.

  Controllers that may include cloud provider dependencies are the following:

  1. Node controller: Monitors the node status in the underlying cloud provider's infrastructure and handles node creation and deletion in the cloud based on their current status. For instance, if a node won't respond the controller makes sure if the specific node has been removed.
  2. Route controller: It manages the configuration and the maintenance of routes in the cloud environment so as to expose Kubernetes Services.
  3. Service controller: Manages and configures cloud provider load balancers.

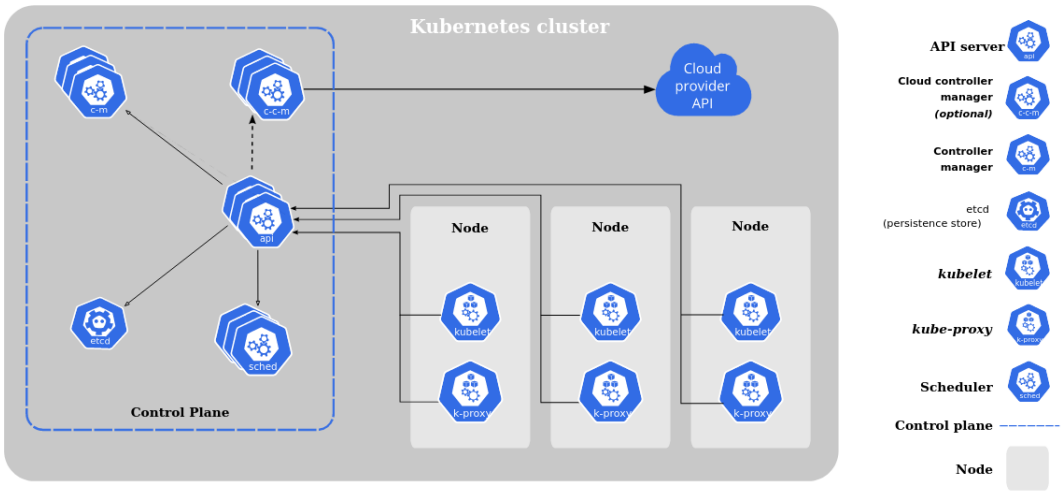


Figure 2.2: Control Plane and Node Components of Kubernetes cluster

2.3.2 Node Components

While the Control Plane manages the overall cluster, node components on the other hand, manage and execute tasks on every worker node in the cluster. They maintain the pod execution on the worker node they are running on and provide the Kubernetes runtime environment. The specified components are following [3]:

- **kubelet:** The kubelet is a process serving as the primary "node agent" which will run on all worker nodes across the Kubernetes cluster. It works according to PodSpecs, which is a file object that includes description of a pod. Using the description of the PodSpec objects, kubelet monitors the health of the containers running on the node. PodSpecs are provided to kubelet primarily through the Kubernetes API server. Moreover, it interacts with the Control Plane receiving updates about the desired state of containers and pods, and instructions. It communicates directly with the container runtime to create, destroy, or update pods and their underlying containers when it is instructed to do so from the Control Plane. Kubelet manages only containers created by Kubernetes.

- **kube-proxy:** This component is running on each cluster node serving as a network proxy. It is responsible for managing network tasks of the worker node it runs on and the main functionality is to provide reliable network communication among pods and services within the cluster. By doing so, it abstracts the Kubernetes Services. Kube-proxy also configures the network policies on the host node to make pods accessible by clients regardless of whether they are located inside of the cluster or outside of it.

This proxy module also interacts with the packet filtering layer of the underlying operating system to route network traffic. If the layer is not provided by the host operating system, the proxy handles traffic without filtering. In addition, it efficiently distributes incoming requests evenly among the services that have multiple replicas or pods.

- **Container runtime:** A container runtime, also reffered as container engine, is the software component that implements container execution on the host node. The container engine takes the responsibility to manage container lifecycle and make actions such as start, stop and restart the container , enforcing resource limits and constrains according to the container resource utilization using cgroup drivers, and ensuring that containers are isolated both from the host system and from each other. The common container runtimes which Kubernetes supports are CRI-O , containered, and container engines implemented by the Kubernetes Container Runtime Interface (CRI).Interfaces provided by CRI are used by kubelet for interaction with the container runtime. It also enables kubelet to use different types of container runtimes concurrently or switching from one to another without having to compile the components of the cluster. This functionality provides the opportunity to find the most appropriate container runtime based on kubelet current version and configuration.

Chapter 3

etcd Key-Value Store

3.1 Introduction

As stated in [4], etcd is an open source, distributed and strongly consistent key-value store that facilitates reliable data storage and management for critical data of ”a distributed system or cluster of machines”. In production enviroments etcd serves as the foundation and main datastore of many projects including Kubernetes, the industry-standard container orchestration technology. Their integration ensures reliable configuration and consistent coordination across the Kubernetes cluster. It is a Raft-based system using the Raft Consensus Algorithm [5] to guarantee data store consistency across all cluster nodes and to provide a fault tolerant distributed environment. Raft provides this consistency by the electing a leader node, which is in charge of distributing data replication to the other nodes, also called followers. Incoming client requests that contain a key-value modification including creation or update are accepted by the leader, which then transmits them to the follower nodes for replication. Once the majority of the followers confirm and acknowledge the incoming request as a log entry, the leader applies the entry, marks it as committed, and send a success message to the corresponding client.

Etcd underlying software mechanism is implemented using the Google programming language, Go [6] and it’s name is derived from the UNIX subdirectory called ”/etc” , where all the system configuration files of a signle UNIX system are located and the ”d” stands for distributed. It was also maintained and developed by the CoreOS community.

3.2 etcd Features

Etcd was created by the developers from scratch with qualities [7] including:

1. Simple: Client applications can interact with the key-value store using common HTTP or JSON tools such as curl to read or write data to etcd, thanks to it's well-defined and user friendly API. It also offers command-line interface (CLI), which makes it simple to communicate with etcd from shell command.

2. Atomic Transactions: Multiple key-value updates can be executed as a single atomic operation through transactions supporded by etcd, which propagate as many changes to the database as ordered. Transactions also make sure that ether every modification is made succesfully inside the unit or the transaction rolls back to its proir state and all preceding activities are stopped at the point where the operation failed.

3. Secure: Automatic Transport Layer Security (Auto TLS) is supporded by etcd whereas secure socket layer (SSL) client certification authentication is optional . Administrators should create role-based access controls [8] within the deployment because etcd maintains crucial and highly sensitive configuration data.

4. Fast: etcd benchmark tools verify that it is capable to make thousands of write operations per instance.

5. Reliably consistent: According to [9], the API ensures strict serializability which is the most robust guarantee in terms of consistency of distributed systems. Etcd clients always read the data with the latest update without taking note of any intermediate date, regardless of which etcd server it sends requests to.

6. Highly available: Due to the Raft-based nature of the etcd system, key-value data is efficiently distributed and replicated from the current leader across the cluster establishing consensus with the Raft algorithm. Replication reflects data to all cluster members and creates redundancy which prevents data loss over single member malfunction. When it comes to failure scenarios [10], an etcd cluster functions as long as the majority of the server nodes are up and running, also called quorum. Additionally, the cluster is divided into majority member and minority member segments if a network partition occurs. If that happens, the majority segment will continue to run in the cluster.

3.3 etcd integration with Kubernetes

The fundamental key-value store for initializing a working, fault-tolernant Kubernetes cluster is etcd, which is one of the main components of Kubernetes Control Plane. Kubernetes requires a distributed data store like etcd, since it is a distributed system. Containerized applications have management requirements [7] that become more complex according to the scaling and the amount of workload, much like any distributed workloads. Kubernetes reduces the complexity of workload management by coordinating operations like service discovery, load balancing, scheduling, health monitoring, deployment and configuration on the entire cluster using the Control Plane, which is portable across multiple machines.

In order to achieve such an efficient management and flexible coordination across the cluster, Kubernetes must rely on a data store that provides a trustworthy source of information about the configuration and the current status of all kubernetes objects and resources used by nodes, pods and their underlying applications at any given time.

Although the API server could simply store its data in a conventional database like MySQL or PostgreSQL, these databases lack of important features which the API server needs for a backbone data storage system. Compared to other databases, etcd covers the need for high-availability and fault tolerance that Kubernetes strives for [11]. To be precise, when a node fails or a network partition occurs , a significant downtime penalty is triggered which makes Kubernetes Control Plane come to a halt. Etcd's architecture , as described in the previous section, supports highly available data through efficient distribution and replication using the Raft algorithm in order to ensure that the cluster will continue to function if a certain member fails. Furthermore, with the API server being the central orchestrator point of the whole Kubernetes cluster forwarding instructions from controllers to pods or nodes, it needs the consistency of data provided by etcd in order to return the latest modified data of the storage and thus the latest cluster state at any given time. Also featuring etcd would prevent unpredictable API server slowdowns generated from concurrent read and write overhead since it has an extremely fast write operation mechanism and read operations can be handled from any of the etcd members due to log replication. Another important feature which plays a crutial role in streaming notifiaction changes to clients is the mechanism called Watch API [12] included in etcd API. It allows Kubernetes components to receive real-time notifications about changes made to configuration data or monitor for changes in etcd storage system. Clients can also subscribe to changes made on a certain key or group of keys (using key prefix).

One more thing worth mentioning, is what the API database does not require from a database. Much of the Kubernetes configuration and state data are stored in format of metadata and configuration information about several Kubernetes instances and resources. Because of this lightweight data format, a large Kubernetes cluster will produce a maximum amount of a few gigabytes of data and thus large and complex datasets like the traditional ones are not needed. The majority of kubernetes objects are accessible by type, namespace and on occasion by name via the Kubernetes API, which has very predictable access patterns. If more filtering is necessary to access a particular key, it will usually be done with the usage of labels and annotations. Because of this behaviour SQL databases, which use complex queries and joins to find data , are excessive for the way the API operates making it hard form them to reach high availability and consistent performance.

3.4 Etcd deployment methods in Kubernetes Clusters

In Kubernetes enviroments, each etcd member can be deployed as a pod in their host master node to enhance communication efficiency with the Control Plane. Figure 3.1 depicts the architecture of the components.

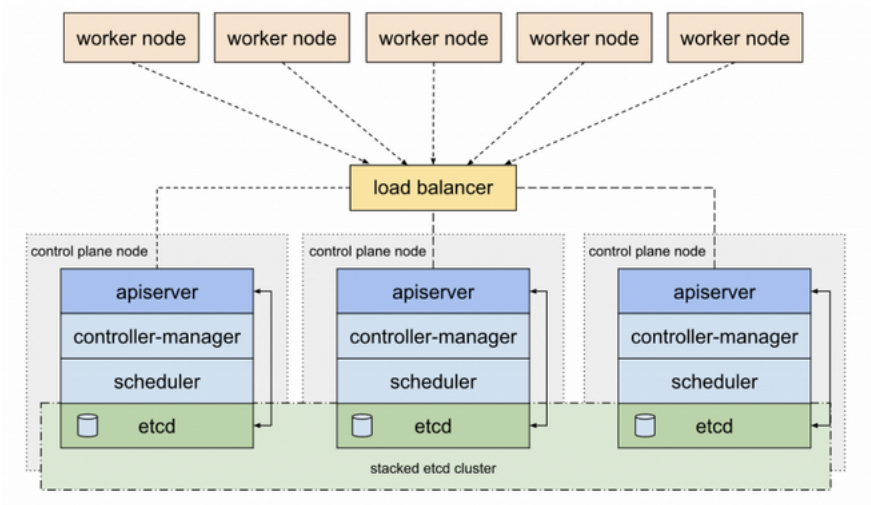


Figure 3.1: etcd and Kubernetes operating in the same cluster

While internal etcd deployment can introduce overhead on the Control Plane and complexity when it comes to manage updates and maintenance operations on etcd, external etcd deployment is introduced as an alternative approach, providing independent management of the standalone etcd cluster and enhanced security and isolation of the used resources. Of

course this deployment has it's own drawbacks too, including security considerations and latency peaks due to the external network infrastructure that lies between Kubernetes API and etcd. Figure 3.2 depicts the topology of this deployment.

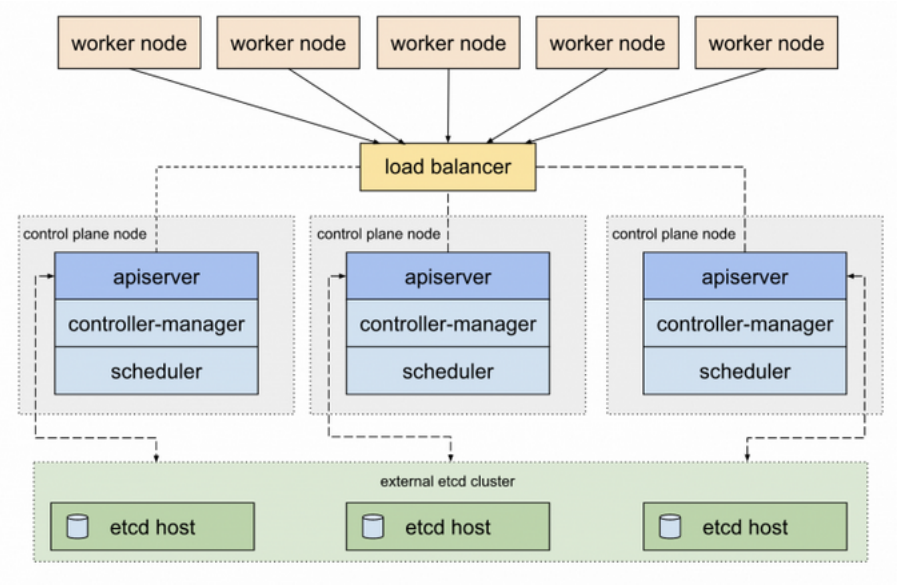


Figure 3.2: etcd external deployment

The decision to whether use internal or external etcd deployment is specified by factors based on the developer priorities on how the cluster will operate and it's infrastructure. For instance major tradeoffs between these deployments include resource isolation over security considerations and network latency overhead or co-located etcd pods with Kubernetes Components for optimized communication performance between them over additional overhead and complexity to Control Plane. In our use case study, we select to deploy etcd externally in order to conduct experiments on the network communication between Kubernetes API and the standalone etcd cluster.

3.5 etcd client architecture

Etcd already handles the application logic on the server side through the guaranteed features discussed in the previous sections, and due to it's Raft-based nature. However, when etcd servers interact with client components, it is necessary to implement a series of protocols that will provide an efficient, reliable and highly available client application logic when the server side malfunctions. This section introduces etcd client implementations and their functionalities. The components contained in the official etcd client implementation are the

following [13]:

- A gRPC Load balancer [14], which creates gRPC connections for RPC traffic distribution to an etcd cluster
- An client application interface that forwards RPC messages to the corresponding etcd server
- An error handler to determine if a failed request should be retried or endpoints should be switched.

The main difference across the released etcd API versions derives from the selected version of the gRPC balancer. Each version provides a different balancing policy:

- **gRPCv1.0 Balancer:** When multiple etcd endpoints are configured by the client, this balancer maintains a TCP connection to each endpoint. It selects one connection to forward all etcd client requests as shown in Figure3.3. Until the client object is closed, the pinned address does not change. When an error is received, the balancer selects another at random and retries the request.

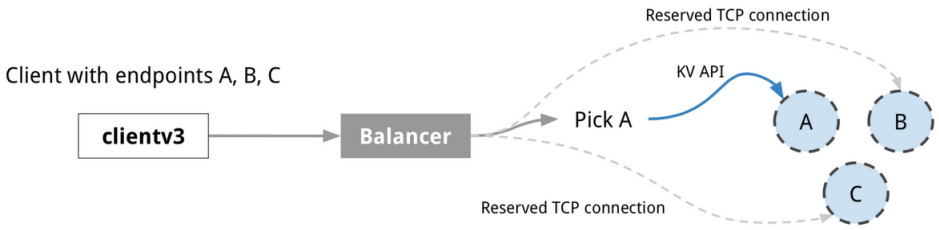


Figure 3.3: gRPCv1.0 Balancer

Although this approach provides fast failover, the maintenance of multiple TCP connections demands more resources. Furthermore, the balancer is unaware of the cluster membership and pinned server's health status.

- **gRPCv1.7 Balancer:** Instead of managing multiple connections, the only connection maintained is the one to the selected etcd server. When the etcd client passes multiple endpoints, the balancer will attempt to interact with every endpoint. When one connection is established, the balancer forwards requests to this endpoint. Until the client object is closed, the pinned address does not change. When an error is returned, it is processed by the client error handler and based on the error code it determines whether

to redirect the request by switching to another address or to retry the request on the same server.

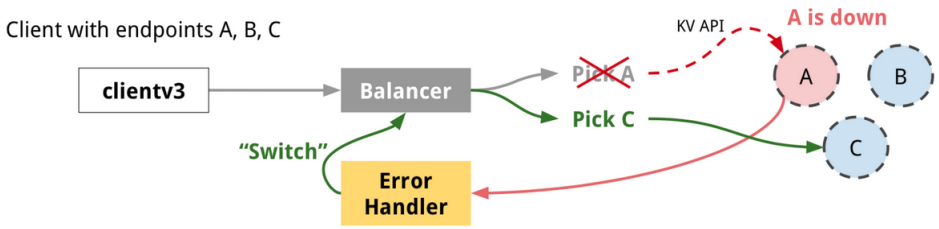


Figure 3.4: gRPCv1.7 Balancer integrated error handler

Furthermore, this balancer keeps track of unhealthy cluster members through a maintained list. When a pinned address is disconnected, it is marked as ”unhealthy” and is appended to the list. The entries are regarded inaccessible for a fixed time interval of 5 seconds called dial timeout [13] and are removed after timeout. On some occasions, however, the list may provide incorrect information about the current health status of the endpoints. For example, server node A in Figure 3.4 may recover immediately after its failure but it won't be utilized because it is marked as ”unhealthy” for the following 5 seconds.

- **gRPCv1.14 Balancer:** Due to the previous balancer's complicated implementation with incorrect server connectivity assumptions, this balancer provides simpler approach for balancer failover. Instead of preserving a list of potentially stale unhealthy server addresses, when the current pinned server address gets disconnected, the balancer just makes a round-robin to the next endpoint [13].Also in normal operation, the balancer forwards client requests through round-robin between available endpoints [14]. Furthermore, in this implementation the round-robin load balancing does not take endpoint status into consideration. Hence, the status tracking list introduced in the previous balancer is not required.

  When several endpoints are given to the balancer, it internally creates a sub-connection for each endpoint, which is an interface of the gRPC mechanism. It may require more resources by retaining this group of TCP connections, but load balancing is more flexible with improved failover performance. This version of the gRPC balancer is used in the latest etcd versions for the etcd client API. In this thesis project, the gRPC balancer's implementation is extended in order to provide a more efficient and optimized type of

balancing. The use case, implementation, and functionality of the altered balancer is described thoroughly in following Chapters.

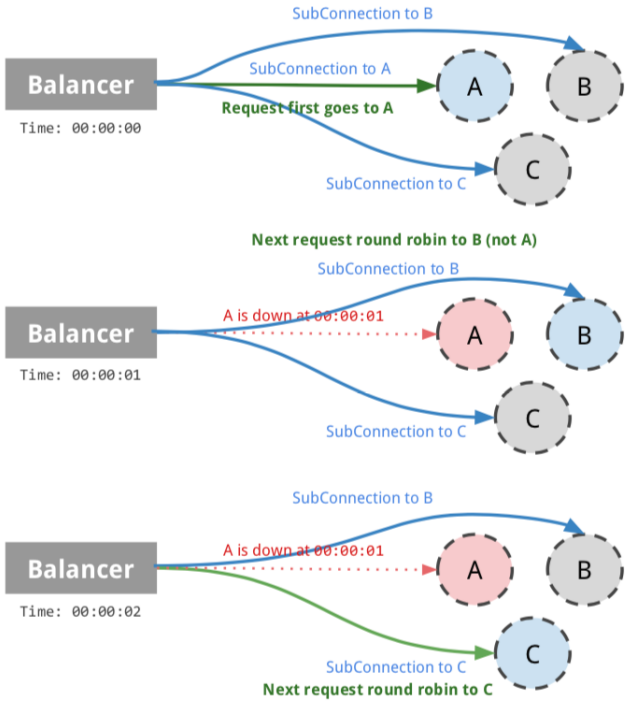


Figure 3.5: gRPCv1.14 roundrobin load balancing policy

Chapter 4

Raft distributed consensus algorithm

4.1 Introduction

A Complex system of interconnected nodes that can each function independently and possibly store distinct copies of the same data forms a distributed system. In a distributed system, nodes very often have to come to joint decisions including agreement on the event order in the system, and maintaining consistent data throughout the system. In order to guarantee that nodes will operate towards the same objective some form of consensus protocol must be established among them.

A widely used mechanism for reaching consensus in distributed systems is the Raft algorithm [1], which was developed by Diego Ongano and John Ousterhout at the Standford University in 2014, and was designed to provide a more understandable and straightforward way of reaching consensus in contrast to it's predecessor, the Paxos algorithm, which was considered the holy grail in achieving consensus. Being an alternative to the Paxos algorithm, it has inherited result correctness, fault-tolerance, and high performance, whereas it introduces a more understandable implementation through it's structure and seperation logic of key elements of consensus including log replication, leader election, and safety.

On the highest level, the Raft protocol is based on leadership, meaning that all read and write client requests forwarded to a system's node for execution must be replicated to the rest of the cluster members, also called followers, through the cluster leader. Follower's client commands are forwarded to the leader in order to handle their execution and to replicate changes to the other nodes of the system. This guarantees a consistent view of the data across all system nodes.

4.2 Understanding consensus

In order to understand raft consensus approach in detail, we have comprehend the consensus problem itself which raft tries to solve. In a distributed system, consensus [15] refers to the coordination requirement of multiple nodes or servers to agree on a specific data value, and once they reach an agreement the decision is immutable. For instance, in the scenario depicted in Figure 4.1, nodes have to agree on a specific transaction request to the systems database. The agreement on a majority value by all servers in the system is one method to reach consensus, meaning that this scenario demands over half of the server votes. However, consensus may not be reached or may be reached inaccurately as a result of some processes malfunctioning or being unreliable in various ways.

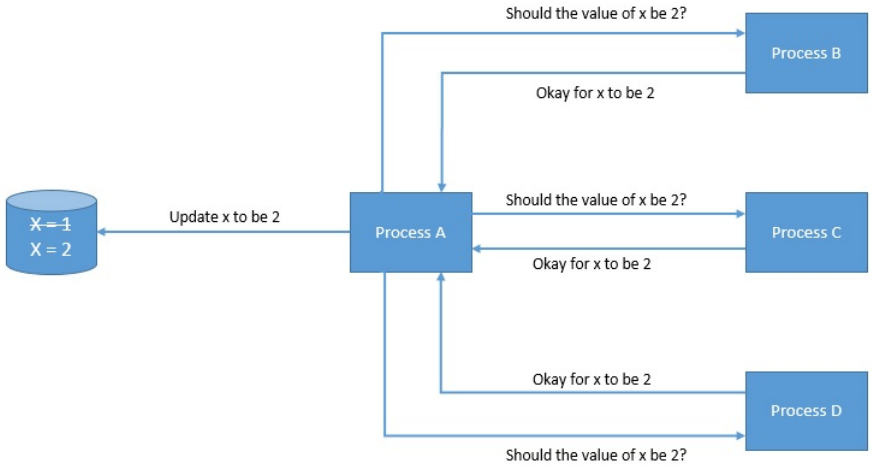


Figure 4.1: Servers in a distributed system reaching consensus for database update

Consensus protocols are designed to maintain consensus and to ensure high fault-tolerance and resilience in a distributed system allowing server nodes to cooperate correctly even when one or more members fail or there are communication problems between them. Common consensus algorithms continue operating as long as at least half of the system servers are up. On the contrary, if the majority of the servers malfunction, the algorithm will stop operating but never produce an incorrect result [16].

4.3 Replicated state machines

In order for a distributed system to keep operating even in the presence of member failures, it is possible to maintain multiple servers in an identical state [1] by using replicated

state machines. In distributed systems, they are used to handle a range of fault-tolerance issues. For instance, in the case that a server crashes, another server can take over inheriting the data and most recent state of the failed server replicated from the state machine.

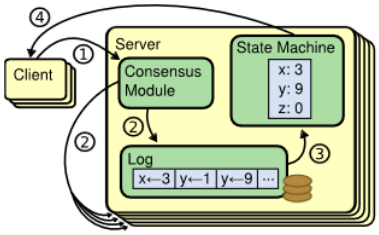


Figure 4.2: Replicated state machine architecture [1]

Replicated state machine provides fault tolerance through its own replication to every server on the system. It's commonly using a replicated log as an input, as shown in Figure 4.2. Every server has a replicated log and a state machine [16]. It is a log used by every server in order to save commands which are successively read as input and executed by the state machine. Replicated state machines operate in the same way due to their deterministic nature. This means that if all replicated state machines across the system use a given sequence with the same ordering as an input, they will all generate the same results and change the system's state in the same way. Furthermore, in order to keep state machines in sync, the commands contained in the log must be identical and in the same order across all replicas.

The component responsible for preserving log consistency on each replica across the servers is called consensus module, and it is deployed on each server. It receives client requests, appends them to the underlying log, and communicates with other consensus modules to guarantee that identical forwarded commands have been inserted into every log in the same order. Subsequently, state machines on each server begin to process the log data once the module certifies that the logs are successfully replicated and then an operation result is sent back to the client. Thus, even if a few nodes in the clusters malfunction or fail, state machines obtain high reliability.

In practical systems, consensus protocols must satisfy the following qualities in order to face consensus problem requirements [15]:

1. **Agreement:** All running nodes must end up selecting an identical value.

2. **Validity:** The resulting value must be proposed by a legitimate running process. This property ensures that at least one of the decision made from the running processes will

be received.

3. **Termination:** Each running process must finally select a value, resulting in algorithm termination.

4. **Integrity:** Once the decision is made, running processes can not change the selected value. It prevents any possible modifications to the agreed value.

Additionally, as in every consensus algorithm [17], Raft operates according to some assumptions, including:

1. Consistency can't be ensured from node timing. There is no threshold on message latency between the nodes, they can function at any pace, and their local clocks can sometimes malfunction.

2. The underlying communication network of the distributed system is unreliable, as it includes packet loss, receiving messages out of order, network latencies, and network partitions. Also there can be no Byzantine faults.

3. As long as most of the nodes are running and have a healthy network interaction between them and with the clients, the consensus mechanism is fully operational and available.

4.4 The Raft Protocol

Replicated logs of the kind described in the previous section can be managed through the Raft protocol. In order to reach consensus, Raft first elects a leader among the active servers, after which the leader is given total obligation to manage the replicated log across the distributed system. The elected leader is listening for incoming client requests, forwards them on the rest of the servers in the form of log entries, and notifies them once it is considered safe from the protocol to store the entries to the underlying log. The overall management of the replicated log is made simpler by the leader approach of the mechanism as well as the streamlined data flow from leader to followers.

Raft breaks down three distinct subproblems while examining the consensus problem. The subsequent sections will thoroughly focus on each one of these subproblems as well as

on the algorithm's key properties shown in Figure 4.3, which according to the designers of Raft they ensure distributed consensus:

- **Leader election:** When the current leader fails or malfunctions , a new one should be selected
- **Log replication:** The obligation of a cluster leader is to receive client log entries and to enforce the logs of the other servers to concur with its own after the replication step.
- **Safety:** As stated in the last property on Figure 4.3 , log entries are applied to the state machines with the same ordering and indexing across all servers on the system. This rule ensures data consistency.

Election Safety: at most one leader can be elected in a given term
Leader Append-Only: a leader never overwrites or deletes entries in its log; it only appends new entries.
Log Matching: if two logs contain an entry with the same index and term, then the logs are identical in all entries up through the given index.
Leader Completeness: if a log entry is committed in a given term, then that entry will be present in the logs of the leaders for all higher-numbered terms.
State Machine Safety: if a node has applied a log entry at a given index to its state machine, no other server will ever apply a different log entry for the same index.

Figure 4.3: Properties continuously preserved by Raft to ensure consensus

Before analyzing the each subproblem individually, it is important to introduce some components provided by the algorithm which play a significant role in the system in terms of timing and availability:

- **Server States:** The servers on the system are always operating under one of the following modes or states [1]:

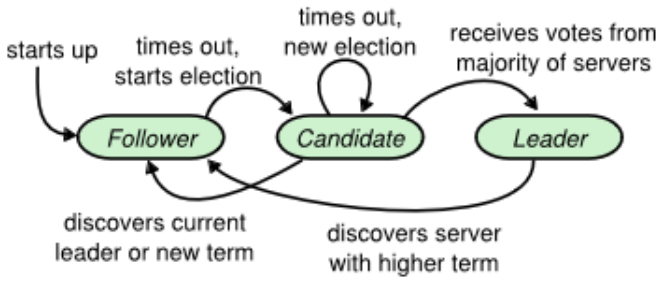


Figure 4.4: Raft server states and the transitions between them [1]

1. Leader: The leader's main responsibility is to handle all incoming requests made by clients. When the cluster operates normally, it is comprised of one leader and the remaining nodes are followers.
2. Follower: Because follower nodes neither initiate nor handle requests by themselves and only reply to those made by candidates and leaders of the cluster, they are considered passive. If any client request is received by these nodes, it is always redirected to the leader node.
3. Candidate: When an elections begins, a server could change to the canditate state. Candidates are voted from the servers in order one of them to become leader. Those who are not voted will fall back to the follower state.

The transitions between these states are depicted in Figure 4.4

- **Terms:** The Raft protocol breaks up operation time into into small terms of arbitrary duration to maintain each server's status, as depicted in Figure 4.5. Terms are recognized through their term number, which increases monotonically. An election is held to pick the next leader among the candidates at the beginning of each term. If a candidate is elected from most of the servers, it obtains the cluster's leadership for the current term. Otherwise, if the majority is not reached, split votes occur and the term stops with no leadership. Thus, leadership is handled by one node on each term.

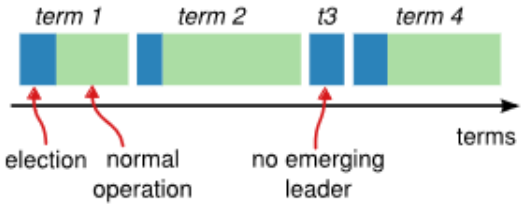


Figure 4.5: Time divided into terms [1]

A current term number is stored on each server and is exchanged whenever communication is initiated between them [1]. The term number on a server is updated to be equal to the received message's term number in case the current term number stored in the server is less than the received one. If a message with higher term number is received by a candidate or the leader, it means that their current term is out of date and they automatically demote themselves back to the follower state.

- **Remote Procedure Calls:** Communication between the Raft servers is carried out by sending Remote Procedure Calls (RPCs) to each other. To be precise, Raft utilizes the following RPC types:

  1. RequestVote RPCs: They are sent by candidates to all the active servers during the election phase.
  2. AppendEntries RPCs: Leader nodes utilize this type to replicate log entries and to periodically sent heartbeats to all the servers across the cluster. Heartbeats are messages with no payload and are sent to notify that the leader is active.

  In case servers don't get a response promptly, they resend RPCs and in terms of performance, they send out multiple RPCs at once.

4.5 Leader Election

Servers always start out as followers when they're activated and stay in that state as long as the leader keeps sending them heartbeats periodically or a candidate sends them RequestVote RPCs. A follower will proceed to the candidate state and trigger a leader election if it doesn't receive any message from the leader of a candidate within a set time interval, called as *election timeout*.

The node will increase its term, cast a vote for itself, and send concurrent RequestVote RPCs to every node in the cluster upon becoming a candidate. The initiated election could result in one of the following results, according to [18]:

1. When the candidate node receives the majority of votes granted by each server on the cluster, it is declared the leader for the particular term and proceeds to the leader State.

2. The term ends without a leader as the Candidate node failed to gather the majority of votes while the election was held, leading to a split vote. In this case elections are restarted after the candidate transitions back to the follower state.

3. The RPC gets rejected and the rest of the Candidates continue to preserve their state if the candidate node sending vote requests has the lowest current term number among the Candidate nodes. On the contrary, the candidate node will be elected as the new leader if its current term number has the greatest value.

Split votes could emerge endlessly in the event of the second outcome described above. In order to cease this behaviour, Raft applies election timeouts that are selected in random from a predetermined range between 150 and 300 milliseconds [1]. This increases the likelihood that a candidate will win the election and start sending heartbeats before another server timeout occurs, and it ensures that in most cases only one server will reach its timeout. Hence, split votes are easily addressed and their occurrance is significantly reduced, because after the split vote every candidate resets the election timeout randomly when a new election takes place.

4.6 Log Replication

Once a candidate node has obtained leadership through an election, it can respond to incoming client requests and extract the included command that the replicated state machines need to process. The received command is stored by the leader to the underlying log as a new entry, and the rest of the servers subsequently receives the replicated entry through concurrent AppendEntries RPC requests from the leader. Three pieces of information usually comprise a log entry [18]:

- The client-specified *command* assigned for execution by the state machines.
- An *Index* which is used to locate an entry's position in a server's log sequentially.
- A *Term Number*, which is used to trace the term in which the leader received the command. It is also used to recover from inconsistencies between the logs as well as to ensure part of the property table shown in Figure 4.3.
- The term number and index of the commited entry stored inside the leader's log which comes right before the new entries.

The entry is applied to the state machine of the leader and responds to the corresponding client with the returned status of the process once most of the server's logs have successfully replicated the entry. Once the procedure is completed, the entry is considered *committed* and the followers are informed from upcoming AppendEntries. In the event of node failure, slow operation, or packet loss due to network malfunctions, entry replicas may not be reached by followers and thus the leader resents these RPC messages until all entries are finally stored in each follower. A possible organized set of logs is depicted in Figure 4.6.

In the depicted state of the logs, the leader has committed the 7th log entry and all the preceding ones as they have been replicated on most of the server logs. The logs of second and forth row belong to servers which might have faced failure or packet loss, resulting to inconsistency with the leader log. The leader is responsible to modify these logs according to its own log.

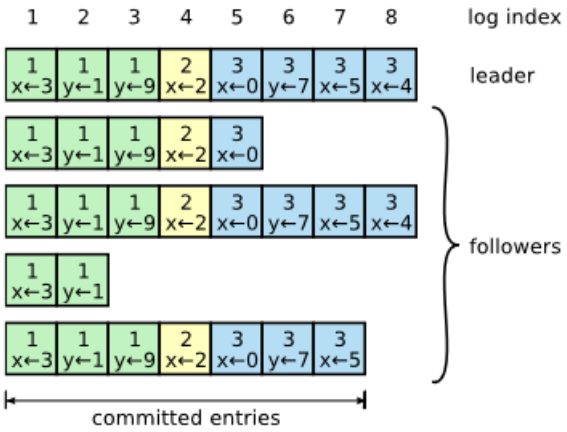


Figure 4.6: Replicated logs for a cluster of 5 servers [1]

In order to maintain the same state across all replicated state machines, server logs must contain identical entries with the same order. Raft provides the demanded consistency among these logs through the *Log Matching Property* [1] which is included in Figure 4.3. It specifies that if an entry in two logs is pointed by the same index and contains the same term number, then among these logs the contained entries and all the preceding ones are identical. In practice, this features is guaranteed by a series of consistency checks implemented inside the AppendEntries and take action when these RPC messages are received [1]. The consistency check includes the following actions:

1. The follower rejects new entries included in AppendEntries if it cannot locate an entry in its log with the same term number and indexing of the message.

2. The follower rejects the received AppendEntry if the included term number is less than the server's current term. This indicates that the sender is likely a stale leader.

3. If an entry in the follower's log and a newly received entry have equal indexes but contain different terms, the conflicting entry within the follower's log is removed along with all entries that follow it.

Consistency checks are held whenever server logs intent to append new entries and if the check fails, the insertion will be aborted. Hence, the leader is always aware that the log of the follower node matches up to his own except of the new entries whenever it receives successful responds from the forwarded AppendEntries.

If the system operates normally, leader and followers logs are kept in sync with identical entries in every term. However, these logs can sometimes appear inconsistent since following a leader crash, it may not have completed the replication of its log entries to every follower in the cluster or because of follower node failures and restarts. Figure 4.7 depicts various inconsistency scenarios between follower and leader logs.

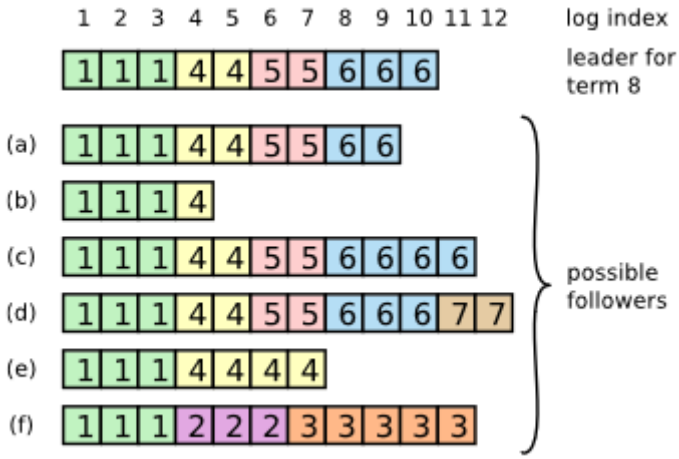


Figure 4.7: Possible follower log scenarios [1]

A follower may not contain all the entries that exist on the leader's log, or it may have stored additional uncommitted entries from an older leader and thus they are not included in the new leader's log, or both scenarios can occur.

The leader is assigned to solve these incosistencies by overwriting entries from it's own log over conflicting entries in the follower logs. To achieve this, it must locate the latest entry which is included in both logs [1]. The variable *nextIndex*, which is maintained by the leader, is an index that indicates the next log entry that the leader will forward across every follower in the cluster. This index always points to the position that follows the last entry in the leader's log during normal operation or on initial leader state. If the logs of the follower and the leader contradict, the consistency check fails and the AppendEntries RPC is rejected by the follower. The leader then decreases the nextIndex and tries again to forward the RPC for the preceding entry. This index will eventually point to a position were the follower and leader logs match until that entry. In that case, the check mechanism will have removed all conflicting entries

from the followers log, and the leader will be able to append it's own entries to synchronise and ensure consistency among the logs. Hence, a node entering the leader state does not need to implement any particular functions to make logs consistent using this approach. It begins to operate, and the logs converge automatically.

4.7 Safety

The approaches discussed until now are insufficient to guarantee that each state machine across the cluster performs identical commands in the same order. The Raft algorithm addresses this issue by limiting which servers can be elected as leaders. To be precise, it prevents candidates from obtaining leadership unless all committed entries are included in it's log.

In order to be elected, a candidate node has to communicate with most of the cluster members, which implies that at least one of them must contain all entries that have been committed [1]. As a result, the algorithm assures that for any given term, the leader serving at that interval will contain all entries committed in preceding terms. To implement this restriction, the RequestVotes RPC delivers additional information about the number of entries stored in the candidate's log as well as the term of the most recent log entry. Raft compares the term number and the index of the latest entries between the candidate and the vote receiver log to determine which of the two logs is the most updated one. If the candidate's message includes an earlier term than the one of the receiver server, the vote will be rejected. If the terms have identical numbers but the candidate's log contains fewer entries compared to the receiver's log, the vote will be rejected.

4.8 Client Interaction

This section briefly describes the client interaction with Raft, such as the way clients locate the current node serving as leader and how the system handles incoming serializable and linearizable commands. When the clients wants to issue the first request, it connects to a server that is picked up at random. If the picked server is not the leader, the client request will be rejected and the server will reply with an AppendEntries RPC message which will include the current leader's network address [1]. Furthermore, client requests won't be completed if

the leader crashes. In that case, clients randomly select a cluster member and attempt to send the request again. Client requests received by any follower that must be approved through Raft consensus are immediately forwarded to the leader [19]. Such requests include linearizable read and write operations. On the contrary, requests that don't need consensus, such as serializable read operations, can be handled by any server in the cluster with the risk of accessing stale data.

Chapter 5

Experimental Tools

5.1 Introduction

in this section we give a brief description of the software equipment which was utilized for the assessment and experimental implementation of this thesis project. The following tools are introduced:

1. *NITOS*, which is a remotely accessible testbed at the University of Thessaly that supports both wireless and wired experimental research.
2. *Docker*, a software tool that allows application deployment inside lightweight containers.
3. *Prometheus*, an open-source toolkit that monitors health status and performance metrics of components in a distributed system.
4. *Grafana*, a web application that provides dashboards that include several charts and graphs for displaying metrics from various data sources.

Each of these components is thoroughly discussed in the following sections.

5.2 NITOS testbed

The Network Implementation Testbed utilizing Open Source platforms [20], which is abbreviated as NITOS, is managed and created by the University of Thessaly's Network Implementation Testbed Laboratory (NITlab) [21]. The NITOS facility was created in order to

provide feasible solutions for deploying and evaluating wired and wireless network-based research topics. The testbed is widely utilized from members of the network research community across the world to conduct experiments as well as it is available around-the-clock and can be accessed remotely.The open-source software infrastructure of the testbed provides the opportunity to create and deploy novel algorithms, allowing for additional functionalities to be implemented on current hardware. Furthermore, the facility is comprised of several functional wireless nodes and it's primary objective is to guarantee reproductive experimental results as well as to provide a flexible way to assess protocols and applications under real environment conditions. The NITOS testbed is controlled and managed through the cOntrol and Management Framework (OMF) open-source software [21]. Figure 5.1 depicts the NITOS architecture. It consists of three individual deployments that are geographically seperated, **the Indoor RF Isolated Testbed, the Office Testbed, and the Outdoor Testbed** [20].

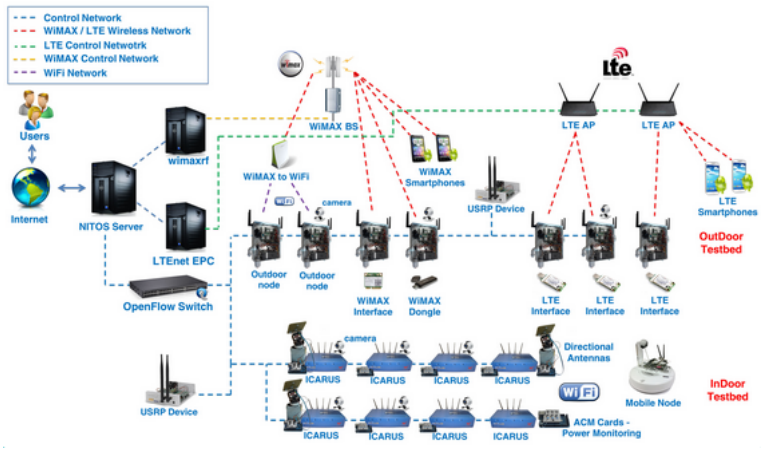


Figure 5.1: NITOS Architecture

5.2.1 Outdoor Testbed

NITOS Outdoor deployment includes 50 robust ICARUS nodes that have several wireless interfaces suitable for conducting experiments with diverse wireless communication standards, including Wi-Fi, WiMAX and LTE [22].The testbed has been set up outdoors in a University of Thessaly's campus building, depicted in Figure 5.2.To be precise, 25 of these nodes are scattered over the floors of the building and 25 are spaced in a grid topology throughout the building's roof.

Figure 5.2: Nitos Outdoor testbed

5.2.2 Indoor RF Isolated Testbed

50 ICARUS nodes comprise the NITOS RF Isolated Indoor testbed which is deployed on the inside of a University of Thessaly's campus building. These machines provide wireless interfaces for Wi-Fi, LTE, and WiMAX and are arranged in a symmetrical way around the isolated environment, establishing a grind topology. Such an extensive testbed provides several opportunities to the experimenters, including execution and evaluation of power-demanding processing algorithms and protocols [23].Moreover, the testbed is outfitted with several cutting-edge technologies such as directional antennas. The nodes additionaly include two ethernet interfaces, *eth0* and *eth1*, which are utilized for distinct purposes.

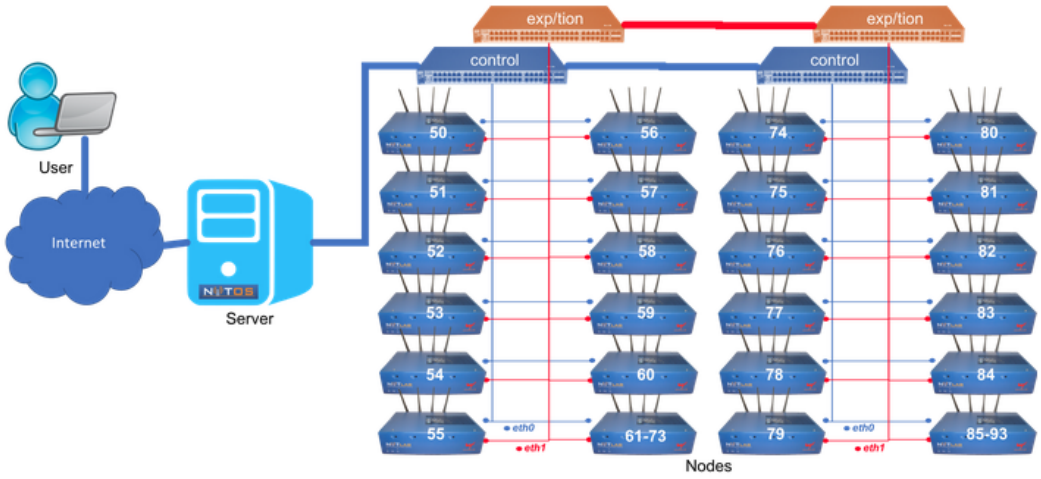


Figure 5.3: Nitos Indoor testbed

The *eth0* interface is used to interact with the nodes and to control them, whereas *eth1* is used for experimentation and is configured according to the experimenter's preferences. This wired topology of the Indoor Testbed is depicted in Figure 5.3. A set of these nodes were utilized to conduct experiments for this thesis project, and their corresponding *eth1* interface was configured to meet the project's needs.

5.2.3 Office Testbed

10 robust second generation ICARUS nodes form the Office Indoor Testbed. These nodes integrate various heterogeneous technologies which include Wi-Fi, 5G, WiMAX and LTE. These technologies enable the experimenter to design, execute, and evaluate real-world scenarios in a deterministic office environment. The Office Testbed is distributed across an entire floor of the NITlab laboratory, as shown in Figure 5.4.

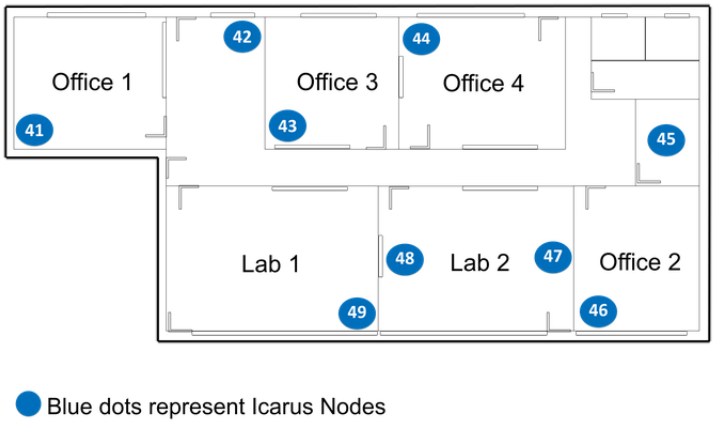


Figure 5.4: Nitos Office testbed

5.3 Docker Engine

As stated in Chapter 2, a container is a software component that provides a way to wrap the containerized application along with all its dependencies so that the application can be executed across different computer environments reliably and quickly. The containerized application's runtime, settings, system libraries and tools, and code are all included in a Docker container image which constitutes an independent, executable software package [24]. When it comes to Docker containers, container images are converted into containers, and Docker provides universal packaging approach and simple tooling that is wrapping up all embedded

application dependencies of a container, which eventually runs on Docker Engine, the industry's most popular container runtime. Figure 5.5 depicts the containerized application layer and it's interaction with the Docker platform.

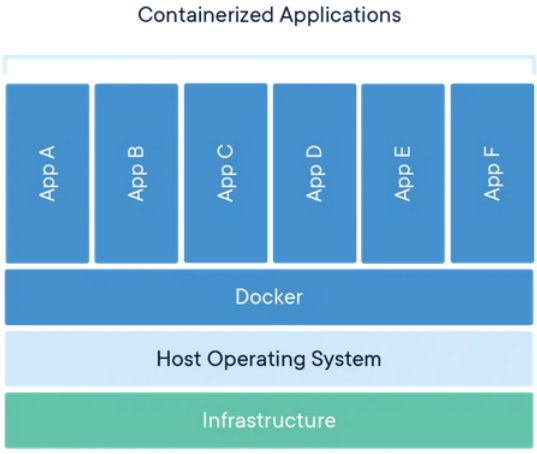


Figure 5.5: Docker Containers

5.4 Prometheus

Prometheus is a software toolkit that provides monitoring and alerting features and is designed to observe health status and performance metrics of components in a distributed system and cloud-native environments. To be percise, it keeps track of metrics and saves them as time series data, implying that the information included in the metric is stored alongside the timestamp that indicates when the metric was recorded. It can also record and collect optional key-value pairs called labels [25].

Prometheus expects an accessible HTTP endpoint as it's target in order to collect various metrics and can start to scrape numerical data from the exposed endpoint once it is available. If these metrics can't be accessed with their current format directly, Prometheus is using third-party exporters which expose the endpoint metrics in the desired format in order to scrape them. After these metrics are recorded in time series format, they are archived in a local time series database, which allows for efficient retrieval for monitoring data behaviour throughout time. In terms of gaining insights into the system's behaviour, Prometheus provides users with the Prometheus Query Language (PromQL) which enables time series data selection and aggregation in real-time. Queries can also be utilized to define alerting events that could arise from a specific condition or a threshold. When the event is triggered,

Prometheus notifies the user through external systems including email, Slack, or PagerDuty. Additionally, Prometheus' web-based user interface on port 9090 can present collected data in tabular or graph form. Grafana and other third-party visualization tools can also be integrated with Prometheus by using the related API [26]. Figure 5.6 demonstrates some of these workflow components and the Prometheus architecture.

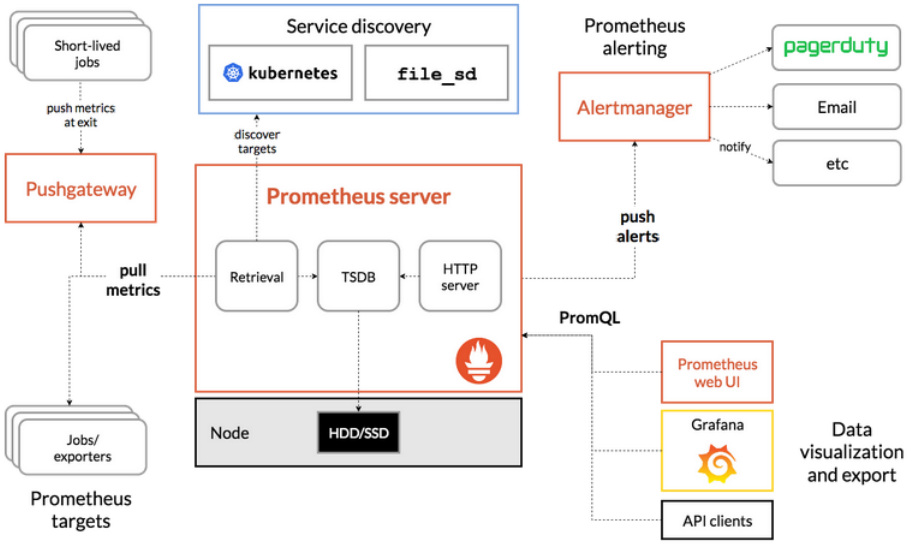


Figure 5.6: Prometheus echosystem components

5.5 Grafana

Grafana is an open-source software (OSS) [26] data visualisation software platform while it also provides analytics for a variety of metrics through interactive and configurable dashboards that can be easily deployed. Users can utilise the platform's tools to transform time series data to informative graphs and charts. Dashboards are components that include these graphs to keep track of the experimental system by pulling data from Grafana's plugin framework which enables data sources like PostgreSQL, Prometheus and Graphite to connect and forward their stored data.

Aside from graphs and charts, a dashboard can include multiple seperate panels on the layout of the grid. Each one of these includes it own set of functions and a variety of visualization alternatives, such as heat maps, geo maps and histograms. For our thesis project, we deployed a handful of dashboards on our Grafana instance from the official website that were suitable for our experimental metrics. One of these dashboard's sample is seen in Figure 5.7.

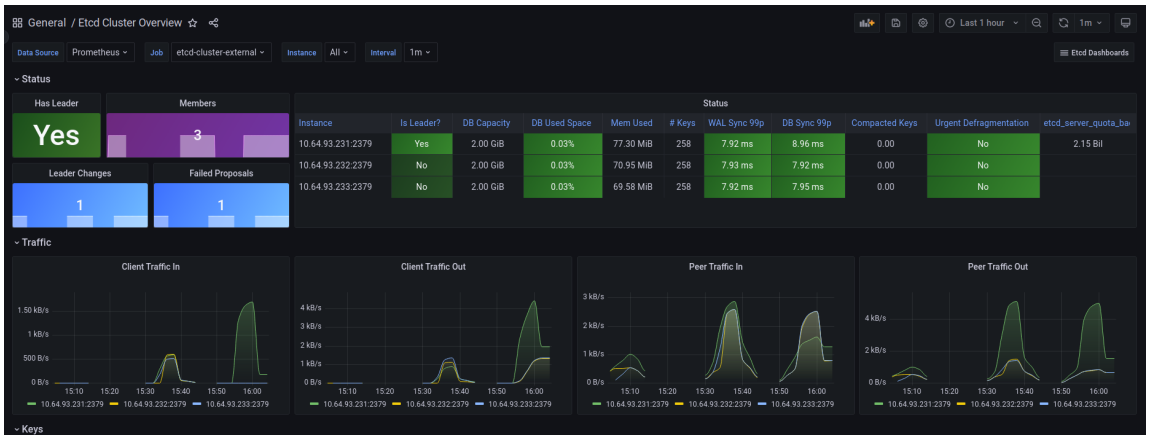


Figure 5.7: Grafana dashboard overview

5.5.1 Prometheus as data source

Grafana features native Prometheus integration as it's data source, allowing it to visualize system metrics scraped by Prometheus in the form of queries through a variety of dashboards. Grafana offers a user-friendly and flexible web interface for building customized dashboards based on the experimenter's monitoring preferences, whereas the Prometheus model exports optimal time-series data for health check and performance analysis via various modifications supported by queries. Thus, their combination generates a powerful stack for effective observation and complete metric analysis of the examined system. In this thesis project, a Prometheus instance is deployed locally on one of the NITOS nodes and the scraped metrics are pushed to the Grafana server instance installed at the same node by altering specific configuration files.

Chapter 6

Implementation and Analysis of Optimized etcd Load Balancing Policies

6.1 Objective

As discussed in Chapter 3, etcd has become the primary data storage method for Kubernetes cluster deployments because it provides a reliable approach to guarantee high consistency over stored metadata that hold information about the state of the cluster. The servers of the etcd cluster can either be configured as Pods in the Kubernetes master node or as parts of a standalone cluster in a separate architecture. The last deployment approach has several benefits, including independent management of the external etcd cluster with less complexity overhead, enhanced security, and more efficient utilization of the isolated system resources. However, these properties come with tradeoffs that increase the latency overhead on the communication network between the etcd members and the Control Plane components. TLS encryption, distance between endpoints, and transmission latency caused by high load on etcd servers are some of the factors that can impact the performance of this communication link. This thesis project attempts to suppress these factors by extending the default load balancing policy used by the etcd client API, Round Robin, to support two types of policies which the kubernetes API server will use to forward requests to the etcd servers:

1. Pick Leader: Forwards all client requests to the leader node of the etcd cluster

2. Pick Leader with Leader Status Forecasting: It maintains the Pick Leader policy along with machine learning support through a classification model that predicts the current

status of the etcd leader and depending on the prediction it decides whether to trigger an election or not.

The implementation of these policies is presented in following sections.

6.2 Experimental System Setup

A set of nodes located in the Indoor RF Isolated testbed had been utilized to deploy a proposed architecture that includes an external etcd cluster and a kubernetes cluster. Figure 6.1 depicts the experimental topology, which includes the following objects:

- etcd node
- Kubernetes master node
- Prometheus & Grafana node

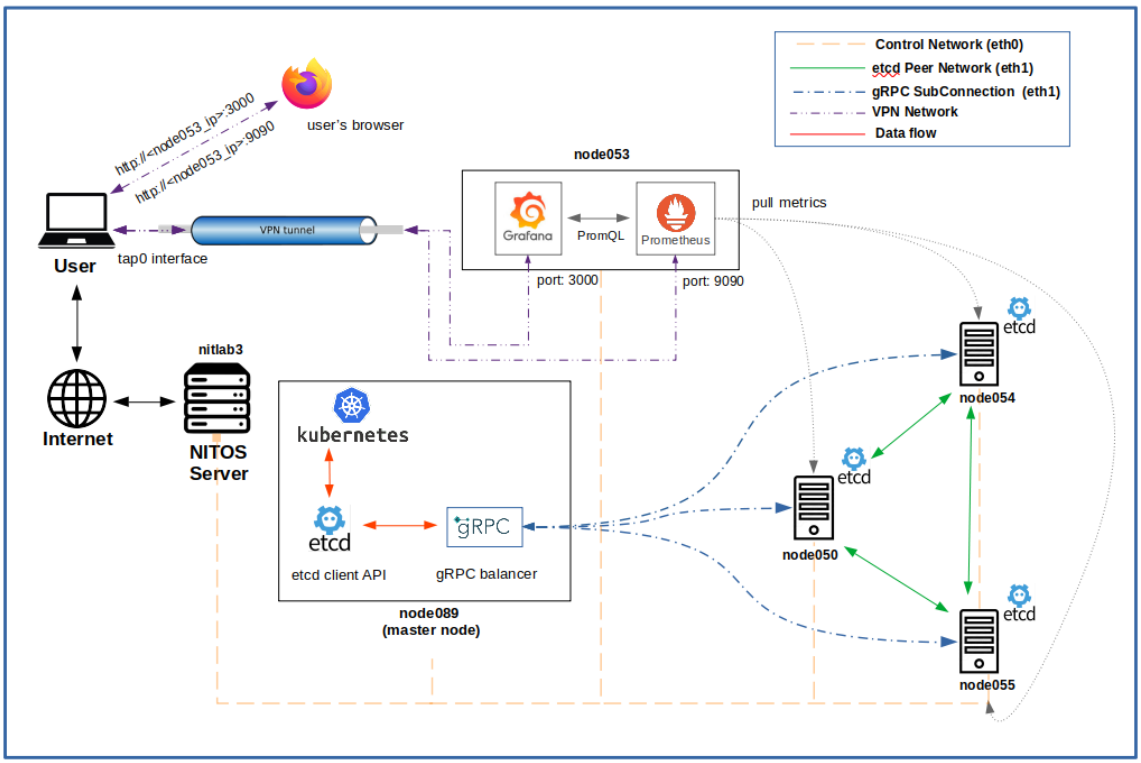


Figure 6.1: thesis topology deployed on NITOS

As outlined in the previous Chapter, each node of the Indoor Testbed contains two ethernet interfaces: one for controlling the node remotely and executing commands via *ssh*, and

another used for experimentation. In order to build the network between all the nodes within the architecture, the second ethernet interface (*eth1*) was configured to create links among the etcd cluster nodes to forward peer traffic, between the kubernetes master node and each etcd server to transmit client requests, and between the Prometheus a Grafana node and each etcd server to scrape metrics. The underlying operating system which the loaded disk image runs on each node is *Ubuntu 20.04.3 LTS (Focal Fossa)*.

6.2.1 etcd node

To set up the etcd cluster, version *v3.5.1* of the available etcd binaries was installed on each node. Afterwards a *systemd unit* file was created on all the three machines to start the etcd service and it's configuration was based on each node's IP address and the IP addresses of the fellow cluster members. The configuration file exploits the eth1 interface to advertise *port 2080* on which peer traffic among the etcd members is exchanged to keep them in sync, and *port 2379* which listens for etcd client requests. Once the etcd service starts running on each node, the etcd cluster will start operating. For simplicity sake, TLS authentication and encryption was not enabled.

6.2.2 Kubernetes master node

To build and configure the kubernetes cluster according to our preferences, the kubernetes Github repository was installed and build from source in order to modify the source code to extend the already existing etcd load balancing policy. At first, Docker Engine *24.0.5*, golang *1.19.5* and etcd *v3.5.1* where installed which are vital for the normal operation of the cluster. Afterwards, the kubernetes source code was installed by cloning the kubernetes *v1.19.16* git repository. At this point we must modify the default configurations of the cluster in order to use the external etcd as it's primary data storage. To accomplish this we modified two configuration scripts, *etcd.sh* and *local-up-cluster.sh* located at the *hack/lib* and *hack/* file path of the repository respectively. The first script is responsible for executing and terminating the etcd binary whereas the second script includes the configuration of every kubernetes Control Plane and Node component to build and run the local kubernetes cluster. In the first script, we applied the binary path of the installed etcd version in order to be compatible with the verison of the external etcd, while on the *local-up-cluster.sh* we specified the list of the external etcd endpoints by modifying the *–etcd-servers* field of the *kube-apiserver* configuration file.

Additionally the eth1 interface was exploited to advertise the address of the API server for the kubernetes traffic. After modifying the configurations, the following command is executed to run the local kubernetes cluster along with the external etcd:

```
$ ${GOPATH_K8S}/hack/local-up-cluster.sh
```

6.2.3 Prometheus & Grafana node

This node hosts the Prometheus and Grafana software components, which where described in the previous Chapter, and is responsible for scraping periodically the etcd metrics which are available on each etcd server at port *2379*. Prometheus version *2.32.1* was installed and it's configuration file *prometheus.yml* was modified in order to target the endpoints of the external etcd cluster. To start scraping metrics, a *systemd unit* file was configured to enable and start the prometheus service. To deploy Grafana we followed the same procedure, which has built-in support to detect prometheus and assign it as it's data source. To use the Grafana's web interface to display the scraped metrics, a VPN link between the node and the experimenter's laptop was created in order to access the Grafana and Prometheus URL remotely using the user's browser.

6.3 Implementation Analysis

By following the aforementioned configurations for the etcd nodes and the kubernetes master node, the kubernetes cluster is successfully launched by storing all of the necessary key value pairs to the external etcd cluster. Their interaction is verified using the *tcpdump* packet analysing tool over the eth1 interface to capture TCP packets that encapsulate the RPC messages that are exchanged between the etcd client and the external servers. We were also able to display on each etcd server the key value pairs which the kubernetes cluster had created via the *etcdctl* tool which is used to interact with the servers through shell commands. The next step was to examine the *kube-apiserver* log file to identify the file path of the source code in the kubernetes repository that implements the Round Robin load balancing policy. After some searching across the repository folders we discovered that both etcd client API and the grpc balancer implementation, which are introduced in Chapter 3, are located at the *vendor/* directory, which contains third-party libraries used by the kubernetes API to interact with external applications. To be precise, *clientv3*, which

is the official client implementation for etcd written in Go, is passing all the etcd requests made by the kubernetes API server through *roundrobin_balanced.go*, located at the *vendor/go.etcd.io/etcd/clientv3/balancer/picker* folder. This is where the Round Robin balancer implementation takes place with the following source code:

```
1  package picker
2
3  import (
4    "fmt"
5    "sync"
6    "go.uber.org/zap/zapcore"
7    "go.uber.org/zap"
8    "google.golang.org/grpc/resolver"
9    "google.golang.org/grpc/balancer"
10   "context"
11 )
12
13 // This function returns the address of a new roundrobin balanced picker.
14 func newRoundrobinBalanced(cnfg Config) Picker {
15   subconns := make([]balancer.SubConn, 0, len(cnfg.
       SubConnToResolverAddress))
16   for sbconn := range cnfg.SubConnToResolverAddress {
17     subconns = append(subconns, sbconn)
18   }
19   return &rrb{
20     plc:         RoundrobinBalanced,
21     logger:        cnfg.Logger,
22     subconns:       subconns,
23     sbconnToAddr: cnfg.SubConnToResolverAddress,
24   }
25 }
26
27 type rrb struct {
28   plc Policy
29
30   logger *zap.Logger
31
32   mtx        sync.RWMutex
33   nxt      int
```

```
34   subconns       []balancer.SubConn
35   sbconnToAddr map[balancer.SubConn]resolver.Address
36 }
37
38 func (rb *rrb) String() string { return rb.plc.String() }
39
40 // Pick is called for every client request.
41 func (rb *rrb) Pick(ctx context.Context, opts balancer.PickInfo) (
       balancer.SubConn, func(balancer.DoneInfo), error) {
42   rb.mtx.RLock()
43   size := len(rb.subconns)
44   rb.mtx.RUnlock()
45   if size == 0 {
46     return nil, nil, balancer.ErrNoSubConnAvailable
47   }
48
49   rb.mtx.Lock()
50   cur_idx := rb.nxt
51   sbconn := rb.subconns[cur_idx]
52   picked_addr := rb.sbconnToAddr[sbconn].Addr
53   rb.nxt = (rb.nxt + 1) % len(rb.subconns)
54   rb.mtx.Unlock()
55
56   rb.logger.Debug(
57     "picked",
58     zap.String("picker", rb.plc.String()),
59     zap.String("address", picked_addr),
60     zap.Int("subconn-index", cur_idx),
61     zap.Int("subconn-size", size),
62   )
63
64     // is called when the RPC is completed
65   doneFunc := func(info balancer.DoneInfo) {
66     fss := []zapcore.Field{
67       zap.Error(info.Err),
68       zap.String("picker", rb.plc.String()),
69       zap.String("address", picked_addr),
70       zap.Bool("success", info.Err == nil),
71       zap.Bool("bytes-sent", info.BytesSent),
```

```
72          zap.Bool("bytes-received", info.BytesReceived),
73      }
74      if info.Err == nil {
75          rb.logger.Debug("balancer done", fss...)
76      } else {
77          rb.logger.Warn("balancer failed", fss...)
78      }
79  }
80  return sbconn, doneFunc, nil
81 }
```

Listing 6.1: roundrobin_balanced.go : Implemented RoundRobin policy

In order to create the Pick Leader policy, all incoming client requests should be redirected to the subconnection which is pinned to the endpoint of the current etcd leader. This was managed by initializing a goroutine (thread in Go language) in the function at line 14 of the preceding code, which creates a new Round Robin balancer object. The goroutine executes every 5 seconds the *etcdctl* shell command that is presented below, which extracts information about the status of each state machine at the etcd cluster, including member ID, term number, current database size and node state (leader or follower). By performing string manipulation on the output we can detect which endpoint serves as the current etcd leader, and the corresponding URL is assigned to the *isLeader* value of the Balancer object. When the Pick function is called for an etcd client request, it examines if the current leader has an available subconnection. If the subconnection doesn't exist in the specified list, the balancer keeps the subconnection that was already chosen by the Round Robin policy (lines 49 - 54 of the preceding code). The goroutine code is presented in the first underlying code instance whereas the selection of the etcd leader's subconnection is shown in the second one.

```
1
2 func newRoundrobinBalanced(cnfg Config) Picker {
3
4     subconns := make([]balancer.SubConn, 0, len(cnfg.
    SubConnToResolverAddress))
5
6     for sbconn := range cnfg.SubConnToResolverAddress {
7     subconns = append(subconns, sbconn)
8   }
9
```

```
r := &rrb{
  plc:          RoundrobinBalanced,
  logger:         cnfg.Logger,
  subconns:        subconns,
  sbconnToAddr: cnfg.SubConnToResolverAddress,
  isLeader: "",
}

//----------------NEW_CODE-------------------------

// Spawn a new goroutine to observe the etcd leader
  go func() {
      for {

    output, err := exec.Command("sh", "-c","ETCDCTL_API=3 etcdctl --endpoints=http://10.64.93.241:2379,
                    http://10.64.93.242:2379,http://10.64.93.243:2379 endpoint status").Output()

    //If an error occurs, log the error
    if err != nil {
      panic(err)
    }

    // Convert the byte slice to a string and split it based on whitespace characters
    lines := strings.Fields(string(output))

    ip := lines[0]

    for _,v := range subconns {

      // Find the IP address of the leader
      for _, line := range lines {

        if strings.Contains(line, "http") {
          ip = strings.TrimSuffix(line,",")
        }

```

```
            if strings.Contains(line, "true,")
                    && ip == cfg.SubConnToResolverAddress[v].Addr{
              r.isLeader = cfg.SubConnToResolverAddress[v].Addr
            }
          }
        }
            // Sleep for a fixed interval before checking again
            time.Sleep(5 * time.Second)
          }
      }()
    //------------------------------------------------
    return r
}
```

Listing 6.2: roundrobin_balanced.go : Pick Leader policy

```
  for _,v := range rb.subconns {
    if rb.sbconnToAddr[v].Addr == rb.isLeader {
      sbconn = v
    }
  }
```

Listing 6.3: roundrobin_balanced.go : picking etcd leader policy

6.4 Evaluation and Experimental Results

6.4.1 Round Robin and Pick Leader

After modifying the source code of the default load balancing policy, we recompiled the kubernetes repository so that the launched kubernetes cluster will make use of the extended load balancer. To visualize the difference between the two policies, we ran two separate experiments, one with the default balance policy and the other with the extended policy. To conduct the experiments under similar environment conditions, we reloaded the dis image of the kubernetes master node and erased all key value pairs from the etcd database before running the second experiment. During that interval, we had set up the Prometheus and Grafana services to collect metrics for both experiments. Figure 6.2 is a graph from Grafana's etcd dashboard which displays the volume of RPC requests issued from the *kube-apiserver* to each

etcd server through the gRPC interface. When using Round Robin, we can clearly see that the workload is evenly spread the three etcd servers, however in the Pick Leader experiment, all the previous workload is managed by the etcd leader.

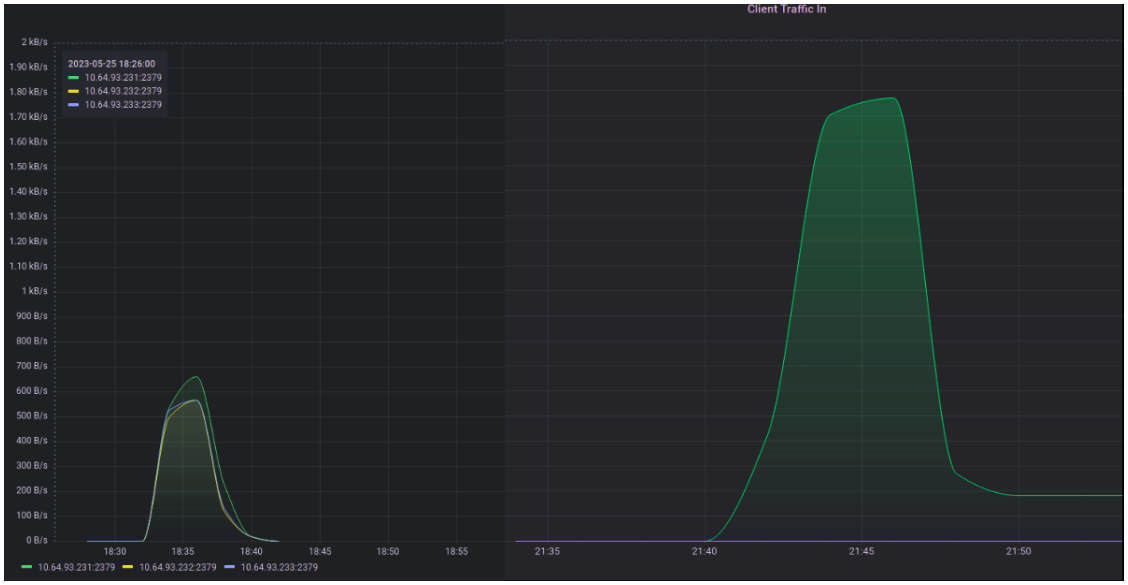


Figure 6.2: etcd cluster incoming client traffic using Round Robin and Pick Leader load balancing policies.

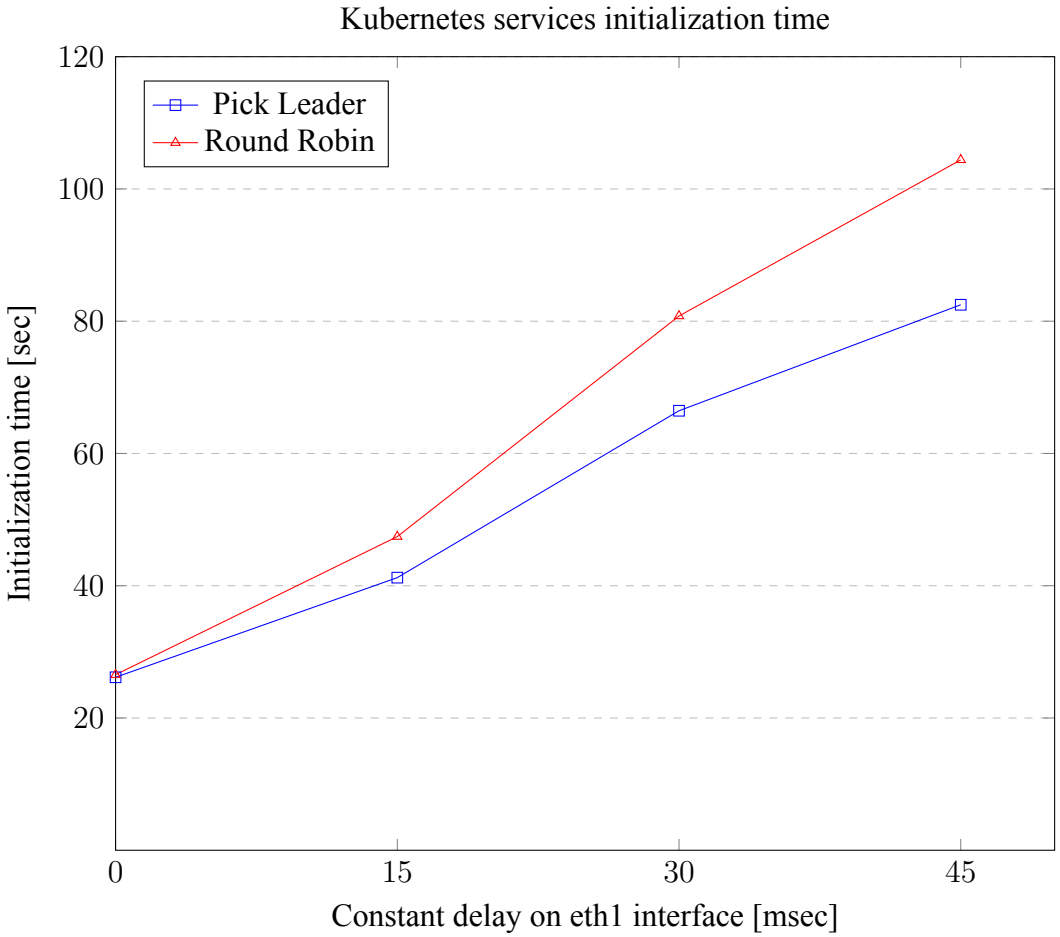
To compare the performance of the Pick Leader policy to Round Robin, we modified the kubernetes repository script that initializes and runs both Control Plane and node components by inserting time variables that compute the total initialization time of the kubernetes cluster. Furthermore, because the etcd servers of an external etcd cluster may not be placed at a research facility in real-world deployment scenarios, we conducted the experiments using various delays on the eth1 interface of each etcd server via the network traffic configuration tool *tc* [27]. Table 6.1 depicts the elapsed initialization time of the kubernetes cluster using both Round Robin and Pick Leader policy with 0, 15, 30, and 45 millisecond added delay on each etcd node's eth1 interface respectively. We can observe from the findings and from

the following graphs that when the traffic delay increases, the difference between the initialization time of the two experiments grows as well. This occurs because, when using Pick Leader balance policy client requests that would have been pushed to the followers if we used Round Robin are now forwarded directly to the etcd leader, allowing us to save one RTT for each client request that whould have been issued to a follower (redirect client request

constant delay on eth1	Round Robin	Pick Leader	Speedup
no delay	26.548 sec	26.157 sec	1.015
15 msec	47.430 sec	41.223 sec	1.15
30 msec	80.785 sec	66.459 sec	1.216
45 msec	104.395 sec	82.486 sec	1.267

Table 6.1: Initialization time for Kubernetes services on different delay scenarios.

to the leader and wait for the response from the leader when the command is committed to send a reply to the client).



6.4.2 Pick Leader with Leader Status Forecasting

The aim of this implementation is to predict the state of the etcd leader using relative etcd metrics while the Pick Leader balancer is running. If the leader status is considered unstable, the algorithm demotes the node to the follower state and elects a new leader. In order to create

a dataset to train the Machine Learning model, the following unstable states were created:

1. Overload: Using the benchmark tool provided by etcd, we generate 1000 etcd clients that send 100 linearizable read requests to the etcd leader for a 700 kB key value stored in the database.

2. Network delay: We utilize the aforementioned *tc* network tool to add large delay values on the eth1 interface and thus to generate a high latency response for incoming client requests.

3. Overload with Network delay: The previous conditions are met at the same time.

When none of these states is triggered, the leader state is considered Idle which means that the etcd cluster operates normally. The creation of the dataset was fully automated using two scripts.The first script selects one of the states at random every 5 minutes and applies it to the experimental environment, while the second extracts the following etcd metrics from the leader node every 5 seconds:

1. *etcd_disk_wal_write_bytes_total*: Total number of bytes written in the node's disk using fsync.

2. *etcd_network_client_grpc_sent_bytes_total*: Volume of RPC messages received by clients in bytes.

3. *etcd_network_peer_sent_bytes_total*: Number of transmitted bytes to other etcd members.

4. *etcd_server_heartbeat_send_failures_total*: Detected node failures to forward heartbeat messages, which are utilized to inform the followers that the node still owns the leadership. This metric indicates that the node is likely overloaded.

5. *etcd_server_slow_apply_total*: This metric tracks the number of consensus proposals made by the etcd node with large commit time.

Since all these metrics are counter variables, we converted them to rates before appending them to the dataset using the following formula:

$$\frac{current\_metric - previous\_metric}{scrape\_interval}$$

where previous and current metric are the values of the same metric that had been scraped before and after the scrape interval respectively. Moreover , since the implementation is based on states, we focused on using the generated dataset to train multi-class classification machine learning models.

Evaluation Metrics

This section presents the metrics utilized to evaluate the performance of the examined machine learning models for our multi-class classification problem as well as to compare them. The evaluation metrics are the following:

1. Precision: For a given class, *Precision* detects the amount of all the instances predicted as this class that actually belong to that class.

$$Precision = \frac{TruePositives}{FlasePositives + TruePositives}$$

   - False Positives (FP) are the instances that actually belong to another class but are predicted as this one.
   - True Positives (TP) are the instances that are accurately predicted as that class.

2. Recall: For a given class, *Recall* is the percentage of all instances that actually belong to that class that are accurately predicted by the classification model.

$$Recall = \frac{TruePositives}{TruePositives + FlaseNegatives}$$

   - False Negatives (FN) are the instances of a class that are incorrectly predicted as another class.

3. F1-score: For a certain class, the *F1-score* is the weighted harmonic average which of the previously mentioned measures. It shows us what percent of predictions for that class were actually correct.

$$F1 - score = \frac{2 * (Precision * Recall)}{Precision + Recall}$$

F1-score metric is also highly useful when dealing with imbalanced datasets where some classes exceed others in terms of samples per class. In such cases, the accuracy metric may be unreliable.

Classification models evaluation

With the generated dataset we trained offline and evaluated the following classification models:

- XGBoost: In XGBoost (eXtreme Gradient Boosting) the implemented decision trees can have an arbitary amount of terminal nodes, which basically form the final decision points of the tree. The algorithm also assigns weights on the leaves according to the calculated evidence and information of that node. Also, weight shrinkage is minimizing the influence of individual trees with high weights to the final prediction. XGBoost additionally utilizes the Newton-Raphson method assigning optimal weights during the tree-building process and compared to the traditional gradient descent technique, it provides a more direct route to the minimum.

- Artificial Neural Network (ANN): An ANN is made up of units called artificial neurons. Neurons receive signals from other neurons as an input, and through a non-linear function that utilizes the specified input, it forwards the computed output signal via edges with set weights which are utilized to optimize the learning process. Typically, these units are aggregated into layers to form an Artificial Neural Network. We use the Tensorflow python library [28] to develop an ANN classifier with three hidden layers having 128, 64, and 32 neurons, respectively. Considering that we have multiple classes, the output layer is made up of neurons proportional to the number of these classes. Each neuron of the output layer will generate a probability for the relative class, and if that class has the highest probability it will be the final prediction.

Table 6.2 shows the F1-score calculated for each class prediction made by the aforementioned models. In order to make a reliable estimation of each model's performance, the presented metrics are the average of the f1-scores derived from 10 different predictions using the same dataset. Based on the evaluation results, we settled on the XGBoost classifier to be trained offline and integrated into our experimental topology to perform status forecasting for the etcd leader.

Class	ANN (F1-score)	XGBoost (F1-score)
0 (Idle state)	81.94%	93.63%
1 (Network delay)	94.10%	97.35%
2 (Overload)	83.84%	85.89%
3 (Network delay & Overload)	89.12%	89.87%

Table 6.2: Evaluation results after 10 different predictions of each ML model.

In this scenario, the pre-trained model is used by a python script implemented within the kubernetes master node which scrapes the etcd leader's metrics every 5 seconds and extracts those that will be used as input for the model to predict the leader's state. In addition, a UDP socket is established between the Python script and the etcd balancer to inform the balancer about the script's actions depending on the prediction output. To be more specific, if the predicted instance belongs to class 0 or 2, the leader is considered stable and the script forwards the current leader's endpoint. Otherwise, if the instance belongs to class 1 or 3, an election is triggered and the new leader's endpoint is forwarded to the balancer through the UDP socket. While the kubernetes cluster operates along with the Pick Leader balancer to distribute traffic to the leader of the external etcd cluster, the balancer listens to the UDP socket for the endpoint to which it must connect. No changes are made until the receiving endpoint differs from the one currently linked to the balancer.

Chapter 7

Conclusions

7.1 Summary and Conclusions

In this thesis, we deployed a system architecture consisting of a kubernetes cluster and a separate external etcd cluster, and we designed an extended load balancing policy to resolve latency issues on their communication in real world environments. To be precise, we altered the default load balancing policy used by the kubernetes API server, which is Round Robin, to route all etcd requests to the etcd leader exclusively. To build our implementation we first conducted a comprehensive investigation about the software components and attributes of the Kubernetes ecosystem, the etcd data storage and the Raft protocol. Following that, we created a network that connects the two distinct clusters hosted on nodes of the NITOS Indoor Testbed in order to conduct experiments to compare the performance of our extended policy to the default one. We also integrated a classification model into our policy to provide improved system reliability by predicting the status of the etcd leader node, such as high load. We evaluated the performance of two classifiers, XGBoost and ANN, using a dataset that was created by storing metrics from the etcd leader during several experiments.

7.2 Future Work

For production Kubernetes cluster deployments with external etcd topology, our extended load balancing policy combined with the integrated machine learning algorithm has the potential to deliver an optimized and more resilient solution compared to the default communication mechanism. Certainly, our implementation can benefit from additional modifications

and improvements. To enable our implementation to adapt to network conditions in various environments, the effects of other network technologies and configurations suitable for communication between the Kubernetes cluster and etcd can be studied. Furthermore, except high load prediction, the machine learning algorithm can be expanded to select the etcd node with the lowest predicted network IO latency and disk IO latency to serve as the etcd leader when the demand for pod creation and cluster state updates on the kube-apiserver scales up quickly or changes dramatically.

Bibliography

[1] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *Proc. USENIX ATC*, 2014.

[2] Overview | kubernetes. https://kubernetes.io/docs/concepts/overview/. Ημερομηνία πρόσβασης: 7-8-2023.

[3] Kubernetes components | kubernetes. https://kubernetes.io/docs/concepts/overview/components/. Ημερομηνία πρόσβασης: 7-8-2023.

[4] etcd. https://etcd.io/. Ημερομηνία πρόσβασης: 8-8-2023.

[5] Raft consensus algorithm. https://raft.github.io/. Ημερομηνία πρόσβασης: 8-8-2023.

[6] The go programming language. https://go.dev/. Ημερομηνία πρόσβασης: 8-8-2023.

[7] What is etcd? | ibm. https://www.ibm.com/topics/etcd. Ημερομηνία πρόσβασης: 8-8-2023.

[8] Role-based access control. https://en.wikipedia.org/wiki/Role-based_access_control. Ημερομηνία πρόσβασης: 8-8-2023.

[9] Kv api guarantees. https://etcd.io/docs/v3.5/learning/api_guarantees/. Ημερομηνία πρόσβασης: 8-8-2023.

[10] Faq | etcd. https://etcd.io/docs/v3.5/faq/#what-is-failure-tolerance. Ημερομηνία πρόσβασης: 8-8-2023.

[11] How etcd works with and without kubernetes. https://learnk8s.io/etcd-kubernetes. Ημερομηνία πρόσβασης: 8-8-2023.

[12] etcd3 api. https://etcd.io/docs/v3.2/learning/api/#watch-api. Ημερομηνία πρόσβασης: 8-8-2023.

[13] etcd client architecture. https://etcd.io/docs/v3.3/learning/client-architecture/. Ημερομηνία πρόσβασης: 23-8-2023.

[14] grpc load balancing. https://grpc.io/blog/grpc-load-balancing/. Ημερομηνία πρόσβασης: 24-8-2023.

[15] Consensus (computer science). https://en.wikipedia.org/wiki/Consensus_(computer_science). Ημερομηνία πρόσβασης: 11-8-2023.

[16] Raft consensus algorithm. https://raft.github.io/. Ημερομηνία πρόσβασης: 10-8-2023.

[17] Heidi Howard. Arc: Analysis of raft consensus. Technical Report UCAM-CL-TR-857, University of Cambridge Computer Laboratory, 15 JJ Thomson Avenue Cambridge CB3 0FD United Kingdom, 2014.

[18] Raft consensus algorithm - geeksforgeeks. https://www.geeksforgeeks.org/raft-consensus-algorithm/. Ημερομηνία πρόσβασης: 20-8-2023.

[19] Frequently asked questions | etcd. https://etcd.io/docs/v3.3/faq/#do-clients-have-to-send-requests-to-the-etcd-leader. Ημερομηνία πρόσβασης: 22-8-2023.

[20] Nitos. https://nitlab.inf.uth.gr/NITlab/nitos. Ημερομηνία πρόσβασης: 26-8-2023.

[21] Nitlab. https://nitlab.inf.uth.gr/NITlab/. Ημερομηνία πρόσβασης: 26-8-2023.

[22] Outdoor hidden - nitlab. https://nitlab.inf.uth.gr/NITlab/outdoor-hidden. Ημερομηνία πρόσβασης: 28-8-2023.

[23] Indoor hidden - nitlab. https://nitlab.inf.uth.gr/NITlab/indoor-hidden. Ημερομηνία πρόσβασης: 28-8-2023.

[24] Docker container. https://www.docker.com/resources/what-container/. Ημερομηνία πρόσβασης: 29-8-2023.

[25] Prometheus. https://prometheus.io/docs/introduction/overview/. Ημερομηνία πρόσβασης: 29-8-2023.

[26] About grafana. https://grafana.com/docs/grafana/latest/introduction/. Ημερομηνία πρόσβασης: 29-8-2023.

[27] How to use the linux traffic control. https://netbeez.net/blog/how-to-use-the-linux-traffic-control/. Ημερομηνία πρόσβασης: 6-9-2023.

[28] Tensorflow. https://www.tensorflow.org/. Ημερομηνία πρόσβασης: 9-9-2023.

[29] What is prometheus | grafana documentation. https://grafana.com/docs/grafana/latest/fundamentals/intro-to-prometheus/. Ημερομηνία πρόσβασης: 30-8-2023.

[2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 1, 16, 17, 15, 18, 19, 13, 14, 21, 20, 22, 23, 24, 25, 26, 29, 27, 28]