



POLITECNICO DI TORINO  
Master Degree course in Computer Engineering

Master Degree Thesis

# State-Aware Migration in Edge Mobile Application

## **Supervisors**

Prof. Carla Fabiana Chiasserini

Prof. Paolo Giaccone

## **Candidate**

Mostafa Tavassoli Norouzi

ACADEMIC YEAR 2022-2023

# Acknowledgements

First and foremost, I would like to thank God for giving me strength and encouragement throughout all the challenging moments of completing this dissertation.

Words cannot express my gratitude to my professors, Carla Fabiana Chiasserini and Paolo Giaccone, for their invaluable patience and feedback. Their expertise and encouragement helped me to complete this research and write this thesis.

I would also like to thank Antonio Calagna, engineer and Ph.D. student for his support whenever I ran into a trouble spot or had a question about my research or writing throughout my thesis.

I want to thank Politecnico di Torino and the technical teams for the opportunity to use CrownLabs to do all my experiments. To my professors, classmates, and friends who were part of my professional development throughout my study.

Finally, I must express my profound gratitude to my parents and my family members for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you.

## Abstract

In modern cloud-based applications, containerized services play a significant role in gaining the benefits of deployment and development. Although virtual machines have become the predominant technology for virtualization, the container has been replaced with VMs because of its benefits. Since it shares the host OS kernel, it brings the benefits of a light container, a fast booting procedure, and migration quickly.

In a large-scale application with numerous services like microservices, manipulating and testing this application with several dependencies and configurations to provide zero downtime services to the customers could be a sophisticated process. In this scenario, the orchestrator applications like Kubernetes come to the rescue to enhance the management of these big applications.

Kubernetes is architected as a collection of microservices to govern the lifecycle of containers that activates the stateless migration thanks to its feature as a container orchestrator. Nevertheless, some microservices might heavily count on the internal state; hence migrating them stateless leads to service disruption. We, consequently, investigate stateful migration as a technique to support microservice mobility while saving the state and minimizing service disruption.

Since stateful migration is a complicated process because it impacts the CPU context state, and TCP connection, we propose an alternative approach to combine stateless migration with a technique of spreading the state. We employ the stateless migration for microservices but keep the state through the distribution of it across the multiple instances of the edge computing nodes. To achieve this objective as state distribution, we investigate various techniques based on a guaranteed level of consistency and propose a performance evaluation. In this case, we evaluate the performance of etcd as the popular solution to fulfill the state distribution across the cluster via providing a strong consistency level and then compare it with OrbitDB as a technique to ensure a weaker consistency level.

Therefore, we present three architectures as Centralized, Reactive, and Proactive approaches to implement our proposal of stateless migration with preserving the state. Then, we implemented a realistic testing scenario to highlight their performance trade-offs. In the Centralized approach, although the etcd is central in one host without needing to migrate, there is a single point of failure. In Reactive, the etcd keeps inside the same host as the microservice, which is ideal for resource consumption, but migration time lasts longer. Lastly, in Proactive, we have many nodes with CPU allocation because having etcd instances in each node is necessary without needing to migrate the state that guarantees quick migration, but we have high consensus overhead.

In conclusion, based on our result, it is clear that this solution is feasible, easy to achieve, effective, and simple to implement using only the provided features by Kubernetes. In the end, we measure service disruption duration caused by migration; for instance, in Centralized and Proactive approaches, this duration is very low compared to Reactive. Although our solution is straightforward to obtain, there is an impact on the throughput of microservice because of distributed state, we need to consider realistic

network conditions and the overhead introduced by the consensus algorithm to ensure state distribution.

# Contents

<b>1</b>	<b>Introduction</b>	<b>5</b>
1.1	Overview	5
1.2	Thesis Objectives	6
1.3	Thesis Structure	7
<b>2</b>	<b>Background on Fundamental Technologies and Tools</b>	<b>9</b>
2.1	Stateless and Stateful Applications	9
2.2	Virtualization Technologies and Tools	11
2.2.1	Hypervisors	11
2.2.2	Containers	12
2.3	Manage the Consistency	17
2.3.1	Data Consistency Challenge	17
2.3.2	CAP theorem	18
2.3.3	Raft Algorithm	19
2.3.4	Conflict-free replicated data type - CRDT	21
2.4	Distributed Storage	22
2.4.1	IPFS concept	22
2.4.2	etcd	26
2.4.3	Redis	27
2.4.4	OrbitDB	28
2.4.5	etcd vs orbitDB	29
<b>3</b>	<b>Literature Review</b>	<b>31</b>
3.1	Migration Concept and Performance	31
3.2	Stateful Migration Techniques	34
3.3	Virtualization Concept	36
<b>4</b>	<b>Experimental Setup</b>	<b>39</b>
4.1	Methodology	39
4.2	Preliminary Experiments	43
4.2.1	Configuring the environment	43
4.2.2	The experiment of migration nodes in Kubernetes	45
4.2.3	Simple CRUD operation using IPFS and OrbitDB	52
4.2.4	Helm Charts experiment	60

<b>5</b>	<b>Experimental Results</b>	65
5.1	Effect of CPU allocation limit on Migration . . . . .	65
5.2	etcd experiment . . . . .	70
5.3	OrbitDB Experiment . . . . .	73
5.4	etcd in Kubernetes cluster experiment . . . . .	77
5.5	Stateless Migration Approaches . . . . .	78
5.6	etcd performance . . . . .	79
5.7	Throughput in Producer and Consumer . . . . .	86
5.8	Realistic Networking Scenario . . . . .	88
<b>6</b>	<b>Conclusion and Future Work</b>	95
	<b>Bibliography</b>	97

# Chapter 1

## Introduction

### 1.1 Overview

One of the most vital concepts in modern cloud and edge environments is the containerization of services for achieving the benefits of development, shipment, and deployment. The container is the standard virtualization technology that packs the code and all its dependencies lead to isolated software from its environment. Therefore, the application can run and transfer reliably and fast from one computing environment to another. This benefit, known as portability, is achievable by containerization leading to application migration.

A container can run a small microservice, software process, or large-scale application. Inside a container are all the necessary executables, binary code, libraries, and configuration files to run the application. However, it does not include operating system images making them more lightweight and portable, with remarkably negligible overhead. Containers can migrate in two modes, a stateless or stateful manner. In the first approach, the state loses because of the new container creation process in the final destination from scratch since the actual container in the source disappears. However, if the state needs to preserve, stateful meets this requirement. It transfers the container state to the destination. As a result, the container is the same as before migration.

In larger-scale application deployments, multiple containers deploy as one or more container clusters. This cluster needs to handle by a container orchestrator such as Kubernetes. It is one of the most famous orchestrators, a system for automating deployment, scaling, and manipulating containerized applications. This open-source system operates a Master / Worker architecture wherein a master node manages and handles worker nodes that execute container workloads via an API Server. It leads to the creation of robust infrastructure and highly available applications.

Kubernetes is architected as a collection of microservices to govern the lifecycle of containers and coordinate application administration tasks such as configuration, and deployment. The nature of distributed microservices architecture demands a reliable and persistent data store that can act as a single source of the truth called etcd. etcd is very popular and plays an essential role to distribute the state across the cluster. In this case, we evaluate the performance of etcd as the popular solution to fulfill the state

distribution across the cluster via providing a strong consistency level and then compare it with OrbitDB as a technique to ensure a weaker consistency level. It is essential to mention that it builds on the Raft consensus algorithm to guarantee strong consistency. Raft divides the nodes in the cluster into a Leader and Followers. The elected Leader node organizes the data replication for all of the Follower nodes in the etcd cluster.

## 1.2 Thesis Objectives

The main objective of the current study is to investigate stateful migration as a technique to support microservice mobility while saving the state and minimizing service disruption. Since stateful migration is a complex process because it impacts the CPU context state, and TCP connection, we propose an alternative approach to stateful migration by combining stateless migration with a technique of distributing the state. We operate the stateless migration for microservices while keeping the state via the distribution of it across the multiple instances of the edge computing nodes.

The etcd is the main character in the study. The migration time was experimented with in several scenarios. The main one is in a condition of etcd using RAFT with OrbitDB using the CRDT technique. In this case, two strategies of consistency, one delivers via RAFT as Strong consistency, and the second provides via CRDT as Eventual, lower level of consistency can compare to each other. Moreover, these scenarios were experimented with both in a virtual machine with manual etcd configuration to create a proper cluster and in a more advanced version with etcd/bitnami using Helm charts within the kubernetes cluster with three nodes and with the different number of etcd peers to realize the effect of consensus overhead.

Gradually, to track the stateless migration, three strategies define to detach the microservices state. We present three architectures as **Centralized**, **Reactive**, and **Proactive** approaches to implement our proposal of stateless migration with preserving the state. Then, we implemented a real testing scenario to emphasize their performance trade-offs. In the Centralized approach, although the etcd is central in one host without needing to migrate, there is a single point of failure. In Reactive, the etcd supports inside the same host as the microservice, which is perfect for resource consumption, whereas migration time lasts longer. Lastly, in Proactive, we have many nodes with CPU allocation because having etcd instances in each node is required without needing to migrate the state that assures quick migration, but we have high consensus overhead.

Moreover, we measured the throughput of etcd operations per second in a scenario with the applications as Producer and Consumer deployed in the pod using Docker to do advanced and realistic experiments. In this case, the R/W request sends from the producer and consumer rather than from the testing application running in the master node. There are various scenarios defined to generate the results, like sending the request directly to the leader or to the load balancer to understand the throughput of our system.

In conclusion, based on our result, it is clear that this solution is feasible, easy to achieve, effective, and simple to implement using only the provided features by kubernetes. In the end, we measure service disruption duration caused by migration; for instance, in Centralized and Proactive approaches, this duration is very low compared



to Reactive. Although our solution is straightforward to obtain, there is an impact on the throughput of microservice because of distributed state, we need to consider realistic network conditions and the overhead introduced by the consensus algorithm to ensure state distribution.

## 1.3 Thesis Structure

The rest of the thesis is organized as follows.

Chapter 2, related to Background on Fundamental Technologies and Tools, presented the essential concepts and technologies used during the research.

Chapter 3 provides the Literature Review of migration in different investigations. Studying this concept in edge, and Fog computing, defining a novel approach in terms of reducing the time of containerization and the proper protocol in a migration scenario.

Chapter 4, the Experimental Setup, provided the methodology that we used to do the entire experiment. Moreover, this chapter contains preliminary experiments before starting the next chapter including the results. In this chapter, a related configuration is provided for creating the testbed scenario.

Chapter 5 is the Experimental Results related to the objective of the study. We did a various number of experiments and provided many outcomes for various scenarios. In this chapter, we discussed the etcd providing strong consistency, then compared it with CRDT as the weaker consistency. In the end, we created a realistic networking scenario to evaluate the performance trade-offs of our three approaches.

The last chapter includes the conclusion of the thesis and future work.



## Chapter 2

# Background on Fundamental Technologies and Tools

This chapter provides the background information about the entire concepts that we need for the rest of the document. The concept of state, virtualization technologies, the CAP theorem, the data consistency challenge, and the different types of distributed storage.

### 2.1 Stateless and Stateful Applications

The application state refers to its condition at a specific moment. Being stateless or stateful depends on the recorded time for interaction with it and the way the information is stored. In stateless applications, no reference to past information preserves by the system, and each transaction starts from scratch like the first time. For example, when the user sends the query to the search engine, the user request gets interrupted because, for some reason, they have to start a new search.

The stateless provide a service or functionality at a time and employs the CDN - content delivery network - or print server to handle the short-term requests processes. The stateless system brings some advantages as having an architecture to scale up and down straightforwardly. Moreover, it declines the number of resources, like storage, since there is no need to preserve the transaction. On the other hand, the performance of the network might decrease due to many requests sent repetitively.

Stateful processes, nevertheless, perform within the context of previous transactions. It means that the current transaction might be affected by the condition of the previous one. Since the context and state of the application have been stored, if the transaction interrupts, it can be picked up from whenever there is an incomplete transaction. A stateful transaction uses the same server to process the requests. It can be defined as a foundation for several technologies like FTP and Telnet. The stateful protocol can provide better performance since it has the proper information for a future transaction. Moreover, it is popular in the banking and finance sector because of the extra layer of security. However, the performance is dependent on the efficiency of the network memory. And the memory needs to be considered as part of its server architecture.

In stateful applications, the databases are used to hold the state from client requests

and employ the stored data for further requests. In this scenario, once the user starts logging into the system after the first time, the server does not call DB because the session information exists on the server. On the other hand, all state and data stored on DB are features of persistent storage in stateless applications. It means that the server generates an auth token and returns it to the client. The next time when the user wants to operate some tasks with the applications needs to pass that token to the server in each request. A stateless system means that each HTTP request will happen in complete isolation. When users send an HTTP request, it contains all the required information to the server to fulfill that request. However, the stateless applications can be treated as stateful ones by using the concept of REST APIs in which developers can augment HTTP making stateless apps produce stateful behavior. Although stateless applications can process some client interactions slowly, it offers scalability effectively by simulating the stateful operation [42].

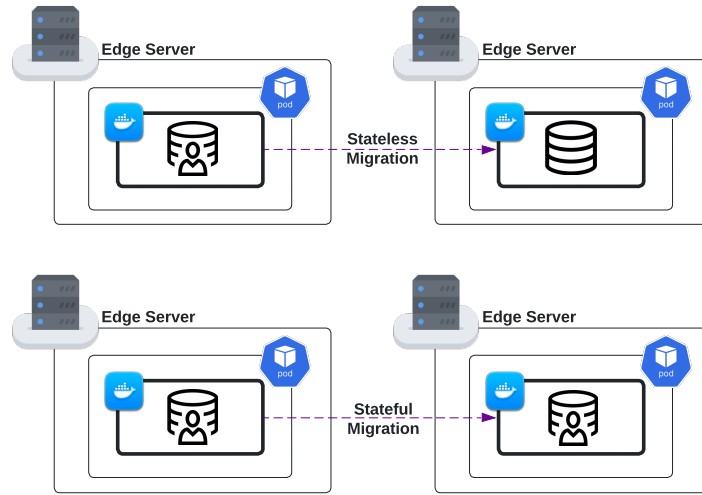


Figure 2.1: The state is not stable in Stateless migration, however, the state is kept in Stateful one

Whether using stateless or stateful applications, using containerized applications has increased drastically as the popularity of cloud computing and microservices grows. Containers are deployed applications bundled with all necessary dependencies and configuration files. All of the elements share the same OS kernel. Since the container is not tied to any one IT infrastructure, it can run on a different system or the cloud. An application that is being developed and deployed is bundled and wrapped together with all its configuration files and dependencies. This bundle is called a container. Therefore, they can move and run in any environment easily, on a desktop, traditional IT infrastructure, or on the cloud. Based on the definition, containers are designed to be stateless in their nature.

The stateless applications need to be put into the microservice architecture. Microservices break the applications into smaller independent components and work together to achieve the same tasks. Each of these components is a microservice with the ability to share similar processes across multiple applications. Containerizing microservice allows running multiple microservices in isolation on the same server as other ones saving resources. In this scenario, a tool like Kubernetes is beneficial for performing automatic load balancing to manage several microservices-based applications. Once a stateless containerized microservice generates, it supports scalability and availability once the traffic increases automatically by generating a new instance of individual application components.

**Scaling Challenges** In stateful applications, scaling is a challenging section, although their applications provide historical context and provide efficient client interactions. The users interacting in this system need to send the requests to the same system or server keeping the user system information repetitively because in this system the users bind to the same server. It leads to preventing sending the users' requests to the other systems running the same needed running application. Stateful applications are not perfect options in an unpredictable amount of traffic leading to delays in sending the response back. Using stateless applications can prevent this challenging situation. It acts like a nameless agent to the clients for interacting with the databases it connects with. In this case, the application uses a load balancer to replicate a new instance of the application and delegate the requests among those instances allowing the system to scale based on any level of traffic. Having the feature of scaling makes stateless applications crucial for creating modern cloud computing. The users start communication via stateless HTTP protocol to connect with stateless web services. Each request sends in an isolated manner to the servers, once the traffic increases drastically, the load balancer begins to replicate the web service in a new server to handle the user's request by redirecting to it.

## 2.2 Virtualization Technologies and Tools

### 2.2.1 Hypervisors

A hypervisor is a delicate software layer also known as a virtual machine monitor (VMM). Before their presence, most computers could only operate one operating system (OS) at a time. With a hypervisor, you can run numerous operating systems using one host machine. This practice helps decline the waste of physical servers and computational resources.

Hypervisors divide a system's operating system and resources from the physical machine. They manage these isolated resources into files called virtual machines (VMs), hence the alias virtual machine monitor. Then, they allocate computing power, data, and storage to each one. A hypervisor restricts these files from interfering with one another, thereby maintaining the system.

Hypervisor technology permits more use of a system's available computing resources. They preserve space and maintenance because they create separate operating systems

that share the resources of a single machine.

### 2.2.2 Containers

Containers are a way to package and deploy applications in a lightweight, portable, and consistent way to package an application and its dependencies into a single unit called a container image that runs on any compatible system. They work based on the isolated environment idea, where each container has its filesystem, networking, and process space, allowing multiple containers to run on the same host without conflicts. They often employ along with the container orchestration platforms, such as Kubernetes providing additional capabilities for managing and coordinating containers in a distributed system [8].

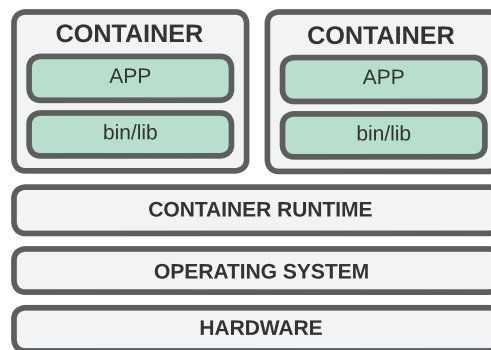


Figure 2.2: Container Deployment

Key benefits of using containers include,

- **Portability:** they can move between different environments, such as development, testing, and production, without changes to the application code.
- **Consistency:** containers guarantee that an application operates the same way regardless of the environment it deploys in.
- **Resource efficiency:** containers allow the users to run multiple applications on the same host while still isolating their resources, leading to better resource utilization and cost savings.
- **Simplicity:** They allow you to focus on producing and deploying your application rather than stressing about the underlying infrastructure.

**Orchestration** Orchestration refers to organizing and harmonizing containers process in a distributed system. It involves planning containers onto nodes in a cluster, managing the network and storage connections between them, and assuring that the containers are running as expected. Orchestration typically manages by a container orchestration

platform, such as Kubernetes, Docker Swarm, or Mesos. These platforms supply various features for managing and coordinating containers,

- **Scheduling:** Container orchestration platforms can automatically organize containers onto nodes based on various criteria, such as resource availability and constraints.
- **Service discovery:** they can provide service discovery capabilities, permitting containers to communicate with each other and with external clients using DNS names.
- **Load balancing:** providing load-balancing capabilities, allowing traffic to broadcast across multiple containers.
- **Self-healing:** watching the health of containers and automatically replacing or restarting them if they fail.
- **Rolling updates:** they can support rolling updates, allowing the users to update the application without experiencing downtime.

**Kubernetes** It [26] is an open-source container orchestration tool which is developed by Google. It Helps to manage the applications that are made up of hundreds of containers. Moreover, it manages them in different environment like physical, virtual, and cloud. The rise of using containerized technologies, like Microservices, the container offers the perfect host for a small independent application like microservices. The growth of using Microservices results in employing several containers. Therefore, managing these situations across multiple environments using scripts and the self-made tool could be really complex and sometimes impossible. So, having a container technology would be necessary.

In the traditional way of deployment, the applications ran on a physical server. The big issue of this kind of deployment was resource boundary allocation. It means that if multiple applications run on a physical server, only one could take up all resources, and others would be unperformed. In terms of scaling the application, more physical servers had to operate leading to expensive maintenance. To address this issue, virtualization technology was introduced, allowing to run of multiple Virtual Machines - VMs - on one single physical server's CPU via hypervisor technology. It provides better resource utilization in a physical server and better scalability. Each VMs has a full machine running all required components like the operating system on top of the virtualized hardware. However, container deployment shares the Operating System across the applications instead. Hence, the containers are considered lightweight since it does not need a full OS. They are an abstraction as the application layer to package the code and its dependencies, running as isolated processes in user space.

### **Kubernetes Components**

A cluster of Kubernetes includes worker nodes running the containerized applications. Each cluster has at least one worker node. Pods hosted by worker nodes are the smallest

unit of k8s which are an abstraction over the container. It creates the running environment or layer on top of the container [26].

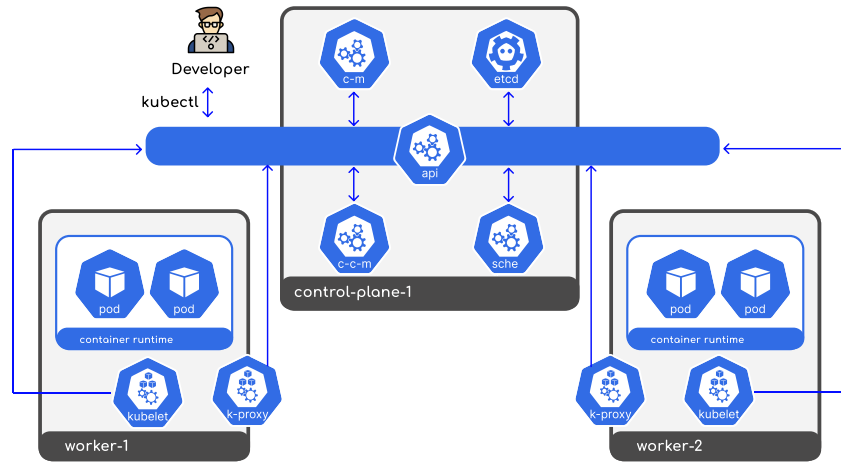


Figure 2.3: Kubernetes Architecture

- **kube-apiserver**, the user can deploy the new app by communicating through this process with the client. The client could be a UI, k8s dashboard, or k8s command line kubelet. It is like a cluster gateway that gets the initial request of any updates into the cluster or even the queries from the cluster. It also acts as a gatekeeper for authentication. To schedule new Pods, deploy a new app, create a new service, or any other components, the users have to talk with the API server on the master node. It validates the request then it allows the request to move forward.
- **etcd**, It is a consistent and highly-available key value store of the cluster state. Every change in the Cluster, new Pods schedule, or when Pods die, all of these changes save and update into this K, V store of etcd. It is important because all of the stuff that happens in other processes do base on the data that is stored in this process. How does the scheduler know about the number of resources available in each node? How does the controller manager know that the state of the cluster has changed? The health of the cluster is required by the API server. All of this information is stored in the etcd cluster.
- **kube-scheduler**, starts Pod in one of the worker nodes. It can pick which pods are going to handle that process instead of choosing randomly. It first looks at the request and analyzes that to understand how much CPU, Memory it needs, then go for each worker node and discovers how many resources are available for each one to handle this request. Several factors need to take into account for scheduling decisions, like hardware, software constraints, affinity and anti-affinity specifications, data locality, and deadlines.



- **kube-controller-manager**, has to detect when pods die on any node to re-schedule it again with new pods as one of its controller's responsibilities. It runs the controller process and detects the cluster state changes like the crashing of pods. It requests the Scheduler to re-scheduling those pods and the rest happens like before as the Scheduler does.

The components related to nodes essentially maintain running pods and provide the k8s runtime environment.

- **kubelet**, is a primary node agent running on each node. It ensures that containers run inside the pod based on the PodSpecs that are provided via several mechanisms. PodSpec is a YAML or JSON format that describes a pod.
- **kube-proxy**, is a Kubernetes component that runs on each node in a Kubernetes cluster. It is responsible for implementing and maintaining network connectivity to the pods running on the node. It does this by setting up NAT rules and port mappings to forward traffic from external sources to the correct pod. In other words, Kube-proxy acts as a load balancer and networking proxy for the pods on a node. It listens for traffic on specific ports and routes it to the correct pod based on the service configuration. It also provides service discovery for the pods, making it possible for them to communicate with each other and with external clients using DNS names.
- **Container runtime**, is software for running the containers supplying the underlying infrastructure that allows containers to run on a host operating system. There are several container runtime options available, including:
  - Docker, a popular container runtime to run containers based on the Docker image format.
  - containerd, a lightweight container runtime designed to be easy to integrate into other systems. It is operated by several container orchestration platforms, including Kubernetes.
  - rkt - (pronounced "rocket") - a container runtime implemented to be simple, secure, and composable. It is an alternative to Docker.

## Pods, ReplicaSets, Deployments

A **pod** is the most diminutive deployable unit expressing a group of containers that are deployed together on the same node. Pods use to host the containers that make up an application, and they are the basic building blocks of a Kubernetes cluster. Each one has its IP address and can include one or more containers, and shared storage, such as a volume. They implement to be temporary, meaning they are expected to be terminated and replaced at any time. It permits Kubernetes to scale and manage applications by creating, destroying, and replacing pods as needed straightforwardly. Pods are typically handled via higher-level Kubernetes objects, such as Deployments,

ReplicaSets, and StatefulSets, managing the details of scaling, rolling updates, and self-healing. Pods are crucial in Kubernetes, providing a way to group related containers and manage them as a single unit. They allow the users to share resources, like storage and networking, between containers in the same pod.

**ReplicaSet** is a controller that has the responsibility of preserving a stable set of replicas of a pod. A ReplicaSet guarantees that a specified number of pod replicas are running at any given time by creating or deleting pods as needed to match the desired state. A ReplicaSet defines by a specification that restricts the desired number of replicas and the template for the pods that the ReplicaSet should manage. It uses this specification to create and delete pods as needed to ensure that the desired number of replicas is always running.

A **deployment** is a higher-level object that is responsible for supervising the rollout and management of a group of replicas of a pod. It uses ReplicaSets to ensure that a specified number of pod replicas are always running and to recover from failures automatically. It defines by a specification that determines the desired number of replicas and the template for the pods that the Deployment should manage. It employs this specification to create and delete ReplicaSets as needed to ensure that the desired number of replicas is always running and to perform rolling updates. Deployments are an important concept in Kubernetes, as they provide a way to manage the rollout and management of a group of replicas in a declarative way. They also provide a way to perform rolling updates, allowing you to update your application without downtime. The `kubectl` command-line tool or the Kubernetes API use to create a Deployment object and specify the desired number of replicas and the pod template. The Deployment will then take care of creating and deleting ReplicaSets and pods as needed.

## Kubernetes Services

A service is an abstract manner to expose an application running on a set of Pods as a network service. Once the service creates, it is necessary to specify which Pods should include in the Service. The service provides a stable, virtual IP address (VIP) and DNS name that other Pods, Services, and external clients can use to access the Pods. It employs to disclose a variety of applications, including microservices, batch jobs, and other workloads. In addition, it can be used to expose applications that are running on different nodes in a cluster, or even in different clusters. There are several types of Services in Kubernetes, each with a specific use case,

- **ClusterIP**: it is the default service that exposes the Service on a cluster-internal IP. The service is only accessible within the cluster.
- **NodePort**: Exposes the Service on each Node's IP at a static port. A ClusterIP Service, to which the NodePort Service routes, is created automatically. It provides the users with freedom to set up the load balancing to configure environments that are not fully supported by Kubernetes, or even to expose one or more nodes' IP addresses directly. It can be targeted from outside the cluster using `<NodeIP>:<NodePort>`.

Services are defined in a YAML file to create, update or delete the service. For example, here is a sample YAML file that defines a ClusterIP service.

```
1 apiVersion: v1
2 kind: Service
3 metadata:
4   name: my-service
5 spec:
6   selector:
7     app: my-app
8   ports:
9     - protocol: TCP
10     port: 80
11     targetPort: 8080
```

This service would reveal the pods with the label `app: my-app` on a cluster-internal IP, on port 80. The pods would be accessible on port 8080.

## Concept of Stateful and Storage Provisioner

A container in its nature is ephemeral meaning that it dies and the new one starts in a clean state. This condition makes an issue for the stateful applications when the state is essential to keep. To provide the feasibility of stateful applications the concept of PersistentVolume and PersistentVolumeClaim can be employed. The PersistentVolume (PV) is a storage piece in the cluster that is provisioned by an administrator or automatically via Storage Class. It is a mechanism in a k8s cluster to provision PV dynamically. Once the Storage class is configured, we can claim the volume via a PersistentVolumeClaim (PVC). It is a kind of request for storage by a user. PVC consumes the PV resources while the Pods use the node resources. Unlike the Pods that request a certain level of resources like Memory and CPU, the PVC sends the request for a specific size and access mode.

## 2.3 Manage the Consistency

### 2.3.1 Data Consistency Challenge

A set of challenges stem from the distributed system like data consistency, task synchronization, and issues because of network partitions. One resolution to tackle these issues in particular for coordination is consensus algorithms, allowing the collection of machines working together as a coherent group to prevail over the failures of the components. In this context, state machines on a server's collection operate similar copies of the same state and continue to perform even if some servers crash.

There would be a data consistency challenge when there might be temporal differences in the state of each distributed replica because of network latency. Suppose a condition where different elements each contain a copy of the same data. Data consistency holds if those copies match each other even if one or more of them are updated. If one node changes data and updates its copy of the data, there is no disagreement about the source of truth. However, there might be an issue to reach the updated data to other nodes leading to failure of distribution of the last version of data. Moreover, if multiple entities

make independent updates simultaneously, in this scenario, there is no single correct version of data. It might cause conflict issues with other peers. There are two potential methods of dealing with such data mutations.

One is strongly consistent replication. In this model, the replicas coordinate with one another to consensus on how and when to apply the changes. This approach activates strong consistency by using serializable transactions and linearizability. Nevertheless, this way decreases the performance of waiting for the coordination. In addition, according to the CAP theorem, it is unfeasible to replicate data to nodes that are disconnected from the rest of the network due to network partitioning.

The second approach is optimistic replication or eventual consistency. The users modify the data on any replica independently of any other replicas, even in offline mode and network partitioning. Although, there would be the risk of consistency and conflicts when the users concurrently modify the same piece of data, leading to maximum performance and high availability. Finally, the created conflicts need to resolve when the replicas initiate contact with together.

### 2.3.2 CAP theorem

The CAP theorem - Consistency, Availability, Partition tolerance - notes that any distributed data store can deliver only two of the three mentioned guarantees. A network that stores data on more than one node physically or virtually simultaneously is called a distributed system. Applying the proper data management system in the distributed system leads to delivering the characteristics of the application required most [27].

The term **consistency** means that all clients across the system have a consensus of the shared values; hence, they see the same data at the same time. It insists that the written data by one node should be propagated to all other nodes in the system before this operation considering successful.

The **availability** concept guarantees that client data requests (read/write) will access the shared values even if one or more nodes are unreachable. In other words, all operational nodes in the system reply with a valid answer to requests without peculiarity.

The last concept, **Partition tolerance**, ensures that the cluster needs to operate despite an arbitrary number of messages being dropped or delayed because of the network partitioning. Network partition happens when a computer network splits into relatively separated subnets by strategy for optimization or because of network device failure. Therefore, the distributed system should design to be partition tolerant so that even after network partitioning, the system is still responsible.

In network partition, one of two concepts needs to be guaranteed by the system. In the first concept, for the sake of consistency, the operation will cancel and decrease availability. On the other hand, accept the risk of inconsistency, proceed with the operation and provide availability. Since partition tolerance is like distributed data stores, meaning there will be unreachable nodes in the network - because of the unstable nature of the internet; CAP theorem states that between Consistency and Availability, in the presence of the Partition, one has to select.

Consistency over Availability (CP)

The outcome of particular information will be an error or a time-out because it cannot

be guaranteed to be updated because of network partitioning. Having this approach in a real-world distributed system leads to making a high-accurate data store; however, the system will be most likely unavailable for a specific time. Because when a partition occurs between any two nodes, the system has to shut down the non-consistent node until the partition is resolved.

#### Availability over Consistency (AP)

This option responds to every request to the network, although the system can not guarantee provided data is up to date due to network partitioning. It makes the system high-available, but the returned data will be outdated in most cases. When a partition happens, all nodes remain available; those at the sinful end of the partition might return an older version of data than others. Once the partition resolves, the system resyncs the nodes to ensure consistency.

etcd ensures both **linearizability** - consistency of replicas - and **serializability** to form a **strict serializability** [9].

The former - **linearizability** [21] - is one of the strongest single-object consistency. It means that every operation appears to take place atomically, in some order, uniform with the real-time series of those operations. It ensures that the read operation will return the most current value. For example, if client A writes at time  $t_1$  then issues read at time  $t_2$  ( $t_1$  less than  $t_2$ ) so the client needs to get the recent value of writing at  $t_1$ . Nonetheless, without having linearizability, the read operation might obtain the stale data at time  $t_3$  since a concurrent write operation may happened between  $t_2$  and  $t_3$ . In this model, once a network partition happens, it can not be sticky available because some nodes will not be able to make improvements.

The latter one, **serializability** [22], guarantees transactions to perform a group of one or more operations over one or more objects to execute a set of transactions over multiple items equal to serial execution. Since it is a transactional model, the operations can affect several primitive sub-operations in order. This model implies the read operation is repeatable since it does not operate under any real-time or per-process constraints. It means that, if process A writes, then process B begins to read, it is not guaranteed to see whatever is written by process B. Moreover, it does not require ordering per process during transactions.

### 2.3.3 Raft Algorithm

Raft [34] is a consensus algorithm that enables a CP system. Consensus is a rudimentary issue in fault-tolerant distributed systems. Distributed consensus is a way to convey the idea of getting more than one party to agree on something. However, having an  $N$  machine agree on a single state is very hard. To solve this issue, Raft comes to the rescue. Raft is an algorithm to manage consensus-based systems. It is also a leader-based algorithm making use of a single leader for all decisions. The role of the leader can alter in a specific situation when network partitioning occurs. This algorithm continues to progress when the majority of servers are available. In a network of  $n$  number of nodes, there must be at least  $2f+1$  servers to handle the failures properly ( $f$  is the number of failing nodes). For instance, in a cluster of five servers, the system operates even if two servers fail; if more servers down, they stop working. Raft servers contact each other by using remote

procedure calls (RPCs). Basically, two types of RPCs need to be used in the consensus algorithm. One is RequestVote RPCs that employ by candidates in election time and another one is AppendEntries RPCs activated by leaders to replicate log entries and to create the heartbeat.

A node can have a specific role in Raft as follows.

- **Leader**, only the node elected as leader can interact with the client. All other nodes sync up themselves with the leader. At any point of time, there can be at most one leader. The leader is responsible for accepting client requests and controlling the replication of the log to other servers. The data flows only in one direction, from the leader to other servers.
- **Follower**, follower nodes sync up their copy of data with that of the leader data after every regular time intervals. When the leader goes down, one of the followers can contest an election and become the leader. It is a passive state in which the node can not send requests but only operates as a recipient.
- **Candidate**, at the time of contesting an election to choose the leader, the nodes can ask others for votes. Hence, they are called candidates when they have requested votes. Initially, all nodes are in the Candidate state.

There are two steps to selecting a leader base on the assumption that there is no existing leader. In the first stage, all nodes start as followers, if the system starts for the first time or any part of the system fails, that part begins as followers upon recovery. In the second step, the first node that its timer stops before the rest of the nodes; transfer to the candidate role. This node increases its term and sends the vote request to every node in the cluster. If the node gets votes back from the majority of other nodes, this node becomes the leader and sends the heartbeats to the other nodes.

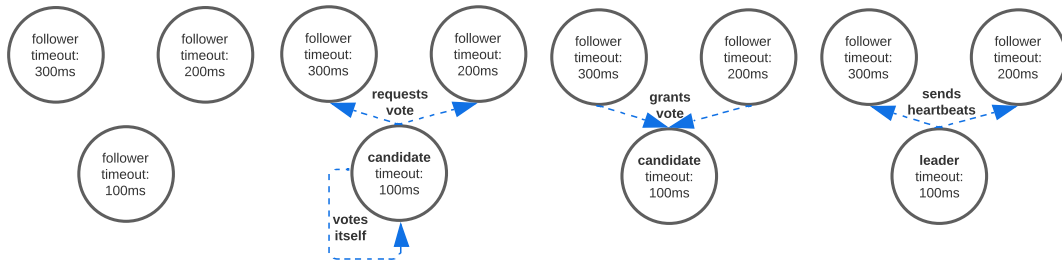


Figure 2.4: leader election in RAFT algorithm

The term concept means an arbitrary period on the server for which a new leader needs to be selected. Each term begins with a leader election when the leader server fails to operate. If the candidate gets the majority of votes and becomes the leader, otherwise no leader is called a split vote and a new term will start and the candidate returns to the follower state.

The leader is responsible to replicate the logs in the log replication approach to ensure consistency among all peers in the cluster. Each client requests consist of a command to be executed by the state machine in the cluster. It is essential to mention that only the leader interacts with the client and any requests need to be redirected to the leader node. Once the leader gets the new log, it forwards it as an `AppendEntries` message to the followers. Then, all nodes get the new logs via confirmation from the followers, it applies the log to its local log. The mentioned request is called committed.

#### 2.3.4 Conflict-free replicated data type - CRDT

It is a replicated data structure that uses an optimistic replication approach across multiple computers and simplifies replication conflicts within a decentralized environment. Storing a copy of data on multiple computers/replicas in some distributed systems is necessary. In this case, the application can update the replica without coordinating with others independently. The inside algorithm guarantees to resolve if there might be any inconsistencies and to converge different states of replicas at any specific point. In other words, all concurrent updates are acceptable, likely leading to inconsistencies, and the result is resolved afterward. The inconsistent condition between replicas is eventually re-establish via merges of differing replicas [4].

The desirable quality of using CRDT in the distribution system is to make the ordering less essential because of performing the replication as commutative operations. Providing an arbitrary order for the replication leads to reframing the race conditions in many distributed systems. In some collaboration software, such as Google Docs, and Figma, in which several users can concurrently make changes to the same file or data; and mobile applications that preserve users' data on the local device and then need to synch data to other devices belong to the same user, the benefit of unplanned order growth as the asynchrony increases.

There are two models of replications for CRDT as an eventually consistent distributed system. The first one is called **state-based replication**, it is also called **Convergence-Based CRDTs or CvRDTs**, which means that once a replica receives an update from a client, it first makes its local state up to date, and then later it sends its full state to other replicas, where the states are integrated by a function which must be commutative, associative, and idempotent. Gradually, all replicas send their full state to others in the system. Each replica that receives the state from another one, employs a merge function to combine its local state with the state that is obtained.

Another approach is **operation-based replication**, it is called **Commutative-Based CRDTs or CmRDTs**, indicates that the replica does not send its full state to others, which is big data. Instead, it transmits the update operation to the entire system and expects them to reply to that update. Since, the update is a broadcast operation, if two replicas received two updates in different orders, they can converge these updates as long as they are commutative. The crucial point is that the transmission infrastructure must ensure that all operations are provided to other replicas with no repeats in any order.

## 2.4 Distributed Storage

### 2.4.1 IPFS concept

IPFS [20] manages to keep and access data, websites, and applications in a distributed system via a peer-to-peer storage network. It stands for the InterPlanetary File System to create a system that works across places as disjoined. It generates a new platform for deploying the application and a new architecture for versioning large data.

It employs the concept of providing information based on its content, not its location. In the IPFS version, the computers use it to send requests to other nodes/computers across the world to share the content looking for with them. Meaning that the user can get their needed information from anyone who owns it, not only a specific source like Wikipedia. Moreover, via IPFS, the user not only accesses the files, but also their computers participate in the distribution phase. The possibility of downloading the content from many locations would benefit the users. If specific services or servers crash, not providing information, the data can still be accessible through other computers in IPFS, leading to resilient internet. Blocking the information or data sponsorship could be hard to happen because files can be accessed through various places in IPFS. Moreover, it can retrieve data from a computer nearby instead of thousands of miles away leading to a better experience to access data.

Generally, the process begins when the user adds a file to IPFS. Then it splits into smaller cryptographic hashed chunks with a provided unique fingerprint called CID. When other peers send the request for the content via asking for the content referenced by the file's CID, once they download or view the file, they become another provider of the file. Each node can either pin the content to keep it permanently or can discard it for the sake of memory. The crucial part is that, when a new version of files is generated, the hash is different meaning that the stored files are resistant to changes and censorship. To find the latest version of a file, the IPNS decentralized naming system, and DNSLink can be used to map CIDs to human-readable DNS names.

There are three crucial principles to understanding IPFS. Content Addressing, Directed acyclic graphs (DAGs) and distributed hash tables (DHTs).

#### Content Addressing

Content Addressing is one pillar to identifying the content based on what is in it rather than by where it is located. Instead of providing a unique address to get data which happens in location addressing on the centralized web. In location addressing with URL, the centralized web creates links and connected data on the web based on the location where data is stored not based on the contents of the resource stored there. The point is that the contents of hosted files on the centralized web have no immediate relation to their location-based addresses. Since the user has no chance to verify the resident content at that particular URL, it is straightforward to get tricked by malicious data.

Content addressing provides a decentralized web by using a particular form of content addressing as CID - Content Identifier. CID is just a label used to indicate the material in IPFS containing both a cryptographic hash and a codec, which holds information about



how to interpret that data. Codecs encode and decode data in certain formats. CID does not point to where the content is kept, however, it forms a kind of address based on the content itself according to the cryptographic hash. It means that different CIDs will produce even by negligible changes in the content. Moreover, adding the same content to the different IPFS nodes by using the same settings will generate the same CID.

```

1  const node = await Ipfs.create();
2  const data = "Hello, My Name is Mostafa Tavassoli";
3  const result = node.add(data);
4  const cid = await result.then((res) => {
5    return res.cid;
6  });
7  // generated CID for the given data is
8  QmWXxBhZnHeBM4RxU7G6wgxvGCU5mqat8ZqmzuZPLHuwg

```

```

1  const stream = node.cat(cid);
2  const decoder = new TextDecoder();
3  let decodedData = "";
4  for await (const chunk of stream) {
5    (decodedData += decoder.decode(chunk)), { stream: true };
6  }
7  // The outcome of the generated CID is
8  Hello, My Name is Mostafa Tavassoli

```

Cryptographic hashing is a crucial tool in decentralized data structures forming the content addressing. The output of hashing is the single, fixed-size hash that represents the input data in any size and type. It is a string of characters that refers to data as a unique name. Since hash stems from the content of the data itself, using the same algorithm on the same data will generate the same hash. By default, it uses an SHA-256 algorithm that also supports other algorithms. CID is generated in different forms and versions. Many of the current IPFS tools produce CID v0, however, the files and object operations use CIDv1. Some several formats and protocols employ content addressing, tools like Git, and protocols like Ethereum and Bitcoin. CID has a specific structure with the following fields and relies on self-describing data representation protocols.

- **Multibase prefix**, represents the supported base encodings that the binary CID has been encoded.
- **CID-Version identifier**, shows the CID version.
- **Multicodec identifier**, indicates the way addressed data is encoded.
- **Multihash**, shows the hash-digest addressed data.

The generated hash by cryptographic algorithms needs to have some features like deterministic, uncorrelated, one-way, and unique. However, there might be a case that an algorithm does not comply with these characteristics leading to insecurity and others may not be sufficient enough for content addressing in IPFS. Hence, the multihash format needs to be used to realize which algorithm was employed for hash creation. Multihash format has both the length and algorithm to generate the hash. It follows the TLV (type-length-value) approach. Type indicates the algorithm uses to generate the hash, the next one defines the actual length of the hash, and the last one is the actual hash

value. To represent the CID as a compact string instead of a string of 0 and 1, it used base58 encoding to generate CID. This approach brings the first version known as CIDv0 which starts by Qm. This version uses as a default for several IPFS operations.

1

QmWXxBhZnHeBM4RxU7G6wgxvGCU5mqat8ZqmzuZPLHuwwg

The second version contains identifiers to detect the representation used with the content-hash per-sei. The identifiers include the multi-base prefix to identify the encoding for the reminder CID. The next one is related to the CID version and the last one is a multicode identifier guiding how to interpret the fetched content.

When new content is added to IPFS, it divides into chunks, and each part has its own CID. As soon as all chunks have their own CID, then Merkle Directed Acyclic Graph (DAG) of the file is constructed.

**Peer Addressing**, Each peer in the IPFS defines by its unique PeerID calculated through its public key as Multihash. The peerID employs for verification to secure the channel when communication establishes. IPFS relies on the Multiaddresses concept to detect the locations of remote peers. This kind of addressing allows the inclusion of multiple protocols and address types which is a self-describing, easy-to-understand, and hierarchically-separated sequence of protocol selects. It has a particular structure, starting from the Network layer, protocol, and address, then, in the transport layer identifies the protocol and port followed by the protocol to address specific peers as p2p and the PeerID. The Multiaddresses provide the nodes with the benefit of checking the connection to the remote peer before attempting it. Moreover, it allows for the extension of the intermediate of communication via prefixing peer addresses.

**file pinning** is a mechanism telling IPFS to store a given Object in a third-party remote pinning service if used in addition to keeping it in the local node as default. IPFS has a caching mechanism that will keep the file for a short time after performing IPFS operations, however, these sorts of objects may be collected via a garbage collector. This can be prevented by pinning the CID or adding it to MFS. The objects added with the ipfs add command, by default are pinned recursively, on the other hand, the Objects in MFS are protected from Garbage collection since they are not pinned.

## Directed acyclic graphs (DAGs)

IPFS takes the benefit of using a data structure called DAG. In particular, they use Merkle DAGs, where each node has a unique identifier which is the hash of the node content. It refers to the CID concept, which means detecting a data object (like a DAG node) by the value of its hash for addressing the content. Merkle DAGs follow the concept of self-verified structures meaning that the CID of the node is unambiguously connected to the content of its payload and its descendants. Since two nodes with the same CID represent the same DAG, provides an essential opportunity to effectively synchronize with the Merkle-CRDTs (Conflict-free Replicated Data Types) with no copy of the full DAG. Control versioning systems like git and others use the concept of DAG to keep the history of the repository to activate object duplication and conflict detections across branches.

To construct the Merkle DAGs of the content, IPFS breaks it into blocks. By doing this, different parts of the file come from different sources and are authenticated fast.

Moreover, if there are two similar contents, they can share parts of different Merkle DAGs and can reference the same subset of data, only updating the files with new content addresses. It leads to a decrease in transferring the large version of datasets efficiently since the only updated parts are new instead of creating new files each time.

### Distributed hash tables (DHTs)

IPFS uses the concept of DHTs to discover the peers holding the required content. It is a database of keys to values that is split across all the peers in a distributed network. To have the DHT mechanism, libp2p handles peers to connect in IPFS ecosystems. After realizing where the content is, the system employs libp2p to send the query to the DHT to find the current location of peers. After discovering the content and its current location of it, the process of connection and exchange of that content starts. IPFS currently employs a module called Bitwap to request blocks from and send blocks to the peers that own the needed content. It takes the responsibility to connect to the peers that the users require with the lists of blocks as wantlists and send the response back. Moreover, the user can verify the received lists by hashing their content to get the CIDs and make a comparison with the requested content.

Bitwap is the essential module in IPFS for exchanging blocks of data. Bitwap is based on the message protocol that all messages contain want lists or blocks. This protocol acquires the requested blocks from the network by the client and sends them back to peers who want them. IPFS divides the files into chunks of data known as blocks which are identified based on the CIDs. When a node requests to fetch a file by running the Bitwaps protocol, they send out the want lists to the other nodes. Want-list is a collection or lists of CIDs for blocks a peer wants to receive.

```

1 want-list {
2   QmZtmD2qt6fJot32nabSP3CUjicnypEBz7bHVDhPQt9aAy, WANT,
3   QmTudJSaoKxtbEnTddJ9vh8hbN84ZLVvD5pNpUaSbxwGoa, WANT,
4 }

```

Finding a peer to have the file the user needs starts by sending a request called want-have to all connected peers. This request includes the CID of the root block of the file. The root block is the top of the DAG of the block. Those peers have that block send the have response to the requested node, otherwise, they send the dont-have response. If no peers have the root block, Biswap sends the query to DHT to ask who has the proper data [Figure 2.5].

### Libp2p

It enables the development of peer-to-peer network applications by using a modular system of protocols, specifications, and libraries [28]. A P2P is a network in that peers communicate together directly as equal participants in comparison to the client/server approach in which a privileged central server might provide services to many client programs on the network. This module provides a consistent way of addressing schemes by encoding multiple layers of addressing information into a single future-proof path structure.

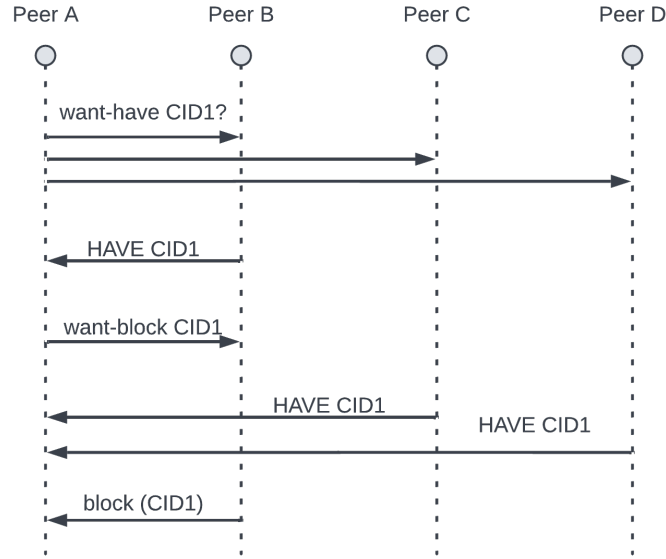


Figure 2.5: Discovery sequence

This approach shows the usage protocol, and address along with protocol and port, like

```
1 /ipv4/171.113.242.172/udp/162
```

In addition, the way of moving data from machine to machine alters by defining two core operations, listening and dialing. The former means that the node can accept the incoming connections from other peers, and the latter is the process of opening an ongoing connection to a listening peer. The transport layer takes responsibility for sending and acquiring bytes between two peers. Libp2p employs public key cryptography as a key part to provide communication security and reliability.

## 2.4.2 etcd

etcd works based on the Raft algorithm to guarantee data consistency across all nodes in the cluster. It sacrifices speed for greater reliability and strong consistency. According to the mentioned algorithm for Raft, it takes its consistency based on the election system. It works based on the distributed consensus according to a quorum model; as the Raft algorithm does. This means the majority of members have to agree upon a proposal before committing to the cluster.

etcd considers an operation as completed via the consensus obligation, then is stored by the storage. etcd also guarantees atomicity and durability. The first means that the operation either finished totally or not at all. Then once the operations are completed, it considers durable. etcd brings the benefits of fully replicated, which means all nodes in the etcd cluster have access to the entire data store. It provides consistent data,

with every node in the cluster reading data and returning the most recent data written. Moreover, it is highly available means that there is no single point of failure.

It is a famous, distributed, and consistent key-value storage. etcd is the main element in Kubernetes datastore and a crucial standard system for container orchestration. It plays a role when an application needs to distribute the provision of data redundancy and resilience across the node's configuration through reading data from and writing to it.

etcd holds data related to both configuration and state for Kubernetes. It employs a "watch" function to realize the inconsistency among data to tell Kubernetes to re-configuration the cluster accordingly. It stores data in a multi-version persistent key-value store. According to Wikipedia, "persistent data structure or not ephemeral data structure is a data structure that always preserves the previous version of itself when it is modified. [40]" In this case, we have two concepts. One defines if all versions are accessible, but the newest version can be modified called partially persistent, and the second one is called fully persistent if every version can be accessed and modified.

etcd brings the following benefits. It is fully replicated, which means all nodes in the etcd cluster have access to the entire data store. It provides consistent data, with every node in the cluster reading data and returning the most recent data written. etcd is based on the Raft algorithm. Moreover, it is highly available means that there is no single point of failure. It is designed to be straightforward to use and to integrate into distributed systems. It provides a simple API for storing and retrieving data, and it can be accessed through a command-line interface or a variety of programming languages.

The performance of etcd depends on two criteria, latency and throughput. The former means the time taken to complete an operation and the latter is related to the total operations finished in some period. If etcd accepts simultaneous client demands, the average latency grows as the general throughput increases. As discussed, the etcd utilizes the Raft consensus algorithm to replicate requests across members and reach an agreement. In this case, the consensus performance depends on network IO latency and disk IO latency as two physical constraints. Round Trip Time - RTT - between the peers is the minimum time to finish the etcd request [12].

### 2.4.3 Redis

Redis Enterprise supports CRDT [35] as an in-memory data store that acts as a database, cache, and message broker. It has better performance in read/write operations in comparison with etcd since in etcd each node has access to the full data store by employing the strong consistency approach. Because it needs coordination with other nodes to apply modifications. Strong consistency does not support by the Redis cluster, meaning that, the possibility of losing write in specific circumstances even after acceptance by the system to the client. CRDT-based databases like Redis enterprise, are accessible even in a time of network problems when there is no data exchange among distributed database replicas. In this case, they provide the local latency to the operation of read and write.

## 2.4.4 OrbitDB

OrbitDB [32] is a serverless and distributed database utilizing IPFS -Interplanetary File System- protocol for storing data and sharing the information across a distributed file system for building p2p decentralized applications, meaning that it does not rely on a central server or authority to manage the data. Instead, it uses a peer-to-peer network to store and replicate data across multiple nodes. It manages and creates mutable databases based on the IPFS get and set functions. Orbitdb is a consistent database that uses CRDT for a conflict-free approach. CRDTs are logs with the special format of "clock" values allowing several users to use disconnected and asynchronous functions on the same distributed database. By sharing the clock values, it guarantees that there is no vagueness about how their disparate entries will be put together. There are several types of databases for different data models implemented by OrbitDB. All databases are designed on top of the ipfs-log which is an immutable operation-based CRDT for distributed systems. This database is written in JavaScript because of its popularity in the programming community, and JavaScript implementation that is called js-ipfs.

**ipfs-log**, it is an immutable, operation-based CRDT in a distributed system to form a module, append-only log, and shared state across the peers in p2p applications. Each entry in the log saves in IPFS and each of them points to a hash of the previous entries forming a graph. The functionality given by ipfs-log is the design of a CRDT utilizing IPFS's built-in DAG to connect data in a specific manner. It is the entries based on the JSON objects that follow a certain schema to generate linked lists or chains in a DAG.

Files in IPFS are based on content-addressed that is achievable through a technique called hashing, despite the content that is location-addressed on the internet. Normally, the request sends to the DNS to determine which servers out of millions process the query and return the response. On the other hand, when content adds to IPFS, the content is given an address based on what it is, not based on its location. Multiple servers would reply to the users' requests that asked by its hash at the same time if they have the data. For hashing, there are two standards, CIDv0 and CIDv1.

```
1 hashes: QmWpvK4bYR7k9b1feM48fskt2XsZfMaPfNnFxdbhJHw7QJ
```

Listing 2.1: CIDv0

```
1 hashes: zdpuAmRtbL62Yt5w3H6rpm8PoMZFoQuqLgxMsDJR5frJGxKJ
```

Listing 2.2: CIDv1

This kind of hashes called a multihash protocol meaning that they have self-describing prefixes for differentiating outputs from various well-established cryptographic hash functions, addressing size plus encoding considerations. Performance and verifiability are the main reasons behind selecting the content addressing instead of location one. The performance achieves because files can be downloaded from multiple peers like the BitTorrent communication protocol. In addition, the users can access the requested data because the hashes are verifiable.

## Concept of Peer-to-Peer

In distributed applications, it is a requirement to have a deep understanding of the p2p concept to realize their interaction as a swarm rather than a client-server approach. In the traditional model, the servers store and manage the user data which sends by the clients that connect to the servers. It has provided some issues like data management, ownership, and security.

**Swarm and Servers**, In the p2p model, there is no need to deal with the concept of center and server in the network, and the users store data locally and quickly connect to other peers in the identical network to form a swarm of related users. There will be no clients if there are no servers. Once the user runs the p2p applications, they run the entire application in their local system. In this scenario, the power stems from the peers' connection and interaction because of the same functionality the peers have across all application instances.

In the **Swarm approach**, the applications can operate offline by putting data locally and safely. On the other hand, the peers in the network are untrusted since they can send the request for the data by using the content address. Moreover, the user might not access data if the peer keeps data offline.

**pubsub**, the way of peer communication, moreover, is different in the p2p application, because there will be a situation where a specific client might not be connected at the time of messaging. Traditionally, the messages send through REST or SAOP, instead in a p2p swarm, they use the pubsub - publish and subscribe - approach to form the communication. In this model, the publishers act as message senders and categorized the published messages into topics, and subscribers show their interest in them and will get only those. All operations happen without direct communication between publishers and subscribers. This approach provides the benefits of network flexibility and scalability.

**Directed Acyclic Graphs (DAGs)**, the concept of the graph means the representation of objects' connection in a mathematical abstraction. In this terminology, the node refers to an object, and the edge points to a relation among them. Normally, graphs use to present the paired relationships between objects. If there is a sense of direction at each edge of the graph, it is called a directed graph. A graph with no loops is called acyclic meaning that there is no way to navigate from a node to itself through the graph's edges.

**Merkle DAGs**, it is necessary to encode the representation of the graph's nodes and edges to illustrate them on the computer. In this model, CID which is a unique identifier can employ to show an edge from one node to another. Merkle DAG is in itself a DAG where each node has an identifier as a result of hashing the nodes' content. The construction of DAG must generate from the leaves then the parent will be added after the children since their identifiers need to be calculated in advance.

### 2.4.5 etcd vs orbitDB

In this section, I provided a simple comparison between the two tools for storing key-value pairs. The etcd uses the RAFT algorithm that provides strong consistency in GET linearizability and PUT operations. However, the orbitDB provides eventual consistency

because of using CRDT. In terms of operations, orbitDB has PUT and DELETE operations that needed time to append operations to the operation log for creating, hashing a new object, and writing the data to disk, which are time-consuming. However, its GET operation is fast because synchronously retrieves a value that is already kept in memory. On the other hand, etcd's operations, GET includes two levels of consistencies, one is Linearizability which requires coordination of a quorum of replicas, meaning more time for consensus between the members. It provides another method with less overhead Serializability, that has lower latencies and higher throughput.

<b>etcd</b>	<b>orbitDB</b>
uses RAFT algorithm	uses CRDT
ensures Linearizability and Serializability	uses IPFS protocol for storing and accessing files
provides Strong Consistency	provides Eventual Cnsistency
is a key-value store	is a data structure

Table 2.1: etcd vs orbitDB



## Chapter 3

# Literature Review

This section is designed for reviewing the migration concept and the criteria that impact it in cloud and edge environments in other studies.

### 3.1 Migration Concept and Performance

Cloud data centers have become an essential part of the service providers. By using the edge computing paradigm, the benefits of offering computation, intelligence, and networking management with low end-to-end latency would be accessible by pushing the workloads of computations and networking capabilities to the edge from the clouds [15].

It is essential to guarantee the quality of service - QoS - in the dynamic environment via the concept of service migration among different computing nodes on the edge. Several studies have proposed dynamic resource management, some of them investigated for enhancing the single live migration of VM and container, and others conducted migration of service in the edge-centric computing paradigm. The live migration process is the main component covering dynamic resource management in both edge and cloud computing environments. Live migration can provide the generic with no application-specific management and configuration.

Moving from the cloud to edge environment, the operation of resource processing and intelligence need to transfer from the cloud to the edge of the network to enhance the service time with higher bandwidth and lower latency [18], [38]. In different paradigms, live migration could perform between servers in the edge environment, physical hosts in the LAN network, or different data centers in the WAN environment [14]. The user position and coverage of each edge server depends on its based station leading to edge computing migration. When the user position alters, the latency of end-to-end would be suffered. In such a scenario, the service might need to be transferred from the previous edge servers to the adjacent one.

The performance of the live migration heavily relies on the network resource, bandwidth, and delay once the state transmission happens. The provision of data transmission for both live migration and service connectivity needs to be established through the edge and the cloud data center network. To address the issues of managing and configuring the

scale of resources which stems from the complexity of edge and cloud computing's expansion, Software-Defined Networking (SDN) is defined as offering centralized networking management through dividing data and control plane in conventional network services. Therefore, it becomes the emerging solution in edge and cloud computing since it can dynamically update the entire network topology.

To manage the migration, it is crucial to reduce the costs of migration, improve its performance via gaining the goals of dynamic resource management. Since, managing the resource dynamically requires several instance migrations, this study emphasizes the management of migration in the context of multiple migrations solution.

Although many studies have been focused on specific migration aspects and neglect other concepts like migration generation in resource policies, migration planning and scheduling algorithms. This study [15] identifies five aspects of migration management in both edge and Cloud computing environments,

- migration performance and cost model
- resource management policy and migration generation
- migration planning and scheduling
- scheduling lifecycle management and orchestration
- evaluation method and platforms

It describes representative activities of dynamic resource management focusing on the migration generation in migration computing and networking parameters, and migration objectives.

Nowadays, services are provided typically by VMs and recently the containers gain the power as lightweight from VMs. One explanation for container migration is to preserve immediacy between edge computing services and mobile users. It is vital to mention that migrating containers have typically ongoing communication with other endpoints like user applications. Moreover, the state and connection which is shared by communicating endpoints in connection-oriented protocols need to be migrated. VM and container are the most commonest way for creating the instances for live migrations. Container runtime is software for creating instances on a computer node, the most famous one is Docker. CRIU is the basis standard for live container migration. CRIU generates checkpointing files for each connected child process during a process three checkpointing [15]. Migration can be applied into various types like process, VM, generally refer to disk less migration. Instance and storage migration can be categorized as cold, non-live migration and live migration. The live migration can be divided into pre-copy, post-copy and hybrid migration. The continuous optimization and improvement of live migration aspects are to decreasing downtime and live migration time [15]. In this case, downtime is the time interval in which the migration service is not available due to synchronization. Migration time, refers to the time interval between starting the pre-migration phase to the finish of the post-migration step that the instance is running at the destination host. Finally, total migration time, is a time interval between the starting of the first migration and the completion of the last migration.

For the performance balance analysis, memory and storage transmission can be categorized into 3 phases. **Push phase**, where the instance is still running in the src, while mem pages and disk block or writing block are pushed through the network to the destination host. **Stop-and-copy**, where the instance is stopped, the memory pages are copied to the destination, at the end, the instance will resume at the destination. **Pull phase**, where the instance executes while pulling faulted memory pages from the source host across the network. Based on the mentioned phases, it can be divided into 3 types,

- Pre-copy focusing on push phase.
- Post-copy using pull phase.
- Hybrid.

Pre-copy migration, during this migration, dirty memory pages are iteratively replicated from the running instance at the source host to the destination instance container. Post-Copy Migration, firstly, it suspends the instance at the source, then resumes it at the target by migrating the tiniest subset of VM implementation states. Then the source pushes the remained pages to the resumed VM. It can decline the time of live migration and downtime in comparison to pre-copy. When the running instance crashes in post-copy migration, the service will be crashed as there is no running one in the original host.

The parameters and metrics play an essential role in this section to realize the migration performance and cost model. The parameters are divided into three categories, computing (CPU load and utilization, memory load and consumption, memory size), networking (Migrating routing, available bandwidth, migrating distance), and storage resources (the number of involved network layers, network delay (link delay and end-to-end latency)). In terms of Metrics, this study expands the study of impacting the single migration on performance to multiple migrations. In this section, the metrics are categorized into time-related, QoS, energy, and SLA. [15]

TCP as a connection-oriented protocol can not persist in the migrating environment because of changes in the IP address or Port number, however, QUIC provides a mechanism for a client-side connection migration. In QUIC, when the IP address changes, it migrates to a new address. However, the server-side connection migration in this protocol has not yet been designed. This study extends this protocol to cover the server-side connection when a specific container migrates across hosts. Two different approaches were designed to satisfy these requirements [3].

In the cloud computing environment users normally access the provided services in the form of VMs. In this scenario, the virtualization activates the multi-tenancy, elasticity, and flexibility. One example of this flexibility is the migration of a VM between servers. VM migration in the cloud proves useful, for server maintenance and load balancing purposes. These days, the VMs have been substituted with containers because it shares the kernel with the host system, however, the VMs have their kernel. As a result, it brings the features of the lightweight container, fast booting process, and migration quickly [37].

Edge computing is an extension of cloud computing at the edge of the network. The servers of edge are typically deployed at a one-hop distance from the end-users. This

proximity activates a wide range of applications, demanding resource-intensive computation and low latency with high bandwidth. The concept of migration applies to containers which bring the benefits like end-users mobility, apart from load-balancing and container migration across edge servers [36].

## 3.2 Stateful Migration Techniques

In stateless migration, the original container deletes from the source server, and the new one creates at the final destination leading to losing the state of the container. On the other hand, Stateful container migration is a condition when it transfers the state (memory pages, the state of CPU, registers, disk content) of the container from a source to the destination server. The following techniques are the most popular for stateful migration.

- **Cold migration**, first stops the container to make sure that the state is no longer altered. Then, checkpoints the state and moves it once the container is not operating anymore. Ultimately, it resumes the container execution at the destination. All in all, this technique causes long container downtime.
- **Pre-copy migration**, in the first phase, the pre-copy (pre-dump) migration checkpoints and transmits the state of the container while it is running at the source. In the next step (dump phase), the container stops and transfers only the differences to the state that captured during pre-dump stage. In the final step, it resumes the container execution at the destination. This strategy provides a short time than the cold one since only the modifications to the state are transferred during the time interval.
- **Post-copy migration**, at the first, suspends the execution of the container at the source, then copies the tiny portion of the state like the CPU state, and registers to the target, leading to resume of the container execution. The remainder of the state transfers to the destination in the background while the container is running at the destination. The downtime in this technique is too short.
- **Hybrid migration** is the mixture of the previous mechanisms. The pre-dump phase is the same as for the pre-copy, then the container stops and only modifications copy to the destination. Gradually, the container resumes its execution at the destination, once it is still running. All in all, this approach provides the shortest downtime, but has the drawbacks of both pre and post copy techniques.

It is crucial to mention that the connectivity holding meaning the communication after migration, like external client applications need to maintain seamlessly [3].

SDN, Software Defined Networking, offers an opportunity to redirect the network traffic after VM migration to a new host [30]. Moreover, the approach as CRIU investigated to implement the container migration by means of `-tcp-established` allowing active TCP container connections [5]. In this scenario, similar IP address should be kept since the TCP can not tolerate this situation.

The alternative to TCP is QUIC. The objective of this study is to extend QUIC for supporting connection migration in a stateful way to a destination server. It provides some benefits, one is that when the client’s IP address changes, the migration happens seamlessly to the new address. It also supports the native client-side connection migration. However, the migration related to the server side in QUIC is not implemented [3].

**QUIC**, as a connection-oriented transport protocol, it works over UDP but provides reliable communications through designing the mechanisms like flow control, congestion control, and loss detection. This protocol surpasses TCP. QUIC implements in the user stack while TCP runs in the kernel space, leading to upgrading the OS when changes push to TCP stacks. Moreover, QUIC keeps away from the issue of head-of-line in TCP. The streaming traffic in QUIC connection could be handled independently from one another means that the loss of packets of one stream has no impact on other streams. Finally, TCP is not secure in its nature and requires TLS to run over it as an additional protocol. However, QUIC is an encrypted one by default that already includes TLS 1.3 handshake. TCP with TLS on top of it needs three Round Trip Times, but QUIC only needs one RTT to establish the connection to an unknown server, on the other hand, the connection for the known server is done only via zero RTT.

The way of these two protocols manage the connections is another significant difference that is related to this study. TCP has a unique identifier that is the combination of some fields like <src IP, src Port, destination IP, and Port>. TCP works properly as long as no changes happen in these elements. Otherwise, the connection is closed and the new one is established. QUIC has a set of connection identifiers that are independent of the network configuration like IP address and Port [3].

In Software-Defined Networking (SDN)-activated cloud data centers, live migration plays an essential role in VMs transmission in the cloud services and VNFs in Service Function Chaining (SFC). Software-defined networking (SDN) is an approach to network virtualization and containerization that helps optimizing network resources and quickly adapt networks to changing business needs, applications, and traffic. It works by separating the network’s control and data planes to create a software-programmable infrastructure. Service Function Chaining (SFC) refers to the use of Software-Defined Networking (SDN) programmability to create a service chain of connected (virtual) network services. One advantage of using SFC is to automate the way virtual network connections can be set up to handle different types of traffic flows.

Cloud providers mitigate the issue of dynamic resource management and fault tolerance with less disturbing users’ service through live migration. It is crucial to have efficient migration planning to decline the effect of live migration overheads when operating multiple live migrations in arbitrary order leading to service degradation in the cloud centers. Moreover, set priorities would be necessary for different live migration requests to prevent QoS degradation and SLA violations. The outcome indicates that selecting the proper algorithm leads to a decline in the number of deadline misses and has a good migration performance in terms of total migration time, downtime, and transfer of data [16].

Because of computing and network resource limitations and migration overhead, migration scheduling is an essential operation in the data centers to obtain the migration

performance with considering preventing SLO violations during the scheduling operation. This paper [16] proposes an algorithm including concurrent migration grouping and an online migration scheduler. Instead of direct grouping migration, the proposed algorithm (SLAMIG) can optimize the order of concurrent migration groups via sorting each migration based on complete dependency subgraphs. The online migration scheduler can guarantee the order of scheduling different migrations and concurrency in a dynamic network environment. It is also crucial to argue that in addition to total migration time, optimizing the average execution time, transferred data, and downtime are essential factors to assess the multiple migration performance.

The time requirements like migration deadlines and SLO violations are considered as total migration time, while the actual time is related to the integration of execution time, transferred data, and service downtime. The SLA and dynamic performance needs of cloud services can be supported by total migration time optimization and QoS of services and achieving suitable revenue as the cloud providers could be guaranteed through optimizing the sum of execution time, transferred data, and downtime. The result of the SLAMIG experimental depicts the reducing the number of migration deadlines missing and good migration performance in total migration time, average execution time, and downtime with acceptable algorithm runtime.

The QoS is the one aspect that needs to be guaranteed for cloud services like Web, big data, augment reality because of increasing the adoption of cloud computing. It is crucial to preserve QoS in heterogeneous environments and to prevent the violations of SLA for offering the cloud administrators. Hence, many studies have focused on the quality, accessibility, and robustness of cloud services. In [13], the latency of SFC studies to optimize it for providing the benefits for both network service providers and end users.

### 3.3 Virtualization Concept

Virtual Machine - VM - is one virtualization technology to provide computing and network services in the cloud data centers. To transfer VMs across physical machines without disrupting services accessibility can be achieved through Live VM migration [2]. Therefore, a dynamic resource management like live VM migration is required to manage several resource rescheduling of data centers like load balancing, host overbooking, fault tolerance, or relocating of VNF because of the user location changing [7].

Study [1], proposes a way to speed up the application startup by getting a snapshot of the fully-deployed containers' state and restarting the future instance of the container from the pre-started application state. Container deployment is a routine operation that is placed in a path where the service is being delivered to the end users. Even though the process of creating the container is fast; however, the application related to the containers requires starting before the container starts producing beneficial output. Since a similar container has to be launched, constructed, and booted repeatedly, it has a crucial impact on the FOG computing environment. The booting process of all containers remains the same. This can provide an opportunity to save the state of a fully-booted container leading to eliminating the boot phase. Its result shows the efficient startups between 1x and 60x through examining 14 microservice containers.

The idea is to checkpoint the state of the container after it has started and starts every subsequent instance of the container by reloading the checkpoint to increase the velocity of the container's boot. The challenge is that the process of the checkpoint and restart does not save the full environment of the running application since the system assumes that the application will be restarted in the same environment where it was checkpointed. So, to address this issue, the entire environment of the container that was present during the operation of the checkpoint has to be captured. Another challenge relates to anticipate in which server within a PoP - Points-of-Presence - a container will be restarted, in this case, it is crucial to share all saved checkpoints across all the servers of PoPs.

This paper [39] proposes the architecture called ContMEC to enable the UEs (User Equipment) to delegate a part of applications to MEC (Multi-access Edge Computing) servers. According to Juniper [23], "Multi-Access Edge Computing (MEC) moves the computing of traffic and services from a centralized cloud to the edge of the network and closer to the customer. Instead of sending all data to a cloud for processing, the network edge analyzes, processes, and stores the data. Collecting and processing data closer to the customer reduces latency and brings real-time performance to high-bandwidth applications." A MEC architecture consists of a small number of centralized data centers and edge stations that distributes geographically in which of each multiple edge servers are installed. The application inside on UE implements like the cloud-native applications and they deploy on computing clusters. ContMEC brings the following features,

- generate the computing cluster for each edge station to have scalability of the number of UEs.
- managing the resources hierarchy for scalability and beneficial resource sharing among computing clusters.
- efficient resource sharing through overlapping the computer clusters.

The PoC (Proof of Concept) shows the benefit of offloading is higher than disadvantage of implementing applications as container clusters, the traffic control is moderate against the number of UEs, and efficient resource sharing among computing clusters is obtained [39].

In the current study, the performance of migration calculates based on various metrics and parameters. By setting delays and bandwidth on the communication of the services/servers trying to simulate the real-world scenario to measure the migration time. Some tested parameters are CPU utilization, resource constraints, and data size. Since selecting the ideal algorithm can affect the migration performance, in the current study, two essential algorithms for database data consistency are experimented with by several parameters. The etcd and OrbitDB are the main objectives under measurement. Various operations like GET, PUT, and DELETE are compared in these two concepts. Moreover, in terms of etcd, some factors like role change, data defragmentation, and compaction are measured at the time of service disruption.





## Chapter 4

# Experimental Setup

In the beginning, this chapter provides the methodology that we used to do the entire experiment. Moreover, we propose a solution to solve the issue related to stateless migration while keeping the state. After that, the instruction provided for configuring the environment for doing the experiments.

### 4.1 Methodology

To do the experiment in this study, I use CrownLabs [6] which provides by the university with the following characteristics. I use a virtual machine with Linux Ubuntu 20.04.5 LTS(x86-64) with a 2cores CPU and 4 GB RAM. In the most of my tests, I have three nodes for the k8s cluster. The three nodes are sufficient to experiment since I need to have one Master node and two worker nodes for the migration scenario.

For the etcd cluster, in the minimum case, I did the test with one member, and in the maximum case, created 6 members with etcd/bitnami Helm Charts [33]. The reason that we used different numbers is to meet the needs for several stateless migration approaches we need to experiment with. We need more than two members to measure the consensus overhead, otherwise, it is not relevant to measure it with only one member.

I calculated the bandwidth with the iperf3 tool as 8.56 Gbits/s between two nodes in the cluster. iperf3 [19] is a popular tool to measure network performance for bandwidth and latency. This tool employs to experiment with the maximum possible bandwidth on IP networks. To measure, I installed this tool in both nodes in my cluster and start one as a server and another one as a client. Then from the client, I set the traffic to the server. Then this tool calculates the bandwidth.

Moreover, the latency information is also essential in my experiments, which I calculated via ping between peers within the cluster, and got the result as "RTT min/avg/max 0.396/0.589/0.740 ms". The Round Trip Time(RTT) and bandwidth are essential to measuring because, in various scenarios, we need to set a different delays for having a real experiment as we experience in the real world. Moreover, they affect the outcome of our experiment. Because we need to understand the delays, since the consensus algorithm takes multiple RTTs, to realize the effect of the latency in the network.

In this case, to set the communication delays between nodes and members, the tc

tool [31] is used providing the proper features to do this throughout the cluster. It is used to establish network delays providing a flexible and efficient way to manage the behavior of the Linux network. We set 50ms latency in all communications between etcd members in one scenario. However, to experiment in the more close situation to reality, we also set different values for latency to do the real experiment.

CrownLabs Test Environment	
CPU cores	2
Physical Memory	4GB
k8s Cluster	one master node, two up to four worker nodes
etcd Cluster	one up to six etcd peers

Table 4.1: characteristics of Test Environment for CrownLabs

**Testbed Setup** Figure 4.1 depicts the architecture in general that we have to do further realistic scenarios in migration approaches. Based on the needs we can have various numbers of pods inside each worker node. The graph shows the delays of 10ms in the communication among workers and the 1ms inside the pods' communication. We did all these delays in their communication to generate a real-world networking scenario.

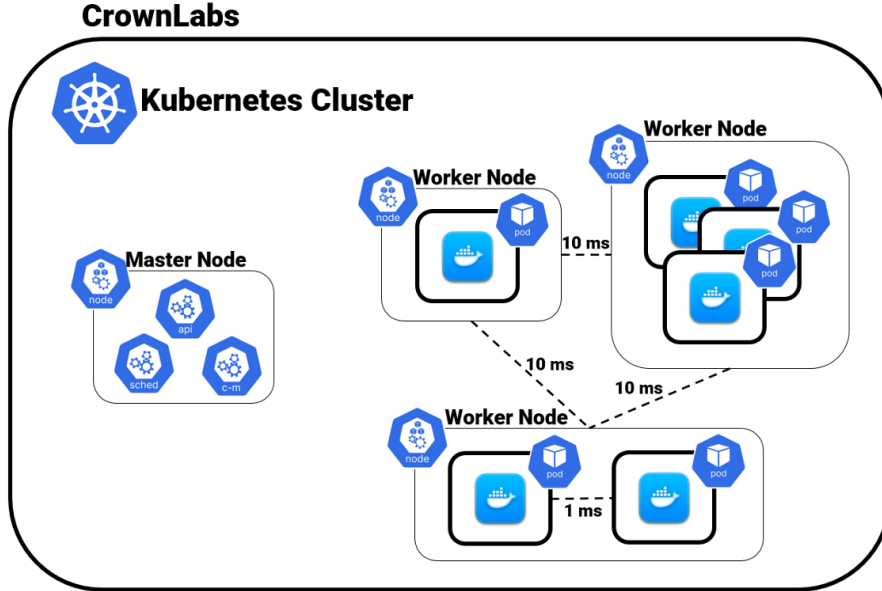


Figure 4.1: Testbed Set

**Stateless Migration Approaches,** In the concept of stateless migration, three main approaches can be defined to detach the microservices state. The objective of this section is to test the different scenarios of stateless migration, Centralized, Reactive distributed, and Proactive distributed to evaluate the etcd performance via measuring the duration of various operations, Put, Get (both Linearizability, Serializability), Role Leader Change, Revisions Compaction, and Database Fragmentation.

In the first scenario, **Centralized Approach** [Figure 4.2], there is a source and destination host, and the etcd is centralized on the database host. When migration starts, since the state is centralized, there is no need to migrate it. In this case, just the microservice moves from source to destination, without caring about its state. In this case, no consensus needs because only one instance exists. Migration time can be calculated based on running the microservice container on the DST host. Since etcd is located in another host to access the state, the network conditions can affect this access because every time each host sends the requests of R/W needs to go through the network to reach the etcd host. Therefore, it is considered a negative point in this scenario. Gradually, since the etcd is centralized, the network IO and the factor related to the disk can lead to performance degradation in this scenario.

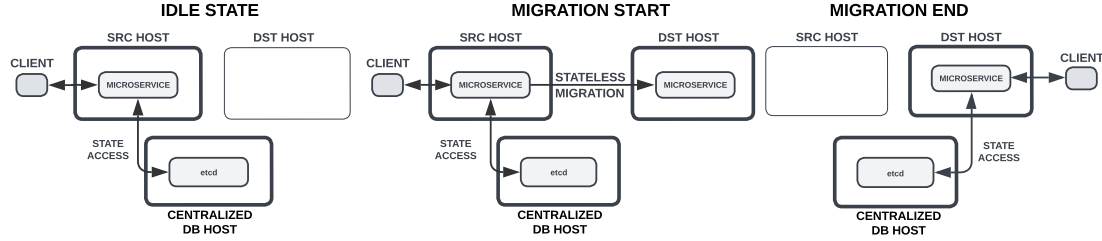


Figure 4.2: Centralized Approach

In the second scenario, **Reactive Distributed Approach** [Figure 4.3], both source, and destination will have the instance of etcd, so there is no centralized etcd. Therefore, when migration starts, there will be one instance of etcd in the destination host that has to replicate. Once it is replicated, migration of microservice can start and both microservice and etcd are down on the source host. In this case, access to the etcd instance is more efficient than the centralized approach concerning state access, however, the state needs to transfer from source to destination and it requires time for migration. In this scenario, the RAFT algorithm performs the initialization step for the new etcd instance, so there is no specific consensus because there is only one etcd instance. The only time, there are two instances of etcd is when the second one is created in the destination host to transfer the state. As a result, the factors in this approach define the performance of migration, related to microservice migration time including boot-up and warm-up time, etcd state transfer time, and etcd container boot up time.

Microservice boot-up time refers to the time a microservice is available and ready to process the request. The amount of time requires for a microservice to start fully operating after the deployment process. This time involves the operating system initialization, loading all libraries and dependencies, and all operations needed to get the services up

and running. On the other hand, the warm-up time refers to loading necessary data into memory, and connection establishment to other services. This time can be affected via several criteria like the service complexity, the number of resources required to load, and service infrastructure and composition.

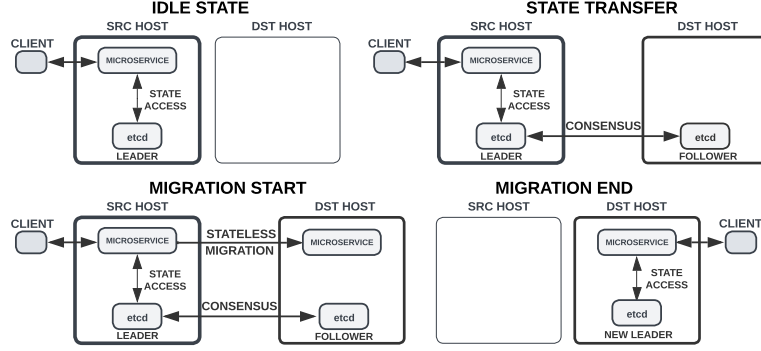


Figure 4.3: Reactive Distributed Approach

The last one related to **Proactive Distributed Approach** [Figure 4.4], discusses a situation when there are several destination hosts, like a mobility scenario. In this case, all base stations can host the microservice since they have an etcd instance inside which they continuously synchronized and the state is always distributed across the entire cluster for consistency. Once migration starts, the proper destination for migration needs to be selected. The only operation is to start a new microservice in the host, and set the etcd as the new leader to push the updates to other etcd followers. In this scenario, the overhead of consensus is higher than in previous scenarios, the majority of instances are aware of consistency whenever the read, and write operations to the database start. However, there is no need to consider the etcd transfer state across the cluster since all hosts have the etcd instance that is continuously synchronized.

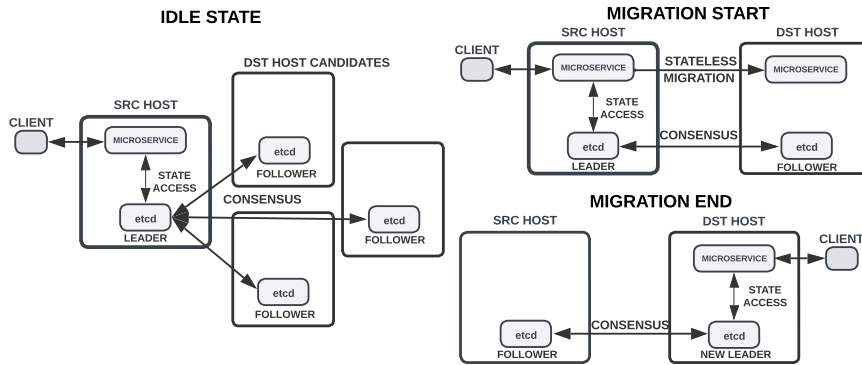


Figure 4.4: Proactive Distributed Approach

The next part is related to the configuration of k8s and installing related tools for doing all experiments for the rest of the study. The entire purpose of this part is to realize

step-by-step configuration with screenshots for each part. The last part is the important part discussing all results that we have got via doing several experiments with details.

## 4.2 Preliminary Experiments

### 4.2.1 Configuring the environment

To do experiments concerning measuring the migration time of pod from one node to another one in etcd, ipfs environment, to calculate the latency of put/get operations, a test environment needs to prepare. All tests experimented on within this section did in the Crownlabs with the following configurations. A Kubernetes cluster needs to create in this case. This cluster composes of a single control plane and two worker nodes. Some packages like kubeadm, kubectl, and kubelet have to install on all servers. To recognize the nodes in the cluster, the name of nodes both master and workers change to control-plane-1, worker-1, and worker-2 respectively.

```
1 sudo hostnamectl set-hostname <node_name>
```

**kubeadm** is a tool that facilitates the configuration of a Kubernetes cluster. It automates the steps involved in setting up a cluster, including creating the necessary infrastructure (such as etcd), installing and configuring the control plane components (such as the API server, etcd, and Kube-scheduler), and installing the necessary networking components (such as CNI). A CNI plugin has the responsibility of inserting a network interface into the container and creating any necessary changes on the host. It then assigns an IP address to the interface and arranges the routes consistent with the IP Address Management section by invoking the appropriate IP Address Management (IPAM) plugin.

To have a cluster with a single node, the following command can use,

```
1 # Initialize the cluster
2 sudo kubeadm init --pod-network-cidr=10.254.0.0/16 --service-cidr=10.255.0.0/16
```

Initially, this command runs a series of pre-flight checks to guarantee that the machine is ready to handle Kubernetes, then downloads and installs the cluster control plane components. This command will print a string to use in the worker nodes to join the cluster.

```
1 # Join the worker nodes to the cluster
2 kubeadm join <cluster-ip>:<port> --token <token> --discovery-token-ca-cert-hash <hash>
```

**kubectl** is a command-line tool for interacting with a Kubernetes cluster. It allows you to deploy applications, views the state of the cluster, and manage the cluster itself. It works by communicating with the Kubernetes API server. kubectl sends requests to the API server, then executes the demanded actions on the cluster. kubectl has a wide range of controls for handling different elements of a Kubernetes cluster. It has a specific syntax to use for communicating through the command line.

```
1 kubectl [command] [resource] [flags]
```

- **command**, establishes the operation want to perform on one or more resources, like create, get, describe, delete.
- **resource**, The resource users want to operate on, such as a pod or a deployment.
- **flags**: optional flags that modify the behavior of the command.

Some examples of kubectl commands and their syntax:

```
1 # get the status of nodes
2 kubectl get nodes
3
4 # get the status of pods with more information
5 kubectl get pod -o wide
6
7 # show information about the pod, ip address, images, container ids
8 kubectl describe pod <pod-name>
9
10 # create a deployment for a pod
11 kubectl run <container-name> --image=<image-name>
12
13 # Alternative way to create the pod with the apply command.
14 Kubectl apply -f <YAML file>
15
16 # get into the pod with the shell in an interactive mode.
17 Kubectl exec <pod-name> -it sh
```

Listing 4.1: kubectl commands examples

After joining the worker nodes into the cluster, the user can check the status of all nodes.

```
1 kubectl get nodes
```

**YAML** file which is human-readable data can be used to define the resources like deployments, services, and pods. It is a series of key-value pairs, with the keys denoted by indentation and the values specified on the same line as the key.

```
1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   name: nginx-deployment
5   labels:
6     app: nginx-deployment
7 spec:
8   replicas: 1
9   selector:
10    matchLabels:
11      app: nginx-deployment
12 template:
13   metadata:
14     labels:
15       app: nginx-deployment
16   spec:
17     containers:
18       - name: nginx
19         image: nginx:1.15.4
20         ports:
21           - containerPort: 80
```

Listing 4.2: simple nginx-deployment

Lines one and two define the k8s API version and the resource type, in this example is Deployment. The section related to metadata - line three, the name of the deployment, and the labels are defined. Label is stuck to this component and should be matched by the selector to generate the connection between services, pods, and deployment. Line seven determines the configuration of the resource that depends on the specified resource type (line 2), its context would be different. The first spec might include the replicas filed that define the number of replicas of pods for running and the template that specifies the configuration for pods.

### 4.2.2 The experiment of migration nodes in Kubernetes

**Taint and Toleration** Taints allow a node to repel a set of pods. Tolerations apply to pods allowing the scheduler to use tolerations to schedule pods with matching tainted nodes. Taints and Tolerations work together to ensure that pods do not schedule onto inappropriate nodes. Taints and Tolerations do not tell the pod to go to the particular node. Instead, it tells the node to accept only pods with specific tolerations. To apply the taint into a node, we can use the following command,

```
1 kubectl taint node {node-name} {taint-key}={taint-value}:{taint-effect}
```

Checking the tainted node via,

```
1 kubectl describe node {node-name} | grep Taint
```

In the first test, some pods created to experiment the concept of T&T. Before applying the taint on the node, the status of the pods is like the figure below.

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
pod-a-564999cb6b-7phzm	1/1	Running	0	6s	10.254.2.188	worker-2
pod-b	1/1	Running	0	50m	10.254.1.79	worker-1
pod-c	1/1	Running	0	50m	10.254.1.62	worker-1
pod-d-699b8589df-2mqt8	1/1	Running	0	48s	10.254.2.23	worker-2

Figure 4.5: pods status

As it shows in the Figure 4.5, pod-a and pod-d are running on worker-2. Then the tolerations apply to the pod-a (Listing 4.3), but not to the pod-d.

```
1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   name: pod-a
5 spec:
6   selector:
7     matchLabels:
8       app: pod-a
9   template:
10    metadata:
11     labels:
12       app: nginx
13    spec:
14     containers:
15     - name: nginx
16       image: nginx:1.15.4
```

```

17     ports:
18       - name: http
19         containerPort: 80
20     tolerations:
21       - key: "node"
22         operator: "Equal"
23         value: "migration"
24         effect: "NoExecute"

```

Listing 4.3: adding toleration

So, after adding taint to worker-2 (Figure 4.6), the result would be as follows (Figure 4.7).

```

netlab@cloud-client:~/mostafa$ kubectl describe node worker-2 | grep Taint
Taints: node=migration:NoExecute

```

Figure 4.6: check added taint to node

```

netlab@cloud-client:~/mostafa$ kubectl get pods -o wide
NAME                                READY   STATUS    RESTARTS   AGE   IP              NODE
pod-a-6d86fd8cf5-cjz99             1/1     Running   0           10m   10.254.2.85     worker-2
pod-b                               1/1     Running   0           70m   10.254.1.79     worker-1
pod-c                               1/1     Running   0           70m   10.254.1.62     worker-1
pod-d-699b8589df-m8cmw             1/1     Running   0           12s   10.254.1.191    worker-1

```

Figure 4.7: status after assigning taint and toleration

Pod-d removes from the worker-2 node because we taint the node. So, those pods that do not have specific tolerations are removed from the tainted node. It removes from worker-2 and starts on worker-1 as a new pod.

Taint effects define how nodes with a taint react to Pods that don not tolerate it. For example, the **NoSchedule** taint effect means that unless a Pod has matching toleration, it will not be able to schedule onto the host<sup>1</sup>. Other supported effects include **PreferNoSchedule** and **NoExecute**. The former is the soft version of NoSchedule. If the **PreferNoSchedule** is applied, the system will try not to place a Pod that does not tolerate the taint on the node, but it is not required. Finally, if the **NoExecute** effect is applied, the node controller will immediately evict all Pods without the matching toleration from the node. If the requirement is to restrict a pod to a certain node, it is achieved via another concept called as nodeAffinity.

**nodeAffinity concept** nodeAffinity is a property of Pods that attracts them to a set of nodes. It does not guarantee that other Pods are not placed into the same nodes. It can be beneficial to ensure that certain pods need to only schedule on nodes with particular characteristics, such as the amount of memory or CPU. There are two types of node affinity: required and preferred.

- Required node affinity: This type of affinity determines that a pod should schedule on a node with certain labels. If no such node exists, the pod will not be scheduled.



- Preferred node affinity: This type of affinity defines that a pod needs to be scheduled on a node with specific labels, but it allows the pod to schedule on a different node if no such node is available.

There are two steps to active the nodeAffinity. In Step 1, we label the nodes by setting a property in key=value format.

```
1 kubectl label nodes {node-name} {label-key=label-value}
```

```
netlab@cloud-client:~/mostafa$ kubectl label nodes worker-2 node=migration
node/worker-2 labeled
netlab@cloud-client:~/mostafa$ kubectl get node worker-2 --show-labels
NAME        STATUS    ROLES    AGE      VERSION   LABELS
worker-2    Ready     <none>    7d20h    v1.22.4    beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,node=migration
```

Figure 4.8: add labels

In Step 2, we specify the node affinity property (in a similar key=value format) on the pod in the pod specification YAML file.

```
1 apiVersion: v1
2 kind: Deployment
3 metadata:
4   name: pod-a
5 spec:
6   affinity:
7     nodeAffinity:
8       requiredDuringSchedulingIgnoredDuringExecution:
9         nodeSelectorTerms:
10          - matchExpressions:
11            - key: node
12              operator: In
13              values:
14                - migration
15   containers:
16   - name: my-app
17     image: my-app:latest
```

Listing 4.4: add nodeAffinity

Two pods generate with one replica, the pod-a is the one with four replicas. As Figure 4.9 shows, two of them are placed on worker-1 and two others on worker-2.

```
netlab@cloud-client:~/mostafa$ kubectl get pods -o wide
NAME                                READY   STATUS    RESTARTS   AGE   IP              NODE
pod-a-564999cb6b-2jjq6              1/1     Running   0           22s   10.254.2.91     worker-2
pod-a-564999cb6b-c59p4              1/1     Running   0           4s    10.254.1.157    worker-1
pod-a-564999cb6b-hxnnr              1/1     Running   0          106s   10.254.2.167    worker-2
pod-a-564999cb6b-krsh7              1/1     Running   0           22s   10.254.1.85     worker-1
pod-b-6699bb9899-4kx7c              1/1     Running   0          118s   10.254.1.97     worker-1
pod-c-769687896d-lq6dr              1/1     Running   0          102s   10.254.1.50     worker-1
```

Figure 4.9: nodes status after adding labels

By adding the nodeAffinity based on Listing 4.4, all replicas of pod-a go to worker-2.

According to Figure 4.10, all replicas of the pod-a move to the worker-2, since it has the same label and the nodeAffinity added to the pod-a. It creates an opportunity to experiment with the migration concept.

```

netlab@cloud-client:~/mostafa$ kubectl get pods -o wide
NAME                                READY   STATUS    RESTARTS   AGE   IP              NODE
pod-a-5945897897-8f8nw             1/1     Running   0           11s   10.254.2.60     worker-2
pod-a-5945897897-g4d4z             1/1     Running   0           11s   10.254.2.190    worker-2
pod-a-5945897897-p46jp             1/1     Running   0           40s   10.254.2.154    worker-2
pod-a-5945897897-v7p8k             1/1     Running   0           11s   10.254.2.113    worker-2
pod-b-6699bb9899-4kx7c             1/1     Running   0           36m   10.254.1.97     worker-1
pod-c-769687896d-lq6dr             1/1     Running   0           36m   10.254.1.50     worker-1

```

Figure 4.10: nodes status after adding nodeAffinity

**Time measurement of migration for stateless deployment** In this section, the migration of one pod from one node to another node is measured by using the Kubernetes Python Client [25]. The image of nginx uses to do this experiment. Since the state of the deployed application is not important, there is no need to configure the statefulset workload API. In this case, one single deployment YAML file is used for generating the pods with some configuration for resource management like CPU and RAM usage. To provide migration, label, and nodeAffinity were added to node and pod respectively, and dynamically.

k8s Python Client provides several APIs that can be used to manage the K8s elements like pods, nodes, and their features. Moreover, it provides a suitable method to programmatically access and manage the resources in a cluster, such as pods, deployments, and services. To use the Kubernetes Python client, Python should be installed on the machine and have access to a Kubernetes cluster. The Python package manager installs via using pip:

```
1 pip install kubernetes
```

Once the client installs, its APIs can be accessed by importing this package.

```

1 import kubernetes
2
3 # create a client
4 client = kubernetes.client.CoreV1Api()
5
6 # list the pods in the default namespace
7 pods = client.list_pod_for_all_namespaces()
8
9 # print the names of the pods
10 for pod in pods.items:
11     print(pod.metadata.name)

```

In this experiment, some functions related to ApiClient(), CoreV1Api() and AppsV1Api() use to measure time migration.

- ApiClient, is a class that represents an HTTP client for interacting with the Kubernetes API. It provides a convenient way to make requests to the API and handle the response.
- CoreV1Api, is another class in the Kubernetes Python client that is designed for managing resources in the v1 API group, which includes core resources such as pods, services, and namespaces. It provides methods for creating, updating, and deleting these resources, as well as for viewing the status of these resources.

- AppsV1Api, is designed for managing resources in the apps/v1 API group, which includes resources such as Deployments, ReplicaSets, and StatefulSets.

Depending on the type of resource to manage, either CoreV1Api or AppsV1Api can be used. For example, if the purpose is to manage a Deployment, use AppsV1Api, while if managing a pod is the priority, would use CoreV1Api.

There are five functions to simulate the migration with the help of Kubernetes Python. It allows the user to interact with a Kubernetes cluster using the Kubernetes API. The initial state of the cluster is clean. There are 3 nodes, one of them is master, and two are worker nodes. They are named control-plane, worker-1, and worker-2. To have this experiment, in the first step, the simple pod by using the deployment file shows on Listing 4.2 with no specific properties is used by employing the function as follows. It takes the name of the file and then, deploys the pod.

```
1 def create_deployment(name):
2     utils.create_from_yaml(api.ApiClient, name, verbose=False)
```

After deployment first pods either in worker-1 or worker-2, we need to add a label to another one that the pod is applied. In this case, we can simulate the migration from that node to another with the concept of nodeAffinity. In the function below (Listing 4.5), gets the information of the node where the pods were deployed (lines 16 till 20). If the pods were deployed on worker-1, the label should be added to worker-2 and vice versa. The label added via using the api\_CoreV1Api.patch\_node. **patch\_node** is a method allowing to update a node in a Kubernetes cluster. It works by sending a PATCH request to the Kubernetes API server, which edits the node with the specified modifications.

```
1 def add_label_node():
2     body = {
3         "metadata": {
4             "labels": {
5                 "node": "migration",
6             }
7         }
8     }
9     ret = api_CoreV1Api.list_namespaced_pod(namespace="default")
10    for i in ret.items:
11        print(i.spec.node_name)
12        worker_node = i.spec.node_name
13    # listing the cluster nodes
14    node_list = api_CoreV1Api.list_node()
15    # patching the node labels
16    for node in node_list.items:
17        if (worker_node == "worker-1"):
18            api_CoreV1Api.patch_node("worker-2", body)
19        else:
20            api_CoreV1Api.patch_node("worker-1", body)
```

Listing 4.5: add label to node

Once the pod is deployed on the node, the target node is labeled, and the nodeAffinity property needs to add to the pod. When this property is added to the pod, it is going to move to the node that possesses the label. So, the running pod terminates and a new one creates on another node. In this case, the migration from one node to another happens. As Figure 4.11 shows, the pod at the first is running on worker-1.

```
netlab@cloud-client:~/mostafa$ kubectl get pods --watch -o wide
```

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE	NOMINATED NODE
nginx-deployment-645657846d-jq7s2	0/1	Pending	0	0s	<none>	<none>	<none>
nginx-deployment-645657846d-jq7s2	0/1	Pending	0	0s	<none>	worker-1	<none>
nginx-deployment-645657846d-jq7s2	0/1	ContainerCreating	0	0s		<none>	worker-1
nginx-deployment-645657846d-jq7s2	1/1	Running	0	2s	10.254.1.26	worker-1	worker-1
nginx-deployment-7f6d75b867-nzfgq	0/1	Pending	0	0s	<none>	<none>	<none>
nginx-deployment-7f6d75b867-nzfgq	0/1	Pending	0	0s	<none>	<none>	worker-2
nginx-deployment-7f6d75b867-nzfgq	0/1	ContainerCreating	0	0s	<none>	<none>	worker-2
nginx-deployment-7f6d75b867-nzfgq	1/1	Running	0	1s	10.254.2.90	worker-2	worker-2
nginx-deployment-645657846d-jq7s2	1/1	Terminating	0	11s	10.254.1.26	worker-1	worker-1
nginx-deployment-645657846d-jq7s2	0/1	Terminating	0	12s	10.254.1.26	worker-1	worker-1
nginx-deployment-645657846d-jq7s2	0/1	Terminating	0	12s	10.254.1.26	worker-1	worker-1
nginx-deployment-645657846d-jq7s2	0/1	Terminating	0	12s	10.254.1.26	worker-1	worker-1

Figure 4.11: pods status during migration

Then, by adding the label and nodeAffinity, the pod is going to move to worker-2. According to the function in Listing 4.6.

**V1NodeSelectorTerm** in line 3, is a type in the Kubernetes Python client that designates a node selector term in a Kubernetes node affinity or anti-affinity configuration. This object has the following fields:

- `match_expressions`, a list of node selector expressions used to select nodes. Each expression consists of a key, an operator, and a list of values. The key and values are strings, and the operator is one of the following: In, NotIn, Exists, or DoesNotExist.
- `match_fields`, a list of node selector expressions that use to set nodes based on the values of fields in the node's metadata.

In this code, the `V1NodeSelectorTerm` object specifies that the pod should be scheduled on nodes that have the node label set to migration.

`replace_namespaced_deployment` in line 18, is a method that allows updating a Deployment resource in a Kubernetes cluster. It works by sending a PUT request to the Kubernetes API server, which replaces the existing Deployment with the new one.

```

1 def add_nodeAffinity():
2     deployment = api AppsV1Api.read_namespaced_deployment(name='nginx-deployment',
3         namespace='default')
4     terms = client.models.V1NodeSelectorTerm(
5         match_expressions=[
6             {'key': 'node',
7              'operator': 'In',
8              'values': ["migration"]}
9         ]
10    )
11    node_selector = client.models.V1NodeSelector(node_selector_terms=[terms])
12    node_affinity = client.models.V1NodeAffinity(
13        required_during_scheduling_ignored_during_execution=node_selector)
14    affinity = client.models.V1Affinity(node_affinity=node_affinity)
15
16    # replace affinity in the deployment object
17    deployment.spec.template.spec.affinity = affinity
18
19    # finally, push the updated deployment configuration to the API-server
20    api AppsV1Api.replace_namespaced_deployment(name=deployment.metadata.name,
21        namespace=deployment.metadata.namespace, body=deployment)

```

Listing 4.6: Add nodeAffinity to pod

In this case, we can measure this time of migration with the help of another function (Listing 5). The result of this measurement shows in Figure 4.12. It depicts that the migration from worker-1 to worker-2 takes 19.590 milliseconds. As the picture shows, at the first the pod deployed on worker node 1, then after adding the label to node and nodeAffinity object to the pod, it terminated from worker-1 and migrated to worker-2.

```
netlab@cloud-client:~/mostafa$ python time-final.py
Watching...
New Pod nginx-deployment-645657846d-jq7s2 runs at: 1664534407.4802918 in node: worker-1
Watching...
Watching...
Watching...
Watching...
New Pod nginx-deployment-7f6d75b867-nzfgq runs at: 1664534408.6205463 in node: worker-2
Watching...
Pod nginx-deployment-645657846d-jq7s2 terminates at: 1664534408.6401365 in node: worker-1
Migration time is: 19590 [MS]
```

Figure 4.12: migration time in ms

```
1 import time
2 from datetime import datetime
3 from kubernetes import client, config, utils, watch
4
5 config.load_kube_config()
6 watch = watch.Watch()
7 api_ApIClient = client.ApiClient()
8 api_CoreV1Api = client.CoreV1Api()
9 api_AppsV1Api = client.AppsV1Api()
10
11 # ***** time measurement function *****
12 def time_measurement():
13     for event in watch.stream(func=api_CoreV1Api.list_namespaced_pod, namespace="
14         default", timeout_seconds=500):
15         print("Watching...")
16         if event["object"].metadata.deletion_timestamp != None and event["object"
17             ].status.phase == 'Running':
18             state = 'Terminating'
19         else:
20             state = str(event["object"].status.phase)
21             if (state == "Terminating"):
22                 end_time = time.time()
23                 print("Pod", event["object"].metadata.name, "terminates at: ",
24                     end_time, "in node: ", event["object"].spec.node_name)
25                 watch.stop()
26             elif (state == "Running"):
27                 start_time = time.time()
28                 print("New Pod", event["object"].metadata.name, "runs at: ",
29                     start_time, "in node: ", event["object"].spec.node_name)
30
31 # ***** add label node *****
32 def add_label_node():
33     body = {
34         "metadata": {
35             "labels": {
36                 "node": "migration",
37             }
38         }
39     }
40
41     ret = api_CoreV1Api.list_namespaced_pod(namespace="default")
42     for i in ret.items:
43         print(i.spec.node_name)
```

```

40     worker_node = i.spec.node_name
41     # Listing the cluster nodes
42     node_list = api_CoreV1Api.list_node()
43     # Patching the node labels
44     for node in node_list.items:
45         if (worker_node == "worker-1"):
46             api_CoreV1Api.patch_node("worker-2", body)
47         else:
48             api_CoreV1Api.patch_node("worker-1", body)
49
50 # ***** create deployment function *****
51 def create_deployment(name):
52     utils.create_from_yaml(api_ApIClient, name, verbose=False)
53
54 # ***** create addNodeAffinity function *****
55 def add_nodeAffinity():
56     deployment = api_AppsV1Api.read_namespaced_deployment(name='nginx-deployment',
57     namespace='default')
58     terms = client.models.V1NodeSelectorTerm(
59         match_expressions=[
60             {'key': 'node',
61              'operator': 'In',
62              'values': ["migration"]}
63     ]
64     )
65     node_selector = client.models.V1NodeSelector(node_selector_terms=[terms])
66     node_affinity = client.models.V1NodeAffinity(
67         required_during_scheduling_ignored_during_execution=node_selector)
68     affinity = client.models.V1Affinity(node_affinity=node_affinity)
69
70     # replace affinity in the deployment object
71     deployment.spec.template.spec.affinity = affinity
72
73     # finally, push the updated deployment configuration to the API-server
74     api_AppsV1Api.replace_namespaced_deployment(name=deployment.metadata.name,
75     namespace=deployment.metadata.namespace, body=deployment)
76
77 # ***** main function *****
78 def main():
79     create_deployment('./simple-depl-nginx.yaml')
80     time.sleep(5)
81     add_label_node()
82     time.sleep(5)
83     add_nodeAffinity()
84     time_measurement()
85
86 if __name__ == '__main__':
87     main()

```

Listing 4.7: migration code

### 4.2.3 Simple CRUD operation using IPFS and OrbitDB

In the following steps, there is an explanation for the code. In the chosen directory, we need to create a project by using the following command. It generates package.json, package-lock.json, and node\_modules folder.

```

1 npm init --yes
2 npm install --save orbit-db ipfs

```

To experiment with simple CRUD operation via Orbitdb and IPFS, after creating the node project, we need to create an IPFS node and an instance of Orbitdb by running the following code.

```

1 export default class NewPiecePlease {
2   constructor(Ipfs, OrbitDB) {
3     this.OrbitDB = OrbitDB;
4     this.Ipfs = Ipfs;
5   }
6   async create() {
7     this.node = await this.Ipfs.create({
8       preload: { enabled: false },
9       repo: "./ipfs",
10      EXPERIMENTAL: { pubsub: true },
11      config: {
12        Bootstrap: [],
13        Addresses: { Swarm: [] },
14      },
15    });
16    await this._init();
17  }
18  async _init() {
19    this.orbitdb = await this.OrbitDB.createInstance(this.node);
20    this.defaultOptions = {
21      accessController: {
22        write: [this.orbitdb.identity.id],
23      },
24    };
25    const docStoreOptions = {
26      ...this.defaultOptions,
27      indexBy: "hash",
28    };
29    this.pieces = await this.orbitdb.docstore("pieces", docStoreOptions);
30    await this.pieces.load();
31    this.onready();
32  }
33 }

```

Listing 4.8: newpieceplease.js

There is a class known as NewPiecePlease in a JS file, and its constructor gets two arguments, Ipfs for the javascript ipfs constructor and OrbitDB for the orbit-db constructor. To run the application, another file as index.js needs to create with the code below.

```

1 import NewPiecePlease from "./newpieceplease.js";
2 import * as Ipfs from "ipfs";
3 import OrbitDB from "orbit-db";
4 import { CID } from "multiformats/cid";
5
6 const main = async () => {
7   try {
8     const NPP = new NewPiecePlease(Ipfs, OrbitDB);
9     NPP.onready = async () => {
10       console.log(NPP.OrbitDB.id);
11     }
12     await NPP.create();
13   } catch (error) {
14     console.log(error);
15   }
16 };
17 main();

```

After creating the class instance of NewPiecePlease, via calling "await NPP.create()", it calls the function for generating a new IPFS node with the default settings.

- preload: { enabled: false }, helps load balance the global network and prevent

DDoS.

- repo: `"/ipfs"`, determines the path of the repo in Node.js.
- EXPERIMENTAL: `{ pubsub: true }`, activates the IPFS pubsub, a method of communicating between nodes and is required for OrbitDB usage.
- config: `{ Bootstrap: [], Addresses: { Swarm: [] } }`, sets both bootstrap peers list and swarm peers list.

The `init()` function has some configurations as below to create a local database of type "docstore".

- `defaultOptions` and `docStoreOptions`, introduce some parameters for the database that is generated.
- `accessController: { write: [this.orbitdb.identity.id] }`, defines ACL.
- `this.pieces = await this.orbitdb.docstore("pieces", docStoreOptions)`, creates the database. Once this line is completed, the database is open and ready to use.
- `await this.pieces.load()`, loads the latest snapshot of data from the database into memory.

The function below can apply to add new data to the database. First of all, it checks for the existing hash for the given piece. If it does not exist, put the new one with the counter into the database, and return its cid. The function, `"await this.pieces.put({...})"`, takes an object to store and returns a multihash, that is the hash of the content added to IPFS.

```

1  async addNewPiece(hash, instrument = "Piano") {
2    const existingPiece = this.getPieceByHash(hash);
3    if (!existingPiece) {
4      const dbName = "counter." + hash.substr(20, 20);
5      const counter = await this.orbitdb.counter(dbName, this.defaultOptions);
6      const cid = await this.pieces.put({
7        hash,
8        instrument,
9        counter: counter.id,
10     });
11     return cid;
12   } else {
13     return await this.updatePieceByHash(hash, instrument);
14   }
15 }

```

Listing 4.9: `addNewPiece` function on `newpieceplease.js`

To run this function and see the output [Figure 4.13], the code below needs to be added into `index.js` file.

```

1  const addCid = await NPP.addNewPiece( "
2  QmNR2n4zywCV61MeMLB6JwPueAPqheqpfiA4fLPMxouEmQ"
3  );
4  if (addCid) {
5    const content = await NPP.node.dag.get(CID.parse(addCid));
6    console.log(content.value.payload);

```



```
6 }

```

Listing 4.10: addNewPiece function on newpieceplease.js

```
[nodemon] restarting due to changes...
[nodemon] starting node index.js
(node:48817) ExperimentalWarning: The Fetch API is an experimental feature. This feature could change at any time
(Use 'node --trace-warnings ...' to show where the warning was created)
{
  op: 'PUT',
  key: 'QmQ54QN18DCceGzKjfmBhLTRExNboQ8opUd988SLEtZpW',
  value: {
    hash: 'QmQ54QN18DCceGzKjfmBhLTRExNboQ8opUd988SLEtZpW',
    counter: '/orbitdb/zdpuAtZvqzo7BaNmYnhUgFgeCK6L5MJ66nF1AQDf7E8U3D1CP/counter.BhLTRExNboQ8opUd988S',
    instrument: 'Piano'
  }
}
```

Figure 4.13: The output of PUT operation

```
1 async updatePieceByHash(hash, instrument = "Piano") {
2   const piece = await this.getPieceByHash(hash);
3   if (piece) {
4     piece.instrument = instrument;
5     const cid = await this.pieces.put(piece);
6     return cid;
7   }
8   return null;
9 }
10
11 async deletePieceByHash(hash) {
12   const piece = await this.getPieceByHash(hash);
13   if (piece) {
14     const cid = await this.pieces.del(hash);
15     return cid;
16   }
17   return null;
18 }
```

Listing 4.11: delete and update functions on newpieceplease.js. language

The output of the code above depicts on Figure 4.14.

**OrbitDB replication Experiment** The purpose is to replicate the orbitdb database from one node to another one running on two different machines in ipfs. To do this experiment, two nodes run with these IP addresses 172.16.203.5, and 172.16.226.132 need to connect via configurations related to ipfs. Both nodes have to configure their ipfs with a specific configuration. One of the important fields in the configuration file is the Swarm address.

The objective of the Swarm is to provide a decentralized network for storing and sharing content. It is made up of nodes that are connected to exchange content and information. When a node wants to retrieve a file from the Swarm, it sends a request to the Swarm, which then forwards the request to the nodes that have the file. The nodes then send the file back to the requesting node, and the file is reconstructed from the different pieces received from the different nodes. Swarm addresses are addresses that the local daemon will listen on for connections from other IPFS peers. A swarm address is multiaddress that specifies the location of a node in the network. It consists of the IP address, node port number, and the peer ID of the node.

```
[nodemon] restarting due to changes...
[nodemon] starting 'node index.js'
(node:60156) ExperimentalWarning: The Fetch API is an experimental feature. This feature could change at any time
(Use 'node --trace-warnings ...' to show where the warning was created)
delete item: {
  op: 'DEL',
  key: 'QmQ54QN18DCceGzKjfmBhLTRExNboQ8opUd988SLEtZpW',
  value: null
}
[
  {
    hash: 'QmcFUVG75QTMok9jrteJzBUXeoamJsuRseNuDRupDhFwA2',
    instrument: 'Piano'
  },
  {
    hash: 'QmNfQhx3WvJRLMnKP5SucMRXEPy9YQ3V1q9dDWNC6QYMS3',
    instrument: 'Piano'
  },
  {
    hash: 'QmcJPfExkBAZe8AVGfYHR7Wx4EW1Btjd5MXX8EnHCkrq54',
    instrument: 'Piano'
  },
  {
    hash: 'QmefKrBYeL58qyVAaJoGHXXEgYgsJrxo763gRRqzYHdL6o',
    instrument: 'Piano'
  }
]
```

Figure 4.14: The output of DELETE operation

```
1 export const Config = {
2   repo: "./ipfs/orbitdb-poc-consumer",
3   config: {
4     Addresses: {
5       Swarm: ["/ip4/0.0.0.0/tcp/4011", "/ip4/0.0.0.0/tcp/4012/ws"],
6     },
7     Bootstrap: [
8       "/ip4/127.0.0.1/tcp/4001/p2p/QmRZs65q9ESgi7Yf43gLQLymJ54CR28sqkdijdAsxrPo67"
9     ],
10    "/ip4/127.0.0.1/tcp/4002/ws/p2p/
11    QmRZs65q9ESgi7Yf43gLQLymJ54CR28sqkdijdAsxrPo67",
12    "/ip4/172.16.203.5/tcp/4001/p2p/
13    QmRZs65q9ESgi7Yf43gLQLymJ54CR28sqkdijdAsxrPo67",
14    "/ip4/172.16.203.5/tcp/4002/ws/p2p/
15    QmRZs65q9ESgi7Yf43gLQLymJ54CR28sqkdijdAsxrPo67",
16    ],
17  },
18  EXPERIMENTAL: {
19    pubsub: true,
20  },
21 };

```

In line 6, there are two Swam addresses. It is an array that specifies the addresses of bootstrap nodes that the IPFS node will use to discover and connect to the IPFS network.

- /ip4/0.0.0.0/tcp/4011, specifies a bootstrap node using the IP4 transport on TCP port 4011. The IP address 0.0.0.0 means that the node will listen on all available network interfaces.
- /ip4/0.0.0.0/tcp/4012/ws, means a bootstrap node using the IP4 transport on TCP port 4012 and the WebSockets (WS) transport.

In line 8, it shows the list of Bootstrap addresses that the IPFS node will try to connect to. The primary difference between Swarm and Bootstrap is that Swarm is used to specify

the addresses that the IPFS node will listen on for incoming connections, while Bootstrap is used to specify the addresses of nodes that the IPFS node will try to join. The addresses related to another node that the node needs to connect are set in each configuration file. The addresses have specific format like `/ip4/{ip-address}/tcp/{port}/ipfs/{peer-id}`. The ip-address is the IP address of the node, port is the port number of the node, and peer-id is the peer ID of the node.

The code below is to boot the ipfs based on the configuration discussed. The result of this part shows on Figure 4.16. Since we need to connect to another node running in another machine, the function of connecting is used in line 6, accepting the address of it to connect [Figure 4.15].

```
=====
Connect to Peer
-----
[
  {
    addr: <Multiaddr 04ac10e284060faba503221220b64c81e02022a62db9e856467ddff7735f0c3fd01b704adc6684
603c9ba7aff8 - /ip4/172.16.226.132/tcp/4011/p2p/QmacGaFBSQGKZweKtWXNX6vTXGEnSPZgrjxme9b9PeEMzf>,
    peer: 'QmacGaFBSQGKZweKtWXNX6vTXGEnSPZgrjxme9b9PeEMzf'
  }
]
=====
```

Figure 4.15: Connect to Peer

```
1  const bootIpfs = async () => {
2    console.info("Booting IPFS...");
3    ipfs = await Ipfs.create(Config);
4    const id = await ipfs.id();
5
6    await ipfs.swarm.connect(
7      "/ip4/172.16.226.132/tcp/4011/p2p/
      QmacGaFBSQGKZweKtWXNX6vTXGEnSPZgrjxme9b9PeEMzf"
8
9    console.info(await ipfs.bootstrap.list());
10
11    ipfs.libp2p.on("peer:disconnect", (peerId) => {
12      console.info('Lost Connection', JSON.stringify(peerId.id));
13    });
14
15    ipfs.libp2p.on("peer:connect", (peer) => {
16      console.info("Producer Found:", peer.id);
17    });
18  };
```

`ipfs.libp2p.on` is a method in the JavaScript IPFS API allowing to register of a callback function to be executed when a specific event occurs on the libp2p module of the IPFS instance. It allows connecting to other nodes in the Swarm, exchange messages with them, and perform other networking tasks. The **on method** takes two arguments, the event name to listen for, and the callback function to execute when the event occurs. The callback function will be passed an event object as an argument, which contains information about the event that occurred.

The `createIdentity` function in line 2, generates a new private key used to create a new peer ID for the identity. The peer ID is a unique identifier to specify the identity on the IPFS.

```

crownlabs@etcd3:~/f2p2/OrbitDB/replication/producer$ node producer.js
Booting IPFS...
Swarm listening on /ip4/127.0.0.1/tcp/4001/p2p/QmSgtxlobNc1FmW9PzJEGhQfjtU5HQ5EakZZ74FfNt45r
Swarm listening on /ip4/172.16.203.5/tcp/4001/p2p/QmSgtxlobNc1FmW9PzJEGhQfjtU5HQ5EakZZ74FfNt45r
Swarm listening on /ip4/127.0.0.1/tcp/4002/ws/p2p/QmSgtxlobNc1FmW9PzJEGhQfjtU5HQ5EakZZ74FfNt45r
Swarm listening on /ip4/172.16.203.5/tcp/4002/ws/p2p/QmSgtxlobNc1FmW9PzJEGhQfjtU5HQ5EakZZ74FfNt45r

=====
IPFS booted
-----
{
  id: 'QmSgtxlobNc1FmW9PzJEGhQfjtU5HQ5EakZZ74FfNt45r',
  publicKey: 'CAASpgIwggE1MA0GC5qGSIB3DQEBAQUAA4IBDwAwggEKAoIBAQDZR0LPKv2J49Ya72d2Q2161F3me+HKgT
0s8SqlihUKgIif9racj1Pt5Wo4bJH94LDf8ftsWYdHZVMgvw+x7086z9n8ig4kch5J9GK0KqCJuuy/oyZVqk3HNwFZXQntfzC
FxqEKQFdda5LQExFuZlf8jM8ed3dIHC9/s+J7TDjTn76hpVzIyBvt2ZQsicM09o0DJ3snZPepqkzPl8ddBldcgw2FxfYpfdk
AGjkfV0m2d04y8IyERpKNMShic/ev8bRmTil2zxt47GP/AnR0Se5Z/yMnAg6V1j2KPrvpjNLwk+zDoS3ULwhpugN3PEEV1jt
E0ZMsoS65eVdwk9N/kuH7AgMBAE=',
  addresses: [
    <Multiaddr 047f000001060fa1a503221220409e7701e7b2a096582860d89cb8b152cf1d0ad78c9076c7bcecca7
0944bab8d - /ip4/127.0.0.1/tcp/4001/p2p/QmSgtxlobNc1FmW9PzJEGhQfjtU5HQ5EakZZ74FfNt45r>,
    <Multiaddr 047f000001060fa2dd03a503221220409e7701e7b2a096582860d89cb8b152cf1d0ad78c9076c7bce
cca70944bab8d - /ip4/127.0.0.1/tcp/4002/ws/p2p/QmSgtxlobNc1FmW9PzJEGhQfjtU5HQ5EakZZ74FfNt45r>,
    <Multiaddr 04ac10cb05060fa1a503221220409e7701e7b2a096582860d89cb8b152cf1d0ad78c9076c7bcecca7
0944bab8d - /ip4/172.16.203.5/tcp/4001/p2p/QmSgtxlobNc1FmW9PzJEGhQfjtU5HQ5EakZZ74FfNt45r>,
    <Multiaddr 04ac10cb05060fa2dd03a503221220409e7701e7b2a096582860d89cb8b152cf1d0ad78c9076c7bce
cca70944bab8d - /ip4/172.16.203.5/tcp/4002/ws/p2p/QmSgtxlobNc1FmW9PzJEGhQfjtU5HQ5EakZZ74FfNt45r>
  ],
  agentVersion: 'js-ipfs/0.46.0',
  protocolVersion: '9000'
}
=====

```

Figure 4.16: Boot the IPFS in one node as Producer

```

1  const bootOrbitdb = async () => {
2    const identity = await createIdentity.createIdentity({ id: "privateKey" });
3    orbitdb = await OrbitDB.createInstance(ipfs, { identity });
4    console.info(
5      'Orbit Database instantiated ${JSON.stringify(orbitdb.identity)}'
6    );
7
8    database = await orbitdb.docstore("producer", {
9      accessController: {
10        write: ["*"],
11      },
12    });
13
14    await database.load(1);
15    console.info("Database initialized");
16    console.info('Address: ${database.address}');
17  };

```

The function in line 3, is a function call that constructs a new instance of the OrbitDB database. An identity is an object that specifies the private key and peer ID of the database to identify the OrbitDB instance on the IPFS network and to sign and verify the database operations performed by the instance.

**Pubsub** is a feature that enables nodes to publish messages to a topic and subscribe to receive messages on that topic. It permits nodes in a decentralized network to transmit each other in a publish/subscribe model without the need for a central server or broker. Pubsub works by establishing a distributed hash table (DHT) for each topic, which contains a list of the nodes that are interested in receiving messages on that topic. When a node wants to publish a message to a topic, it adds the message to the DHT for that topic. Other nodes that are subscribed to the topic can then retrieve the message

```
Starting OrbitDb...
Orbit Database instantiated {"id":"032c38d3e6a72d7615348a0c47ae9f4d4ae3824fa2dae0f294d3c39898058752cf","p
ublicKey":"04b52a55cb95ddcf5fa2764868d43773931f13d80861a022a59d0f9451e46ef3a788852185ff45ec9a3a0b867b96ca
31cd400b613df8936b0bd456e3771c0fd9d5","signatures":{"id":"3044022040251ae533c4c2b8f5c919214fbd3598a82b1b4
c72541b964dcf09e93b76350302201311c69d6d9571c310023df7468f4c352eaf4a8e5505111224dff738b54cb382","publicKey
":"30440220220fcf4a7b5d843bd0ff766b65eb907a9c52624f2acd70f4c00c3bdc98faec8002205f00a4ef2e7a14130ff01d0a44
adbebl195ee624b84d92444b8dc5f6f1d7661d"},"type":"orbitdb"}

=====
Database initialized
Address: /orbitdb/zdpuAzoyTYPqa7BNa2Npm9hyrKcFX1HNNMzh5r8FK7nSdwg5W/producer
=====
```

Figure 4.17: Starting OrbitDB with its initialized address

from the DHT. It can be used for a variety of purposes in IPFS, including sending messages between nodes, synchronizing data across the network, and enabling real-time communication between nodes.

```
1  /**
2   * Publish a message to IPFS
3   * @param {string} topic - The topic to publish to
4   * @param {Buffer} message - The message to publish
5   * @returns {Promise<void>}
6   * @async
7   * @name publishIpfsMessage
8   * @description Publish a message to IPFS
9   */
10 const publishIpfsMessage = async (topic, message) => {
11   await ipfs.pubsub.publish(topic, message);
12   console.info('published [${message.toString()}] to ${topic}');
13 };
14
15 let i = await getLatestId();
16
17 // Publish a message every 5 seconds
18 intervalHandle = setInterval(async () => {
19   i += 1;
20   await publishIpfsMessage(
21     "replication-orbitdb",
22     Buffer.from('producer_${i}')
23   );
24   await writeDatabase({ _id: 'p-${i}', value: "from producer" + i });
25 }, 5000);
```

In line 19, the topic called "replication-orbitdb" subscribes to ipfs. The message has to be sent in a form of a Buffer object. A Buffer is a data type in JavaScript that represents a fixed-size sequence of bytes. It is used to store binary data, such as the contents of a file or an image. **Buffer.from** is a static method that creates a new Buffer object from an array of bytes, a string, or an array of integers. In this case, the message string is passed as an argument, and the method returns a new Buffer object containing the bytes of the message string.

After running the code on both sides, the producer publishes the specific topic with messages and writes on the database. On another side, the consumer gets the message and also connects to the created database.

```
published [producer 188] to orbitdb-remote-poc
{ _id: 'p-188', value: 'from producer188' }
put into database: zdpuAy1W1r9r1T5t7FNMQQqA6XXz5Ca9cb3vraBPAgAWNQcn { "_id": "p-188", "value": "from produce
r188" }
/orbitdb/zdpuAzoyTYPqa7BNa2Npm9hyrKcFX1HNNMzh5r8FK7nSdwg5W/producer updated database {"op": "PUT", "key": "c
-188", "value": {"_id": "c-188", "msg": "from consumer: 188"}}
published [producer 189] to orbitdb-remote-poc
{ _id: 'p-189', value: 'from producer189' }
put into database: zdpuAx8LzUNHxmQkfnz3w2yaRP8XkeoKCA2jTW6EE1BCWUz82 { "_id": "p-189", "value": "from produce
r189" }
/orbitdb/zdpuAzoyTYPqa7BNa2Npm9hyrKcFX1HNNMzh5r8FK7nSdwg5W/producer updated database {"op": "PUT", "key": "c
-189", "value": {"_id": "c-189", "msg": "from consumer: 189"}}
published [producer 190] to orbitdb-remote-poc
{ _id: 'p-190', value: 'from producer190' }
put into database: zdpuAurQuV33xLQgJnz3qPkVZMzbDJS4dxxUhzveeEK3pk2Z { "_id": "p-190", "value": "from produce
r190" }
/orbitdb/zdpuAzoyTYPqa7BNa2Npm9hyrKcFX1HNNMzh5r8FK7nSdwg5W/producer updated database {"op": "PUT", "key": "c
-190", "value": {"_id": "c-190", "msg": "from consumer: 190"}}
published [producer 191] to orbitdb-remote-poc
{ _id: 'p-191', value: 'from producer191' }
put into database: zdpuAqcK3mBTGSnMTRzw9YSKAD8JZEyFL7wo1Pk5iImTtHkqH { "_id": "p-191", "value": "from produce
r191" }
/orbitdb/zdpuAzoyTYPqa7BNa2Npm9hyrKcFX1HNNMzh5r8FK7nSdwg5W/producer updated database {"op": "PUT", "key": "c
-191", "value": {"_id": "c-191", "msg": "from consumer: 191"}}
published [producer 192] to orbitdb-remote-poc
{ _id: 'p-192', value: 'from producer192' }
```

Figure 4.18: pubsub

#### 4.2.4 Helm Charts experiment

Helm is a package manager for kubernetes helping to manage the sophisticated k8s applications via tools to streamline the installation and applications management. It simplifies packing all the resources needed to run an application like deployment, services, etc into a single chart that can be handled easily. To have the benefits of the helm, its CLI needs to be installed and set up a local Helm chart repository. It has simple instructions to install, first, the desired version needs to be downloaded from its official website, then unpack it and move to the proper destination in our operating system [17].

```
1 Download desired version
2 Unpack it (tar -zxvf helm-v3.0.0-linux-amd64.tar.gz)
3 mv linux-amd64/helm /usr/local/bin
```

To start working on the helm, its structure needs to be created via create command,

```
1 helm create chartjenkins
2 tree chartjenkins
```

This will form a new directory called chartjenkins that contains the files and directories needed for a Helm chart. The directory structure of a Helm chart looks like this:

```
crownlabs@etcd1:~/helm$ tree chartjenkins/
chartjenkins/
|-- charts
|-- Chart.yaml
|-- templates
|   |-- helpers.tpl
|   |-- jenkins-deployment.yaml
|   |-- jenkins-service.yaml
|   |-- NOTES.txt
|-- values.yaml
```

Figure 4.19: Helm Chart Tree

```
1 chartjenkins/
2   Chart.yaml           # A YAML file containing information about the chart
3   values.yaml         # The default configuration values for this chart
4   charts/              # A directory containing any charts upon which this chart
5   depends.            # A directory of templates that, when combined with values,
   templates/
```

```

6      # will generate valid Kubernetes manifest files.
7  templates/NOTES.txt # OPTIONAL: A plain text file containing short usage notes

```

In this experiment, two YAML files related to Jenkins were provided to deploy it via the helm command.

```

1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: {{ .Values.name }}
5  spec:
6    replicas: 1
7    selector:
8      matchLabels:
9        app: {{ .Values.name }}
10   template:
11     metadata:
12       labels:
13         app: {{ .Values.name }}
14     spec:
15       containers:
16       - name: {{ .Values.name }}
17         image: {{ printf "%s:%s" .Values.image.repository .Values.image.tag }}
18         env:
19         - name: JENKINS_USERNAME
20           value: {{ default "test" .Values.env.jenkinsUsername | quote }}
21         - name: JENKINS_PASSWORD
22           {{- if .Values.env.jenkinsPassword }}
23           value: {{ .Values.env.jenkinsPassword }}
24           {{- else }}
25           value: testPassword
26           {{- end }}
27       ports:
28       {{- range .Values.containerPorts}}
29       - name: {{ .name }}
30         containerPort: {{ .port }}
31       {{- end }}

```

Listing 4.12: jenkins-deployment.yaml

Instead of hardcoding the information in the YAML file as before, create a template that is based on the Go programming language. The braces `{{` and `}}` are the opening and closing brackets to enter and exit template logic. For instance the template like `name: {{ .Values.name }}`, refers to the `values.yaml` file and read the value of the `name` and replace it with it.

```

1  image:
2    repository: jenkins/jenkins
3    tag: latest
4
5  env:
6    jenkinsUsername: "mostafa"
7    jenkinsPassword: test
8
9  name: jenkins
10
11 containerPorts:
12   - name: http
13     port: 8080
14
15 service:
16   type: NodePort
17   nodePort: 30080

```

Listing 4.13: values.yaml

All Helm-packed applications have an associated values.yaml file which dictates the configuration of an application. This object provides access to values passed into the chart. To test the template, commands like 'helm template <template name> <template folder>' can be used. In this case, we can check if the created manifest is suitable for deploying into the cluster.

```
1 helm template jenkins ./chartjenkins/
```

```
crownlabs@control-plane-1:~/helm$ helm template jenkins ./chartjenkins/
---
# Source: chartjenkins/templates/jenkins-service.yaml
apiVersion: v1
kind: Service
metadata:
  name: jenkins
spec:
  type: NodePort
  ports:
    - name: http
      port: 8080
      targetPort: http
      nodePort: 30080
  selector:
    app: jenkins
---
# Source: chartjenkins/templates/jenkins-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: jenkins
spec:
  replicas: 1
  selector:
    matchLabels:
      app: jenkins
  template:
    metadata:
      labels:
        app: jenkins
    spec:
      containers:
        - name: jenkins
          image: jenkins/jenkins:latest
          env:
            - name: JENKINS_USERNAME
              value: "mostafa"
            - name: JENKINS_PASSWORD
              value: test
          ports:
            - name: http
              containerPort: 8080
```

Figure 4.20: Helm Template Creation

As can be seen from the Figure 4.20, all template defines in the deployment and service yaml files are substituted with the values defined in the values.yaml file.

A values file is passed into helm install or helm upgrade with the command (helm install <template name> <template folder> -values <values yaml address>) 'helm install jenkins . -values values.yaml'. Individual parameters passed with --set (such as helm install --set name=jenkins ./chartjenkins/)

```
1 apiVersion: v1
2 kind: Service
```



```

3 metadata:
4   name: {{ .Values.name }}
5 spec:
6   type: {{ .Values.service.type }}
7   ports:
8     - name: http
9       port: 8080
10      targetPort: http
11      nodePort: {{ .Values.service.nodePort }}
12   selector:
13     app: {{ .Values.name }}

```

Listing 4.14: jenkins-service.yaml

It provides an opportunity to write conditions in the template,

```

1 - name: JENKINS_PASSWORD
2   {{- if .Values.env.jenkinsPassword }}
3   value: {{ .Values.env.jenkinsPassword }}
4   {{- else }}
5   value: testPassword
6   {{- end }}

```

This means that, if the password is created in the values files or provided via a set flag in the command, sets it as the value, otherwise, testPassword would be the value for Jenkins password. After installing the created template, the outcome shows the creation step is successful [Figure 4.21].

```

crownlabs@control-plane-1:~/helm/chartjenkins$ helm install jenkins . --values values.yaml
NAME: jenkins
LAST DEPLOYED: Sun Jan  8 10:14:00 2023
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
This is my first chart

```

Figure 4.21: Helm install template

By using the 'kubectl get all' command, we can see the pod, deployment, and service related to Jenkins deployed correctly and we can access it, via our local address, 172.16.226.132:30080.

```

crownlabs@control-plane-1:~/helm$ k get all
NAME                                READY    STATUS    RESTARTS   AGE
pod/jenkins-8f8cd8b6-455nx          1/1      Running   0           4h15m

NAME                                TYPE          CLUSTER-IP      EXTERNAL-IP      PORT(S)          AGE
service/jenkins                     NodePort      10.255.169.107  <none>           8080:30080/TCP   4h15m
service/kubernetes                   ClusterIP     10.255.0.1      <none>           443/TCP          2d4h

NAME                                READY    UP-TO-DATE    AVAILABLE    AGE
deployment.apps/jenkins              1/1      1              1            4h15m

NAME                                DESIRED    CURRENT    READY    AGE
replicaset.apps/jenkins-8f8cd8b6    1          1          1        4h15m

```

Figure 4.22: The status of deployment, pod and service of Jenkins

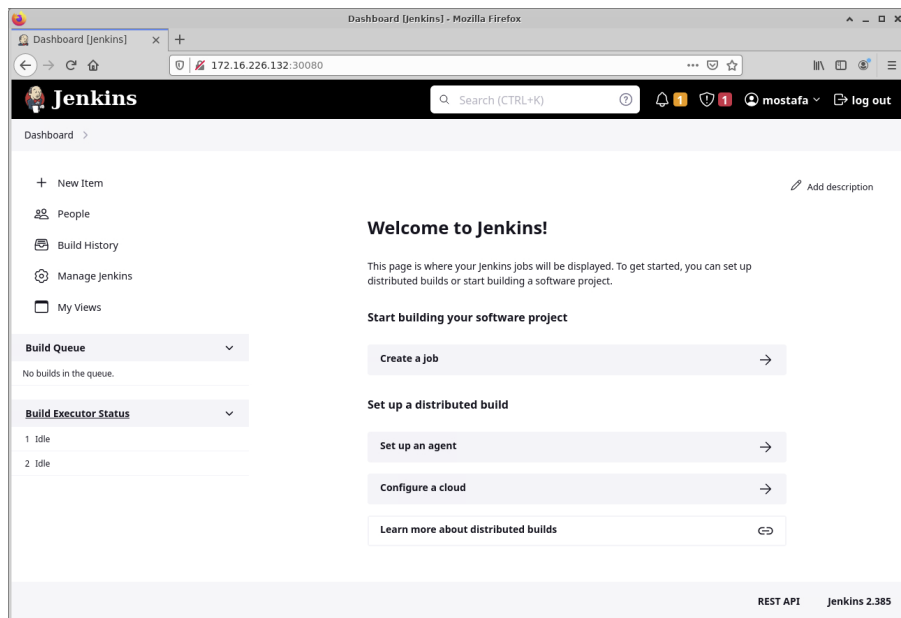


Figure 4.23: Jenkins Dashboard

## Chapter 5

# Experimental Results

This is the chapter showing the entire results of various tests that we did. Starting from a simple migration, then move to the comparison between two technologies that provide strong and weaker consistency and compare their results. Then we provided the outputs of our three defined approaches and discussed the pros and cons of each one to highlight the performance trade-offs.

There are different experiments related to migration done in this part. The first one is related to understanding the impact of resources like CPU allocation limit on the migration duration. After that, we focused more on the comparison of etcd and orbitDB as the key-value storing technologies. In the last part, the entire tests concerning etcd performance in different migration scenarios that explained it before.

### 5.1 Effect of CPU allocation limit on Migration

In this experiment, migration time was measured on five levels of CPU limits to compare them under the overloaded CPU condition. The objective is to understand the effect of CPU allocation limit as one parameter on migration duration. The nginx uses in this scenario as stateless deployment because we have no state defined in the deployment. For simulating the user behavior, an infinite request sends from another node, here is the Master node, to the deployed server to measure the impact of idle and overloaded CPU on migration duration. Requests and limits can be specified for the resources that a pod or container needs, such as CPU and memory. They use to guarantee that a pod or container has access to the resources it needs to operate correctly and to prevent it from consuming more resources than it should. In this case, we need to discuss two fields in our manifest as follows.

**Requests**, A request is the minimum amount of a resource that a pod or container needs to function correctly. When a pod or container is scheduled on a node, the scheduler ensures that the node has enough resources available to meet the pod's or container's requests.

**Limits**, is the maximum amount of a resource that a pod or container is allowed to consume. When a pod or container exceeds its limits, it may be terminated or have its resource usage decreased.

To specify these fields for CPU and memory resources, the `resources.requests` and `resources.limits` fields can be applied. The unit of measurement for CPU resources is the CPU core. This amount can be specified in terms of millicores, which are 1000th of a CPU core. The value 250m in the `cpu` field indicates that 250 millicores are being requested or limited. This is equivalent to 0.25 CPU cores. And the value 512Mi in the `memory` field indicates that 512 megabytes are being requested. This is equivalent to  $512 * 1024 = 524288$  kibibytes.

For example, to specify that a pod requires 0.25 CPU core and 512 MB of memory, and has a limit of 1 CPU core and 1 GB of memory, it would be as follows,

```

1 apiVersion: v1
2 kind: Pod
3 metadata:
4   name: my-pod
5 spec:
6   containers:
7   - name: my-app
8     image: my-app:latest
9     resources:
10      requests:
11        cpu: 250m
12        memory: 512Mi
13      limits:
14        cpu: 1000m
15        memory: 1Gi

```

Listing 5.1: specify requests and limits for the resources

To increase the CPU load, the busybox image as a load generator uses and sends an infinite requests to the nginx-deployment service on port 80 [24]. A load generator is a tool used to simulate load on a system or application, typically for testing and performance evaluation. It generates traffic or requests to a system and measures the performance under different load conditions. In the context of Kubernetes, load generators can be used to test the performance of a Kubernetes cluster, or to test the performance of a specific application or service running in the cluster. The following commands use to increase the load,

```

1 kubectl run -it --rm load-generator --image=busybox -- /bin/sh
2 while true; do wget -q -O- http://php-apache.default.svc.cluster.local; done

```

There is a loop in the script that simulates 100 times migration between two worker nodes ["worker-1" and "worker-2"]. The difference between the current time when a pod terminates, and a new one starts on another node is registered as the migration time in milliseconds/ms.

In the following experiment, another node takes responsibility to generate the traffic. It means that one node that is different from the nodes that are acting in the experiment for migration generates the traffic for deployed server. Hence, the total amount of CPU allocates for the experiment [Figure 5.1].

As you can see in the diagram, the master node sent infinite requests to the deployed server to increase the load of the CPU. So the experiment did base on two important factors. One is when we set different CPU allocation limits with an overloaded CPU, and another is the scenario when setting no CPU allocation limit, which means 100% uses in the CPU allocation limit field with and without stress on the CPU.

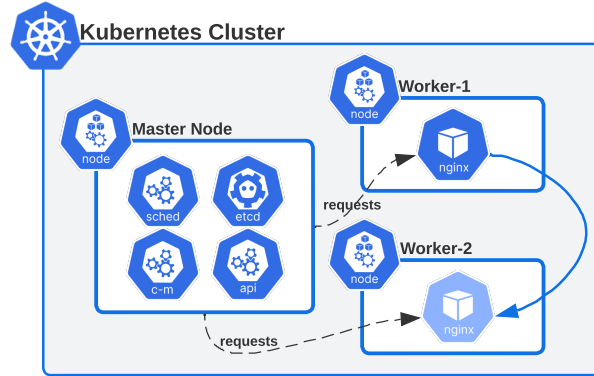


Figure 5.1: Migration scenario

According to the results, increasing the CPU allocation limit does not seriously affect the migration time. Based on the 100 collected migration data in five levels of limits for CPU allocation, the mean value experienced a decline from 27.02ms to 25.74ms for 10%, and 25% CPU limits respectively. Then its value remains stable at around 25.59ms when the value for CPU limits grows to 500m, 750m, and reaches 1000m (100%) [Figure 5.2]. For each collected data, mean value, Standard Deviation(S.D), and confidence interval

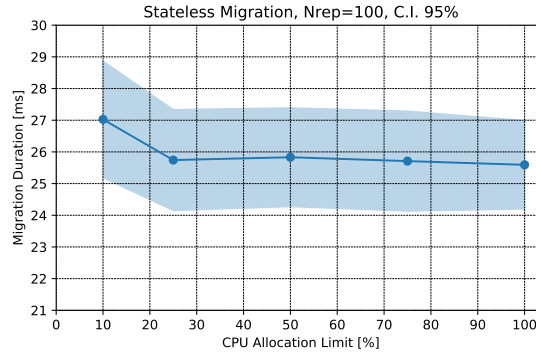


Figure 5.2: The effect of CPU allocation limit [10%,25%,50%,75%,100%] on migration duration of stateless application with Confidence Interval 95%

with a confidence coefficient of 95% were calculated and saved with the collected data in a separate file. The Confidence Interval here shows the uncertainty in the sample variables. The CI has bounded above and below the calculated mean, which likely would contain an unknown population parameter. Confidence level refers to the percentage of probability that the true population parameter when repeating this sampling process many times, we would expect the true average value to fall within the interval we calculate 95% of the time. The calculated interval gives a range of values that we are reasonably sure contain the true average migration time.

One result shows below.

```

1 Number of Data: 100
2 CPU_limit : 750m
3 Collected Data: [23.286, 39.934, 32.597, 20.749, 16.686, 10.921, 22.694, 30.696,
  24.869, 20.166, 37.808, 31.639, 23.069, 31.775, 18.588, 17.548000000000002,
  15.461, 23.997999999999998, 24.314, 16.416, 65.468, 27.519,
  27.349999999999998, 16.962, 19.529999999999998, 34.452000000000005, 7.498,
  14.741999999999999, 21.955, 30.945, 27.453000000000003, 19.88, 23.046, 19.892,
  23.95, 30.994, 30.016000000000002, 23.569, 20.815, 28.441000000000003,
  21.717, 36.651, 23.499, 31.61, 20.47, 22.939999999999998, 24.563000000000002,
  30.837, 22.043, 19.294999999999998, 28.682, 27.896, 48.228, 40.497, 19.994,
  26.622, 18.316, 23.494999999999997, 25.245, 19.98, 19.787, 21.104000000000003,
  22.53, 22.354, 25.966, 16.917, 25.564, 21.945, 25.339, 50.118,
  30.169999999999998, 24.945999999999998, 31.983999999999998, 17.658, 19.288,
  24.255, 36.995, 19.2, 26.107000000000003, 28.632, 32.69, 36.038000000000004,
  20.330000000000002, 19.157, 24.278000000000002, 21.441000000000003, 31.86,
  25.29, 21.764, 34.609, 26.693, 30.537000000000003, 25.618, 21.817, 38.003,
  23.07, 21.362, 25.285, 25.733, 24.258]
4 Mean of the Value: 25.709629999999997
5 Standard Deviation: 8.054985948659377
6 Confidence Interval, min_value: 24.130852754062758
7 Confidence Interval, max_value: 27.288407245937236

```

The same experiment occurred to compare the migration duration containing no CPU limits, means 100% CPU allocation limit but with and without a load generator. The result shows that the mean value in both cases is lower than in the previous situation. It is because, in one scenario, there is no traffic and requests sent to the services [Figure 5.3]. Therefore, the mean value of migration is in the lowest situation [there are also no CPU limit constraints].

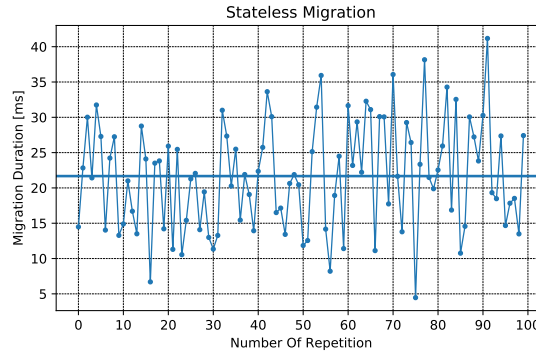


Figure 5.3: Migration time without load on CPU and no CPU allocation limits

In the second case [Figure 5.4], there is a load generator for sending an infinite requests to the deployment service. The results of this experiment illustrate that the property of CPU limits in the pod has no essential effect on the migration time. And as the graph indicates, the average value in this scenario is nearly equal to the Figure 5.2.

**In conclusion,** according to the table 5.1, there is a few difference between the value of the mean, standard deviation, and confidence interval in several conditions of CPU allocation limits. Moreover, by doing this experiment, we realize that the CPU allocation

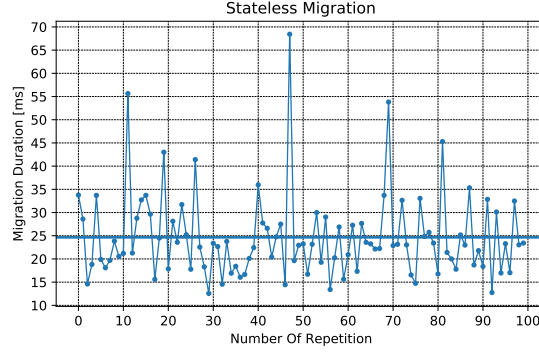


Figure 5.4: Migration time with load on CPU and no CPU allocation limits

limit does not impact the migration time, leading to a positive attitude in migration pods in different nodes.

In addition, during a migration, the state of the deployment, including its configuration and data, must be moved from the source to the target node. In some cases, it seems that by increasing the CPU allocation limit, the migration duration decline, allowing the pods to use better resources on the resource node. Based on our result, this factor does not affect migration duration, and increasing the limit might not necessarily result in faster migration. This result provides a piece of information that the relationship between CPU allocation limit and migration duration leads to helping resource utilization better.

Stateless Migration			
CPU Allocation Limit	Mean [ms]	S.D. [ms]	C.I. [ms]
100m (10%)	27.02	9.38	[25.18 - 28.86]
250m (25%)	25.74	8.13	[24.14 - 27.33]
500m (50%)	25.83	7.95	[24.27 - 27.39]
750m (75%)	25.71	8.05	[24.13 - 27.28]
1000m (100%)	25.59	7.12	[24.19 - 26.98]
No CPU Limits, with load	24.65	9.02	-
No CPU Limits, without load	21.67	7.60	-

Table 5.1: Comparison of migration duration in different CPU Allocation Limit, Nrep=100

## 5.2 etcd experiment

The objective of this section is to realize the effect of the number of keys in the latency of put, get and reset operations in the etcd ecosystem. We did this experiment for two important reasons. One is to satisfy our expectations, the other is to compare the GET operation for two levels of consistency. We expect that duration per key is independent of state size. Our expectation meets via the result from Figure 5.10. However, we have no specific observation concerning consensus overhead because we do not set any latency in communication between peers. Since the experiment happened in CrownLabs with high bandwidth and low latency characteristics, the result can not show this overhead. In this case, three nodes are required to create an etcd cluster. In the first step, etcd needs to be installed in all nodes by using the following installation steps from its official repository [10].

If the command "etcd --version" shows the version of installed etcd means that it works properly [Figure 5.5].

```
crownlabs@cloud-client:~$ etcd --version
etcd Version: 3.5.5
Git SHA: 19002cfc6
Go Version: go1.16.15
Go OS/Arch: linux/amd64
```

Figure 5.5: etcd version

To create a cluster, a specific approach shell file is created to run in each node. This simple example shows the simple way to write and read a key from an etcd cluster [Figure 5.6].

```
crownlabs@cloud-client:~$ etcdctl put name Mostafa
OK
crownlabs@cloud-client:~$ etcdctl get name
name
Mostafa
```

Figure 5.6: etcd simple put and get

In the further step, a simple cluster with three nodes is performed to realize the concept of etcd better, the etcd needs to be installed in all of them. After that, the code below has to run in each of them. It is crucial to mention that, in each machine, the `NODE_IP` in the first line has to change to the IP of that machine accordingly.

The command below is to start an etcd server in a cluster. it has several flags,

- `--name`, specifies the name of the etcd serve. This name is used to identify the server within the cluster and is used in the `--initial-cluster` flag to specify the list of servers in the cluster.
- `--initial-advertise-peer-urls`, this flag sets the URL at which other etcd servers in the cluster can reach this server for peer communication. Peer communication uses for things like replicating data, electing a leader, and detecting and removing failed servers from the cluster.



- `-listen-peer-urls`, specifies the URL at which this server will listen for peer communication from other etcd servers in the cluster.
- `-advertise-client-urls`, specifies the URL at which clients can reach this server to read and write data.
- `-listen-client-urls`, establishes the URL at which this node will listen for client requests..
- `-initial-cluster-token`, uses to identify the cluster.
- `-initial-cluster`, sets the initial set of etcd servers in the cluster, including their names and URLs.
- `-initial-cluster-state`, specifies the initial state of the cluster. In this case, it is set to "new" because this is the first time the cluster is being set up.

After following the instruction code provided by [41] in each node, in any node, we can verify the status of the cluster by using the command as follow [Figure 5.7].

```
1 ETCDCCTL_API=3 etcdctl --endpoints=http://127.0.0.1:2379 member list --write-out=
  table
```

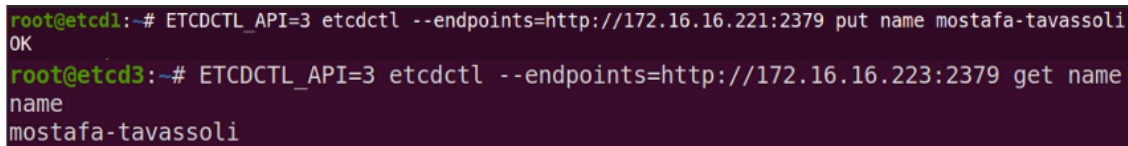
Listing 5.2: command to see the etcd cluster Status



ID	STATUS	NAME	PEER ADDRS	CLIENT ADDRS	IS LEARNER
458e4e886d23770e	started	etcd2	http://172.16.16.222:2380	http://172.16.16.222:2379	false
49f5f38fb52a7508	started	etcd1	http://172.16.16.221:2380	http://172.16.16.221:2379	false
c650959f39d8e3f4	started	etcd3	http://172.16.16.223:2380	http://172.16.16.223:2379	false

Figure 5.7: etcd cluster status with three peers

After the mentioned configuration, in each node/server, a simple reading and writing key value can do to experiment. As the Figure 5.8 indicates, in one node a key is written by the put command, and in another node that key is ready by the get command.



```
root@etcd1:~# ETCDCCTL_API=3 etcdctl --endpoints=http://172.16.16.221:2379 put name mostafa-tavassoli
OK
root@etcd3:~# ETCDCCTL_API=3 etcdctl --endpoints=http://172.16.16.223:2379 get name
name
mostafa-tavassoli
```

Figure 5.8: etcd cluster put and get operations

After cluster creation, an experiment needs to do for testing the latency of essential operations. In this scenario, three nodes act an important role like before. The experiment started by providing several keys and key sizes as an argument for each operation. In each function for every operation, the duration of operations calculates and register. There is a function that generates a random string for putting into a database which is called `randStr()`. The function is below related to putting data.

```

1 //Initializing buffer on etcd
2 duration = std::chrono::milliseconds::zero();
3 for (int i=0; i<N/key_size; i++) {
4     std::string key_i = std::to_string(i); //coverting iterator to string
5     std::string key = key_prefix+key_i; //complete key string
6     std::string rand_string = randStr(key_size);
7     KV kv_t; kv_t.key=key; kv_t.value=rand_string;
8     start = std::chrono::high_resolution_clock::now();
9     etcdPut(kv_t);
10    stop = std::chrono::high_resolution_clock::now();
11    duration = duration + (stop - start);
12 }
13 double init_duration = duration.count();

```

Since etcd can support both linearizability and serializability, the put operation tested with both aspects. After repeating the experiment many times, finally, the chars below show the result for the number of 1, 10, and 100 keys, with the size of 100K.

**In conclusion,** the outcome depicts that, the number of keys has no essential effects on latency for put, get, and reset operations. In other words, the duration per key remains steady while the overall duration increases. The calculated CIs related to the RESET operation are compatible with the constant duration of this command. As it shows, it decreases firstly, and again increases. This is a result we expect to reach and doing this test to prove our expectations. Another reason for experimenting with this test is to compare GET for two levels of consistency.

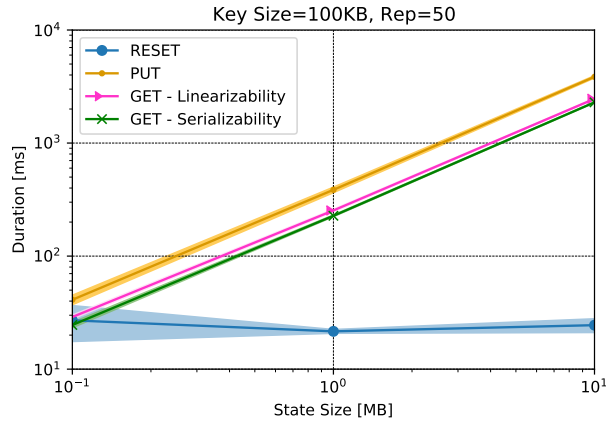


Figure 5.9: Overall time duration for 1, 10, and 100 key numbers with the size of 100KB and the repetition of 50 times.

In addition to this, the experiment shows the difference between two levels of GET consistency in etcd. One is Serializability and the other is Linerazibility. They are very similar in this test because the latency of the peers' communication is negligible. After all, the test did in Crownlabs with a small communication delay(rtt) and high throughput [Table 4.1]. As a result, the test will not be able to experience the overhead due to the consensus algorithm. To experience this overhead, we have to generate the delay in the communication.

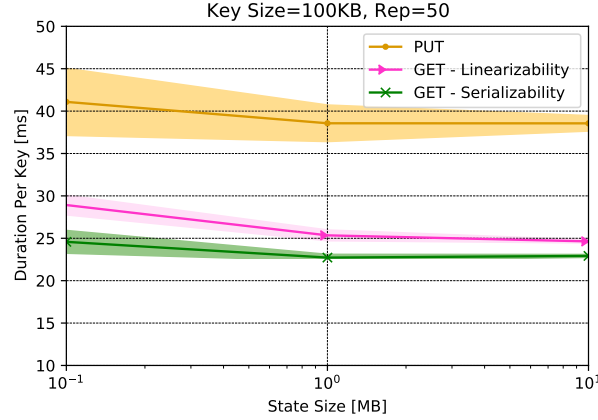


Figure 5.10: Duration per key for 1,10 and 100 key numbers

In the next section, the scenario will change from etcd to OrbitDB. The main objective is to compare the performance of these two distributed storage. One is etcd with a high level of consistency and the OrbitDB uses CRDT with a low level of consistency with an eventual consistency approach. OrbitDB is the most effective strategy to use CRDT.

### 5.3 OrbitDB Experiment

The purpose is to measure the time of operations like put, get and delete in key-value orbitdb data model. Orbitdb has types of databases with different purposes to create. The selected one is beneficial for loading data from keywords or an id. The code below shows the possibility of creating an instance of ipfs and connecting to the key-value database.

From lines one to three, the required packages are imported for using Ipfs, and Orbitdb for the rest of the project. In line six, a function as DB connection uses to create the Ipfs node based on some options to employ it for the OrbitDb database. After running this code, a new IPFS node is generated that works locally. Line 17 creates a key-value database type as a test name with the proposed options.

**overwrite: true** enables the overwriting of the current database which has a false value in default.

**accessController: {write: ["\*"]}**, is an object that contains a write attribute to set the write access to the database. The value of ["\*"] can be used to give the write access to everyone.

**replicate: true** means that the database replicates with peers in the network that is required IPFS pubsub.

```

1 import * as Ipfs from "ipfs";
2 import OrbitDB from "orbit-db";
3 import seedrandom from "seedrandom";
4
5 // create a new instance of ipfs and connect to the orbitdb database
6 const dbConnection = async () => {

```

```

7 console.log("connecting to db");
8 const ipfsOptions = {
9   preload: { enabled: false },
10   repo: "./ipfs",
11   EXPERIMENTAL: {
12     pubsub: true,
13   },
14 };
15 const ipfs = await Ipfs.create(ipfsOptions);
16 const orbitdb = await OrbitDB.createInstance(ipfs);
17 const db = await orbitdb.keyvalue("test", {
18   overwrite: true,
19   replicate: true,
20   accessController: {
21     write: ["*"],
22   },
23 });
24 await db.load();
25 return db;
26 };
27
28 // create a random number generator
29 const rng = seedrandom();
30 const data = {
31   dbConnection,
32   rng,
33 };
34 export default data;

```

The **preload: { enabled: false }** option, enables a load balance approach in the network for preventing DDoS attacks. In this case, it sets to false since the node is in offline mode.

The **repo: "./ipfs"** option, sets the location folder in Node.js. The default setting in the home directory.

The **EXPERIMENTAL: { pubsub: true }** option, activates IPFS pubsub, which is a required option for OrbitDB usage as a pattern to handle events in large-scale networks and a method for communicating between nodes.

A simple function uses to generate the random strings as a value for each key to put into the database. The value is generated through looping for specific numbers that insert from the terminal as an argument.

```

1 const randStr = () => {
2   const rng = data.rng;
3   const chars =
4     "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789!@#%&'()*";
5   let result = "";
6   for (let j = 0; j < _valueSize; j++) {
7     result += chars.charAt(Math.floor(rng() * chars.length));
8   }
9   return result;
10 };

```

The variable **rng** is a seedable random number generator in Javascript to produce the deterministic sequence of pseudorandom numbers. This function uses to pick the random character from the provided string to generate a simple string in the length of value size to put in the database as a value in each iteration.

A specific function like **put** as an API is written in which the time difference between

each operation calculates to discover the duration. To put value into the database, its operation needs to be done asynchronously. Otherwise, the get operation returns the wrong values. The async/await approach in javascript is used to satisfy this operation as follows.

```

1 // Put all keys to db - async
2 const put = async (keyNumber) => {
3   for (let i = 0; i < keyNumber; i++) {
4     const rand_string = randStr();
5     const start = performance.now();
6     await connction.put('key-${i}', rand_string);
7     const end = performance.now();
8     putDuration += end - start;
9   }
10 };

```

This function loops through the value of keyNumber received via the terminal as an array of input data. Inside it, the function for the random string generator calls in each iteration, then the time of putting this value into the database calculates via the difference between the start and end time of this operation.

```

1 // Get values from db
2 const get = async (keyNumber) => {
3   for (let i = 0; i < keyNumber; i++) {
4     const start = performance.now();
5     const value = await connction.get('key-${i}');
6     const end = performance.now();
7     getDuration += end - start;
8   }
9 };
10
11 // Delete all values from db - async
12 const del = async (keyNumber) => {
13   for (let i = 0; i < keyNumber; i++) {
14     const start = performance.now();
15     await connction.del('key-${i}');
16     const end = performance.now();
17     delDuration += end - start;
18   }
19   _writeToFile(keyNumber, _valueSize, putDuration, getDuration, delDuration);
20   putDuration = 0;
21   getDuration = 0;
22   delDuration = 0;
23 };

```

The function below uses to write the result into a file for further processing.

```

1 import fs from "fs";
2
3 // Write to file
4 const WriteToFile = (
5   keyNumber,
6   valueSize,
7   putDuration,
8   getDuration,
9   delDuration
10 ) => {
11   const date = new Date();
12   const data = `${date.getTime()},${keyNumber},${valueSize},${putDuration},${
13     getDuration
14   },${delDuration}\n`;
15   const stream = fs.createWriteStream(`./logs/orbitdb_stats_${valueSize}.csv`, {
16     flags: "a",

```

```
16   });  
17   stream.write(data);  
18   stream.end();  
19 };  
20  
21 export default WriteToFile;
```

Before discussing the result, the essential point is that GET is not an operation in OrbitDB, the get method is synchronously retrieving a value that is already kept in memory. On the other hand, the put and delete methods create put and delete operations which must be appended to the operation log. Appends to the operation log involve creating, digitally signing, encoding/hashing a new object, and writing the data to disk, leading to the huge difference between these methods on the chart.

In the end, based on different series of inputs for the key numbers and their sizes, the overall outputs depict that the number of keys has a potential impact on the duration time for the operation of PUT and DELETE. All experiments had similar reputations but with different state sizes from 1KB to 100MB with key sizes of 1KB, 100KB, and 1MB to realize the behavior of PUT, DELETE, and GET methods in OrbitDB as shown in Figure 5.11.

Time duration based on the plots depends on the number of keys and state size which shows an increasing trend. This trend happened in PUT and DELETE operations according to the behavior I explained before. However, duration per key for GET operation experienced no raising based on the State Size.

**In conclusion,** since the OrbitDB during the time of doing this experiment is under-developed for a couple of months, and based on the obtained results, we decided to do the rest of the experiments based on the etcd instead. The important reason to do this experiment with OrbitDB since it is not developed because it is the main distributed storage that uses CRDT.

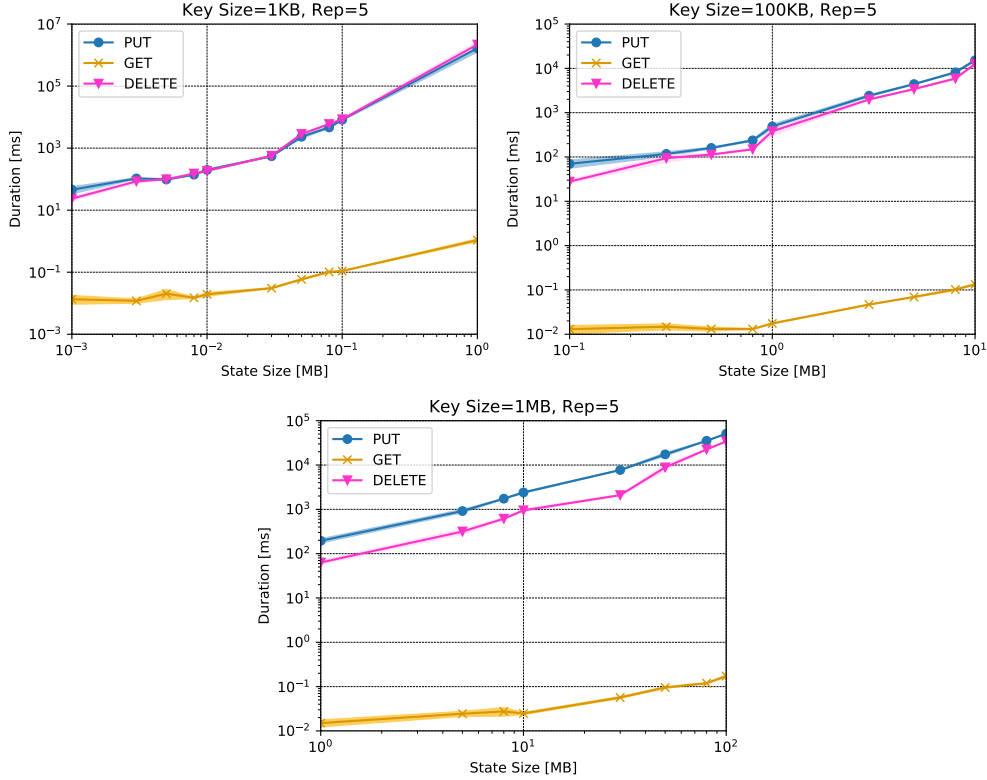


Figure 5.11: Overall duration for three main operations in OrbitDB. Number of keys are 1,10,100 with size of 1KB, 100KB, and 1MB.

## 5.4 etcd in Kubernetes cluster experiment

This section has the goal of performance measuring of etcd and RAFT algorithm in various scenarios. As discussed before, two criteria act as an important role in the performance of etcd. One is latency and the other is throughput. The first one is related to the time needed to terminate an operation. The second concept refers to the entire operations achieved within some period. Since etcd employs the RAFT algorithm in a way to replicate requests among the entire cluster to reach a consensus, it is essential to measure its performance. The factors that can be led to performance degradation in etcd are network IO and disk IO latencies [12]. In this section, by introducing several approaches, the effect of these factors on etcd calculates.

Testing etcd via etcd/bitnami inside the kubernetes leads to making the deployment of etcd instances easier. Instead of driving the etcd instance on the virtual machine like did it before, it runs inside the container that is operating on the k8s cluster, allowing us to do an advanced experiment. For example, we can increase the number of instances without difficulties to configure it. Moreover, it allows using the sidecar container along with the etcd in the same pod to generate an environment close to the real world for testing. A sidecar container is a structural pattern in which an extra container operates alongside

the main container in a single pod in a Kubernetes cluster. The sidecar container serves complementary duties or usefulness to the main container, allowing them to communicate the same network and storage resources. In this case, this container contains the `iproute2` tool, which can shape the traffic of the pod.

## 5.5 Stateless Migration Approaches

In this experiment, one master node and two worker nodes act as the main components. The master node plays a role in sending the requests via testing application, and two worker nodes use to deploy the containers related to etcd members and tools for traffic shaping like the `tc` tool.

After the k8s cluster configuration, with the help of the `etcd/bitnami` helm package manager bootstraps the etcd deployment on the created Kubernetes cluster. This package provides templates and default values for the configuration. Some essential templates need to be configured based on the needed experiment. One of them is the **statefulset** that is required for persistence. This API object maintains a sticky for each pod means that the pods are even created from similar specifications, not substitutable and each has a constant and unique identifier. It is a way to handle stateful applications. Stateful needs to implement in a scenario like utilizing the storage volumes to provide persistence.

Some aspects need to be considered for using the statefulset. One is related to the storage for a given Pod that needs to be provisioned by **PersistentVolume (PV)** according to the requested storage class. In addition to this, the statefulset requires to have **Headless Service**.

The PersistentVolume (PV) is a piece of storage in the cluster that has been provisioned by an administrator or automatically via Storage Class. It is a mechanism in a k8s cluster to provision PV dynamically. To configure the Storage Class, we need to specify the Provisioner that determines the volume plugin used for provisioning PVs. In this experiment, the Local one was selected. However, this one does not support dynamic provisioning which needs to be done by ourselves. To solve this issue, the Local Path Provisioner uses to manage the PV on the node automatically [29].

Once the Storage class is configured, we can claim the volume via a **PersistentVolumeClaim (PVC)**. It is a kind of request for storage by a user. PVC consumes the PV resources while the Pods use the node resources. Unlike the Pods that request a certain level of resources like Memory and CPU, the PVC sends the request for a specific size and access mode.

In this experiment, there is no need to have load-balancing and a single ServiceIP. Since we have the statefulset to create a stateful application, Pods are not identical and each one has a specific id. In this case, we need to have Headless Service via setting the Cluster Ip (`.spec.clusterIP`) to None for creating a service grouping. It allows clients or Pods to directly access pods without using Kubernetes' load balancing via referring to service.

This experiment did in a k8s cluster with three nodes and a size of 390Mi for the size of persistence, related to Storage Class (`persistence.size`) inside the `values.yaml` for Helm chart. The result was based on 1, 10, and 100 key numbers with size 100K for three



groups of peers, one, two, and six to satisfy the different types of stateless migration, Centralized, Reactive and Proactive respectively.

It is crucial to mention the latency and its relation to the experiment. The generated plots on the left of each figure have a delay. The delay imposes to show the latencies of the RAFT algorithm. For instance, the operations of PUT and GET with a Linearizability level of consistency express latencies because of RAFT algorithm behavior that we expected to have this result.

To emulate a certain amount of communication delay, the tc container as a sidecar runs with the etcd instance. Tc is employed to configure the Traffic Control in the Linux Kernel. The traffic controls via qdisc object. It is a way that when the kernel needs to send the traffic to an interface, it is enqueued in configured enqueued for that interface. Then, the kernel starts to manage the many packets from the qdisc.

```

1 tc qdisc add dev DEV root QDISC QDISC-PARAMETERS
2 tc qdisc add dev eth0 root netem delay 50ms
3 tc qdisc add dev enp1s0 root tbf rate 1000mbit latency 50ms burst 1000k

```

In the command above, tc uses to configure the Linux Kernel traffic that is responsible to manage the traffic on the system. The qdisc is used to configure the queue discipline like tbf (token bucket filter) in the mentioned command. The option, dev enp1s0 specifies that the configuration needs to apply for this interface. The rate of TBF queue discipline sets at 1Gbps with a maximum delay of 50ms and finally, with burst 1000k specifies the maximum number of bytes that can be sent in a burst before the TBF queue discipline begins to drop packets.

## 5.6 etcd performance

According to the result in Figure 5.12, there is no significant difference in GET and PUT operations in the case of one peer because the etcd is centralized, and no need to have the consensus algorithm. However, the impact of consensus can be seen by increasing the number of peers. For instance, for 2 and 6 peers, the consensus effect shows in the PUT operation and GET operation for its consistency levels.

Therefore, the cost of GET operation in the Linearizability mode of consistency is higher than the Serializability. Because in the case of Linearizability, the request for GET or Read data needs to go through a quorum of cluster members for consensus to bring the latest written data. On the other hand, GET/READ requests in the Serializable approach serve via any single etcd member, hence; it is cheaper than the linearizable reads.

In conclusion, the main objective is to evaluate the consequence of consensus overhead on the time required to R/W a single key-value pair when a realistic network delay is supposed. However, since we set the 50ms as delays for all communication between peers, the difference between 2 and 6 peers is not observable. In some case, the PUT operation cost more than the same operation in 6 peers. In this case, in the next scenario, we set different delays for the communications that show this distinction.

The Figure 5.13, reports the time needed to perform fundamental operations for etcd. The essential factor that needs to discuss and measure is the process of changing role

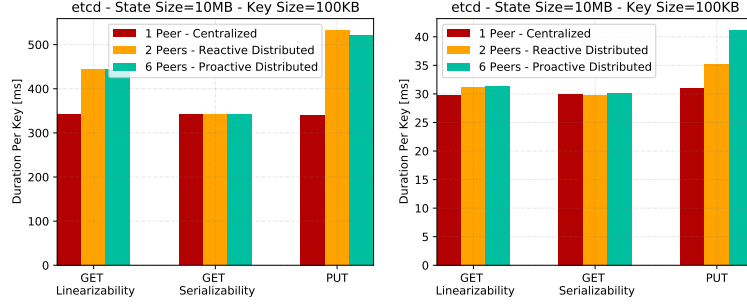


Figure 5.12: etcd performance evaluation based on 10MB state size with 100KB key size, assuming a realistic network delay 50ms(left) and without delay(right).

of leader leading to the lack of service during the re-election of a new leader in an etcd instance. Since there is no leader, no access to DB is possible and the client needs to wait till the completion of this operation to do the R/W operations on DB. Moreover, database defragmentation is the second factor disturbing the service.

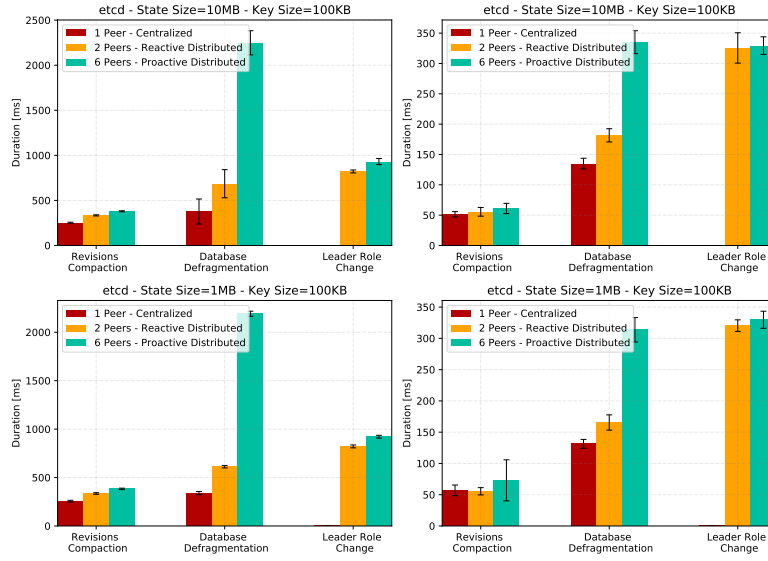


Figure 5.13: duration of etcd operations based on different state size, 10MB(top charts) and 1MB(bottom charts) with 100KB key size, for various etcd peers with delay 50ms(left) and without delay(right).

Since the space is limited and etcd stores the history of its keyspace, a set of all keys in an etcd cluster, a compaction process is necessary to clear out the keyspace history. The space that becomes free by removing the information of substituted keys before a given keyspace revision, uses for extra writing on keyspace. This process needs to be done periodically because of the performance. Some auto-compaction modes can be used to satisfy this approach. Once the compaction happens, the database requires another

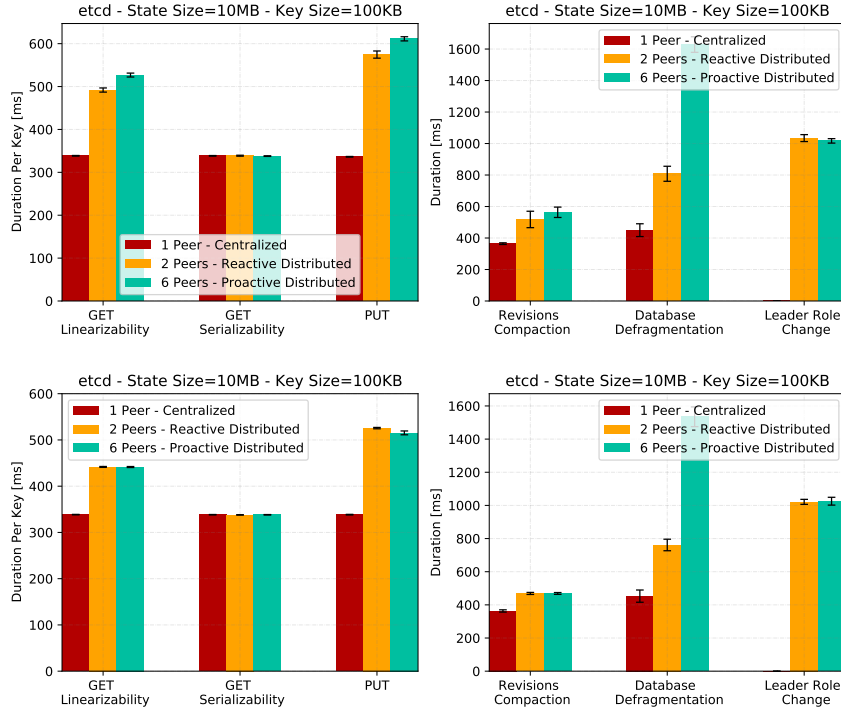


Figure 5.14: Measure the time for subset of Reactive Components, TOP requests send to Cluster, BOTTOM sending requests to the Leader

operation known as defragmentation. Compacting the old revisions fragments etcd by leaving the gaps in the backend database. So the process of defragmentation occurs every time in migration. Based on the statement on etcd official website, defragmentation can block the system from R/W operations for rebuilding the state. So, in this case, it is necessary to calculate the time when the service is unavailable [11]. We have to consider this time because we do not want to transfer useless data. As a result, this time should be considered as the time for migration of the state in the reactive approach.

In this case, the Figure 5.13 depicts the effect of peers on these factors. In terms of Role Changing, when we have one peer, this item is zero. However, we can see there the amount of time that this operation can disturb the service for the users by increasing the number of etcd peers. The Role Changing takes place in the Proactive scenario when we have several peers for selecting the leader. Once the microservice migration happens, the time needs to re-elect the leader. In general, the result shows that the number of peers has an essential impact on the mentioned factors because of the consensus algorithm.

**communication with different delays** In the previous experiment, the result is based on the fixed amount of delays of 50ms on all etcd servers. As a result, since it is the same for all peers, there is no significant difference between 2 peers and 6peers. To show this difference better, we set different delays for each peer to communicate. In this case, the experiment shows a better result as we set the [50,80,100,120,150] ms for each etcd server.

In Figure 5.15, we assume that each etcd server gets the given delays as the picture shows to represent real-world communication. In this scenario, the request sends directly to the Leader of the cluster. The leader sends its local log as the log replication process to all followers to ensure consistency. Then, the followers reply to the leader as a Log Reply scenario with different delays.

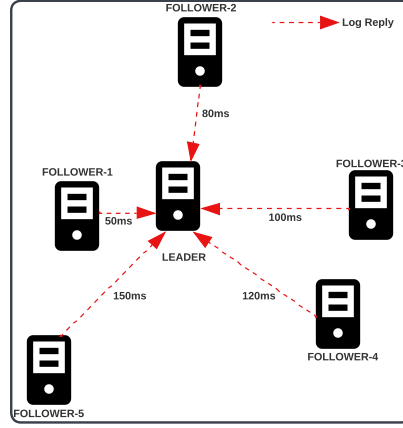


Figure 5.15: Communication delays between leader and workers to represent a realistic scenario

The provided result [Figure 5.17] is based on the scenario in the Figure 5.16, in which the request sends directly to the leader. When the leader gets the request, it puts it into its local log and then sends the Append Entries request which is the sub-process of Log Replications to the followers. Once the leader receives the major confirmation from the followers, the leader commits the request and sends the commitment request to the leaders through Log Commit.

Another test is needed to do with the same architecture to observe the difference when the client sends the request to the cluster rather than to the leader directly [Figure 5.18]. This scenario is different in a way that, once the request goes directly to the leader, the PUT or write operation, the leader does the activities that I described in the previous section.

On the other hand, If the PUT request sends to the cluster, this might get by a follower rather than a leader. When a request transmits to the cluster IP of an etcd cluster, it refers to one of the etcd nodes in the cluster, which performs as the entry point for the request. The request is then processed by the etcd node, which can either be a read or write operation. Since the followers can process the read operations and the leader is responsible to handle the write operation to ensure consistency across all the members. Hence, the request redirects to the leader and the leader do the mentioned activity. As a result, the PUT operation takes more time to complete.

If we consider the assumptions as follows,

$$\delta_F, \text{the follower delay} - \delta_L, \text{the leader delay} - \delta_Q, \text{the quorum delay} \quad (5.1)$$

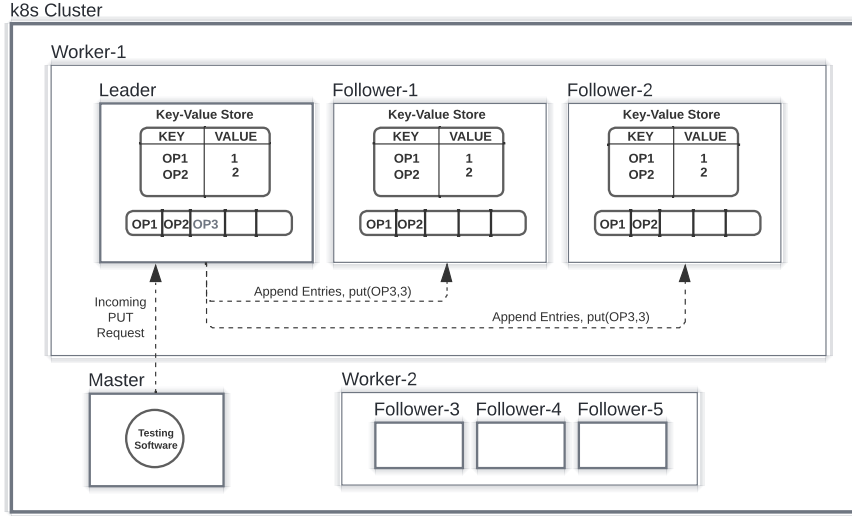


Figure 5.16: Request send directly to the Leader from the software running on the Master Node

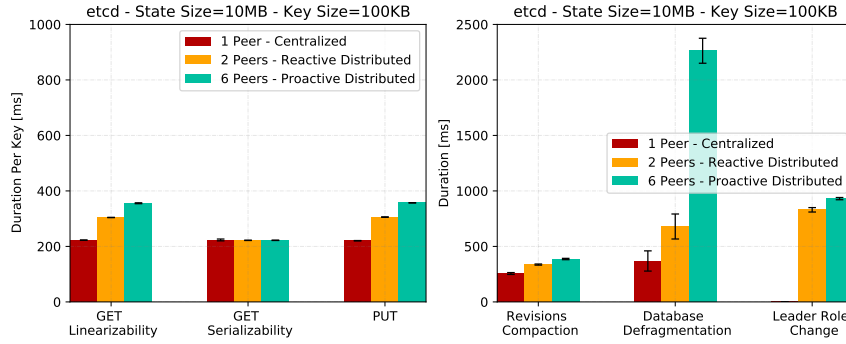


Figure 5.17: Send requests to the Leader directly, assuming a realistic different network delays [30,50,80,100,120] ms.

When the PUT and Get-L (Linearizable GET) requests that require consensus are sent to the Cluster IP, then to the follower, the delays are calculated as this

$$4.\delta_F + 2.\delta_L + \delta_Q \quad (5.2)$$

However, if the request does not need the consensus like Get-S (Serializable GET), it can be achieved as

$$3.\delta_F \quad (5.3)$$

Therefore, if the request is sent to the followers, for those operations that need consensus (PUT and Linearizable GET), the delays are high because of the redirection of the request to the leader for guaranteeing consistency.

The outcome [5.19] meets our expectation that PUT operations cost more in comparison to the previous case. In general, the cluster IP of an etcd cluster operates as a load balancer, controlling requests to the proper etcd node, and guaranteeing that the cluster can handle a large number of requests by distributing the load across its nodes.

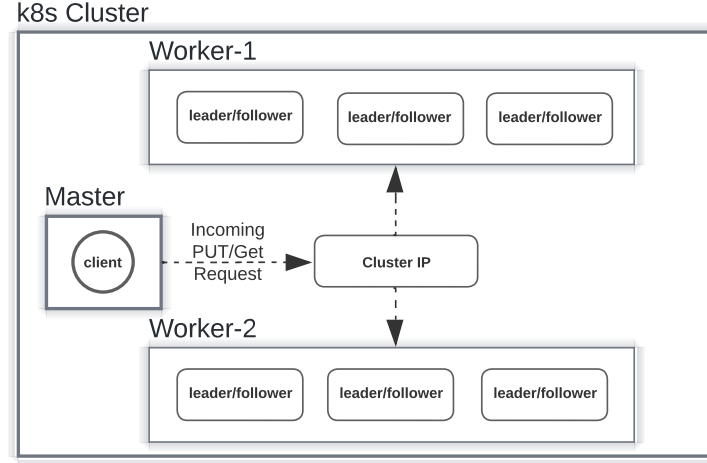


Figure 5.18: Requests send to the Cluster from the software running on the Master Node

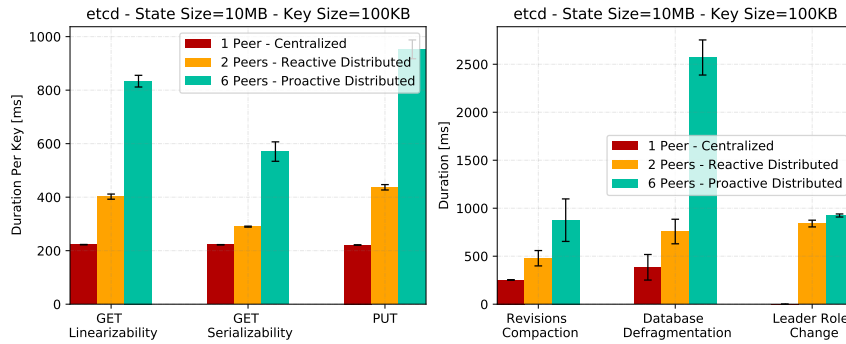


Figure 5.19: Send requests to the Cluster instead of Leader directly, assuming a realistic different network delays [30,50,80,100,120] ms.

**To be concluded,** in this series of experiments, we consider the performance of etcd in different scenarios. Based on our observations, we can conclude that the time duration is higher when the requests go to the Cluster rather than the Leader in particular for those operations that need consensus like PUT and GET linearizability since both require to have consensus among all members. If the requests go to the cluster, it gets by Cluster IP acting as Load Balancer, hence we do not know which member is the leader, if the request of write operation or read (linearizability) gets by the followers, it redirects it into

the leader because write needs consistency and the leader is responsible for it. On the other hand, as we expected, when we send the request to the leader, the time duration shows lower than the previous condition.

In the last experiment of this section, we consider the subsets of the reactive approach components to measure the migration duration time. The goal is to realize the Effect of Bandwidth on subsets of Reactive components Approaches. The components in this approach that need to be evaluated include raising the etcd container within the state, needing time to replicate etcd fully-state, the stateless migration of microservice, boot up and warm up time of microservice. This is the only scenario in which we have these components. Other approaches have only stateless microservice migration. The only scenario that has the etcd-related migration duration time is reactive.

As Figure 5.20 shows that the time migration for microservice remains steady since there is no Bandwidth limitation causing delays in terms of its booting process. However, the bandwidth effect can be seen in the fully-state replicated etcd step when the migration time decreases in a realistic situation when we have lower bandwidth in the network.

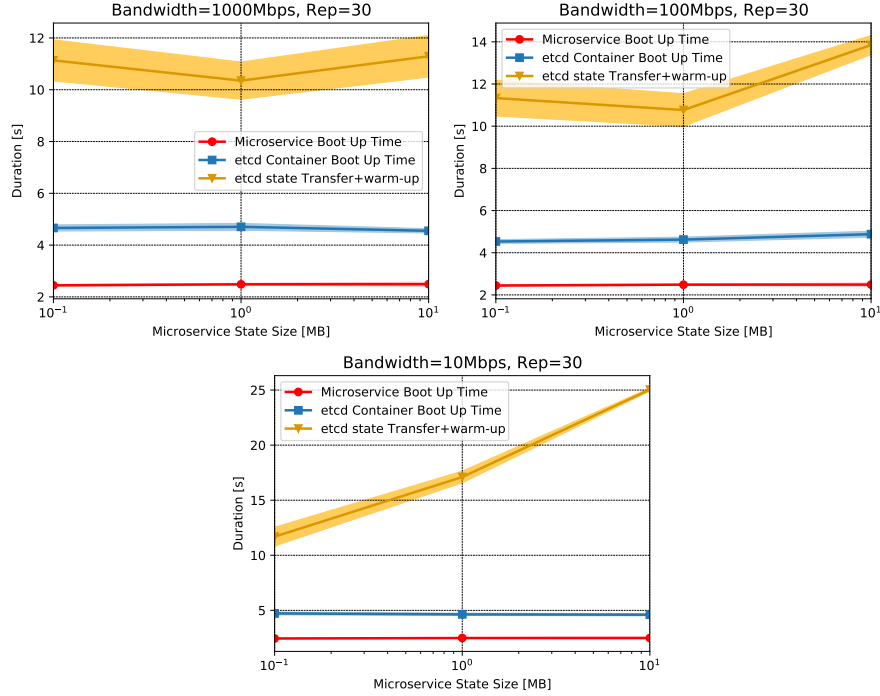


Figure 5.20: Time duration of main components migration based on the Microservice State Size

**In conclusion**, both microservice and etcd container boot-up time are not affected by bandwidth and state size. As the plots illustrate that they remain constant. On the other hand, the bandwidth can affect the transferring of the etcd, when the size increases the time duration also goes up. In the bottom plot, we can see this increasing trend. In addition, it is also crucial to mention that the etcd warm-up is involved in this plot at a certain time which plays an essential role in the Reactive scenario.

	CENTRALIZED	REACTIVE	PROACTIVE
consensus overhead	no	no	yes
migration time	microservice boot up	etcd warm-up time etcd transfer time etcd container boot up microservice boot up compaction defragmentation re-election	microservice boot up re-election
state transfer	no	yes	no

Table 5.2: characteristics of Test Environment for CrownLabs

The table 5.2 shows the main distinction between the three approaches based on three criteria. The objective is to understand the trade-off to using the architectures we introduced. For instance, although the migration time in the case of Centralized and Proactive is less than Reactive, Centralized is a single point of failure, and Proactive consumes more resources because the etcd instances need to be ready in destinations.

Figure 5.21 indicates the total amount of migration duration for all three approaches. It shows that the reactive one needs more time for the duration since it requires time for making different components to be ready as discussed above. On the other hand, the centralized takes less time for migration since there it needs only the migration of microservice. And the proactive one stands in the middle near to centralized one.

## 5.7 Throughput in Producer and Consumer

The goal of this experiment is to understand the throughput of etcd operations by using Producer and Consumer Microservice. The scenario is like that we have the Producer put a key with size 100KB into one etcd instance in the cluster 50 times. Then there is the consumer who has the duty of reading those keys from the etcd. In this case, we want to experiment with the number of keys that the producer can put in and the consumer can get in one second. The schema [Figure 5.22] shows the scenario. It depicts that the producer sends the requests into the load balancer, Cluster IP and consumer requests get data from it.

The main difference between this experiment with the previous one is that here we send the put and get requests from our testing application running inside the container that deployed in one of our worker's nodes instead of sending the requests from the



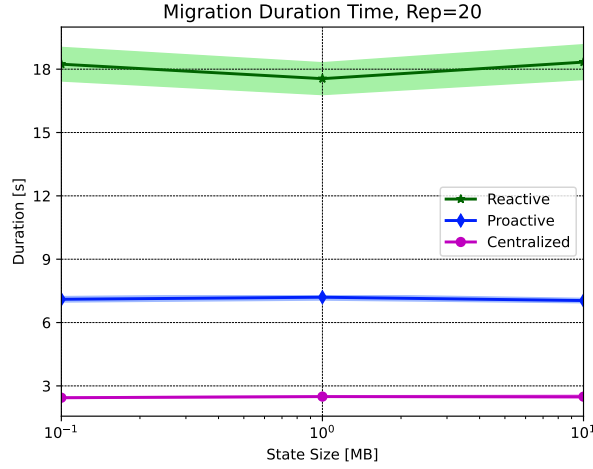


Figure 5.21: Migration duration time for all three approaches

Master node. To do this, we used the Dockerfile with all necessities and dependencies for running the application inside it. In this case, we need to install the gcc to compile our testing application, kubectl, and etcdctl to communicate via k8s and etcd cluster inside our container. Then the image creating from this Dockerfile is kept inside the docker hub for deployment of our application in the pod.

The application starts by creating the etcd cluster with different members 6,2, and 1 inside k8s cluster. Then the producer and consumer deployed in one of the worker nodes. Figure [5.22] depicts a situation in that we set different delays in communication of 6 etcd members like the previous experiment. The request from the producer goes to the cluster. Once the request goes to the cluster IP, it contacts to load balancer, so in this case, realizing which etcd member acts as the leader is obscure in advance.

According to the result and based on the mentioned scenario, the requests go to the cluster IP; they need to redirect to the leader that is responsible for handling all client requests which require cluster consensus. In this case, the number of keys putting inside the etcd storage is higher in Get Serializability mode rather than PUT and Linearizability because of consensus overhead.

The Figure 5.23 shows the result when no delays are set in the etcd members' communication. In general, the throughput is more in a situation when the requests send to the leader in comparison to when send to the cluster. This pattern is true when we put the delays in our communication with the lower throughput.

According to the Figure 5.24, there is no significant difference in one instance when the request goes for the leader or cluster. However, the impact of sending the request to each can be visible in two and six instances on throughput. In two operations like PUT and linearizable GET, the left is lower than the right because of the request redirection and consensus overhead.

Ultimately, delays and sending requests to the Leader or Cluster IP affect the throughput in etcd operations. The idea is to migrate the producer and consumer from one worker

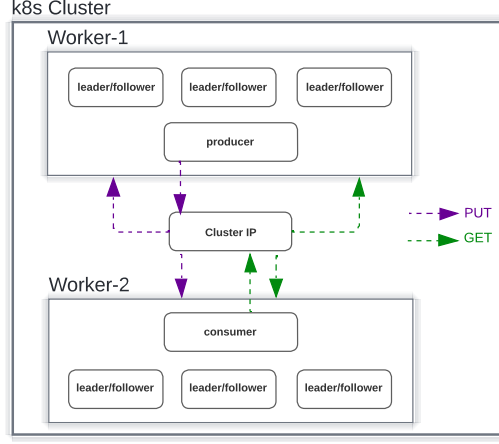


Figure 5.22: Consumer/Producer schema with 6 etcd instances in k8s Cluster (Request for R/W goes to the Cluster IP).

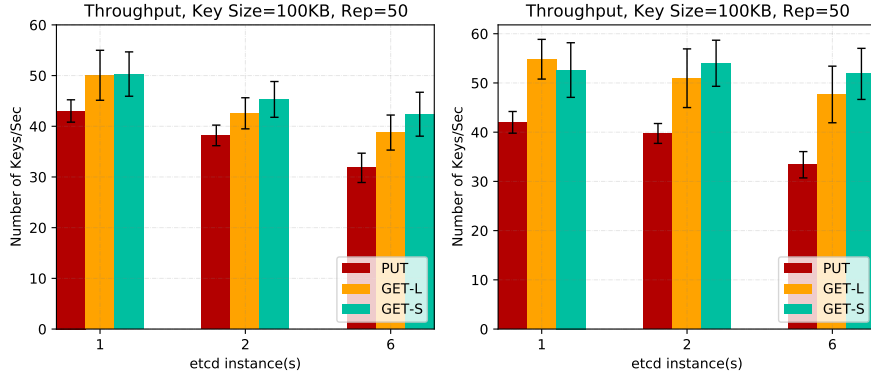


Figure 5.23: etcd operations throughput when the requests are sent to the cluster(left) and to the leader(right) with no delays in their communications.

node to another. We can assume each worker node is like an edge server. For instance, in a scenario like Proactive, we suppose that each edge server has the etcd, no need to migrate it, while in Reactive, we have to migrate both producer/consumer and etcd instance. Thus, we can comprehend the effect of migration producer/consumer on throughput in the different methodologies.

## 5.8 Realistic Networking Scenario

In this section, we experimented with comparing the throughput according to the three approaches defined for the stateless migration of microservices across different nodes(edges). The tested scenario is divided into three approaches.

In this scenario like before, we deployed the tested application in the pod, instead of

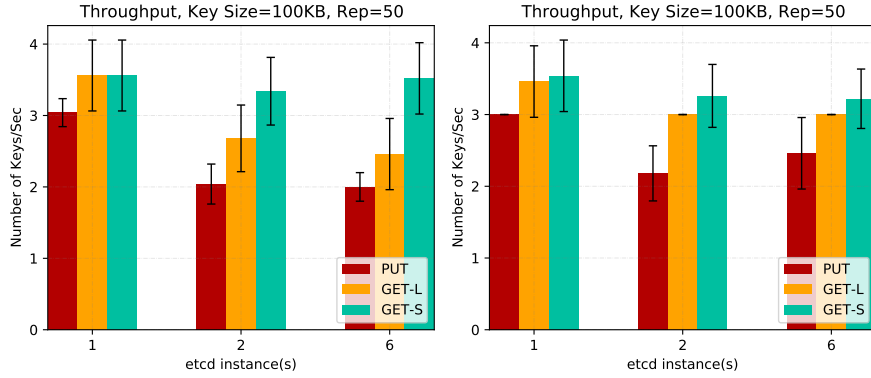


Figure 5.24: etcd operations throughput when the requests are sent to the cluster(left) and to the leader(right) with delays in their communications.

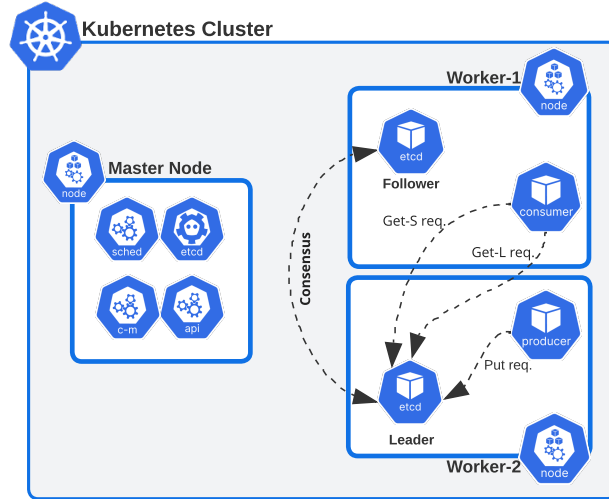


Figure 5.25: Consumer/Producer schema with 2 etcd instances in k8s Cluster (Request for R/W goes to the Leader).

having this application running on the master node to put data into the etcd. The tested application is a simple C++ program that has an infinite loop. And during a specific duration time, here as 500ms, count the number of keys that are put inside the etcd.

- Centralized, Figure 5.26, three worker nodes (worker-1, worker-2 and worker-3) involved in this approach. Here, the etcd needs to be located in one node (worker-3) as a central database. And the microservices migrated statelessly between worker-1 and worker-2. To experience realistic communication, we put 10ms delays between nodes(external delay). This is because, in the real world, there is a latency in node/edge communications. The etcd instance started on worker3, and other nodes

hosted the tested application. Once the producer and consumer are deployed, start putting the key inside the etcd and count its number. After a specific time, they moved to another worker node and started sending the requests to etcd.

This approach has pros and cons. Firstly, delays in communication in the edge can affect the outcome, meaning less throughput in comparison to other approaches. The pros are that the etcd is centralized and there is no need to migrate it since it is time and resource-consuming. The downside is to have a single point of failure. This means that, if the worker node or the etcd gets down, the data will be unavailable, causing service disruption since we have only one running etcd instance.

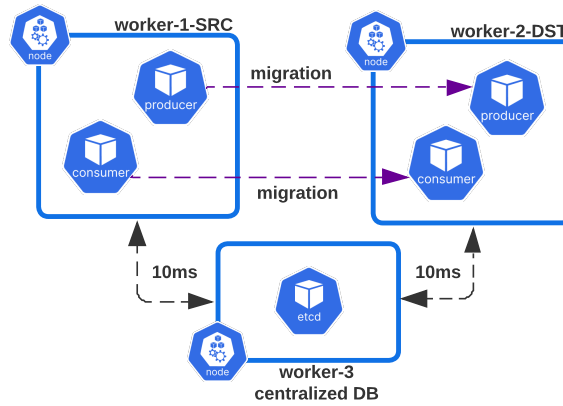


Figure 5.26: Centralized approach, migrating producer and consumer from worker-1 to worker-2, no migration needed for etcd

- Reactive, Figure 5.27, the most benefit of this approach in comparison to the previous case is that the etcd and producer both run in the same worker node. In this approach, we selected two needs to play the role (worker-1 and worker-2). In addition, we ran the sidecar along with the main pod to set a 1ms delay in the pod's communication (Internal Delay).

Both the producer, consumer, and etcd instances started to deploy on the same worker, worker-1. Then our test application put the keys on etcd and then after a certain time, the migration happened to worker-2. This approach made good progress in terms of accessing data since the etcd runs along with the producer, not in another node, like a Centralized approach. The expectation would be far higher throughput in comparison to the Centralized approach.

On the other hand, it needs time not only for migration of etcd and microservices but also for booting up both pods in the new node, fully database replicated in the source worker node. Therefore, as we expected the result shows more migration

time duration in this approach. Since they deployed in the same node, we put lower latency among their communication as 1ms by using the sidecar container alongside them.

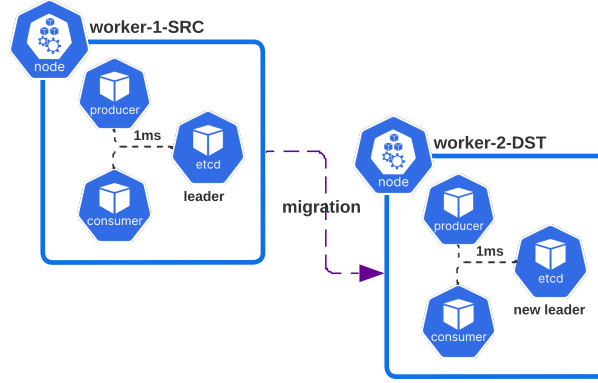


Figure 5.27: Reactive approach, migrating producer, consumer and etcd from worker-1 to worker-2 at the same time

- Proactive, Figure 5.28, in this scenario, we have four etcd instances running on four worker nodes, and they are ready to host the producer and consumer. The running etcd instances are fully replicated because of consensus among each other. The application started on worker-3 and after that they migrated to worker-4. The most benefit of this approach is that the etcd is ready to host the producer, so there is no need to consume the time for migrating the etcd itself, it's booting up, warming up, and replication process. After migration happened, since we want to always contact the leader, the process of moving the leader from the previous etcd instance to the one located in the destination needs to be done. It means that after migration, the previous leader became a follower and the new one changed into the leader.

In this case, the only migration happened only for microservices like the Centralized approach. However, like the reactive one, both the etcd instance and microservice run inside the same node. Our expectation would have more throughput than the centralized, but less in comparison to the reactive because of the consensus overhead among four etcd instances caused by RAFT algorithm.

Figure 5.29 illustrates the result of these approaches. The first scenario, Centralized, experienced less throughput because the etcd and microservices are separately deployed on different workers' nodes. So the delays between them can affect the result of putting data. It means that the number of keys per second, based on the result, is near 12 for both Reading and Writing operations. The gap inside the plots indicates the time that is the microservices migrating, so it shows the amount of time that the services are disrupted.

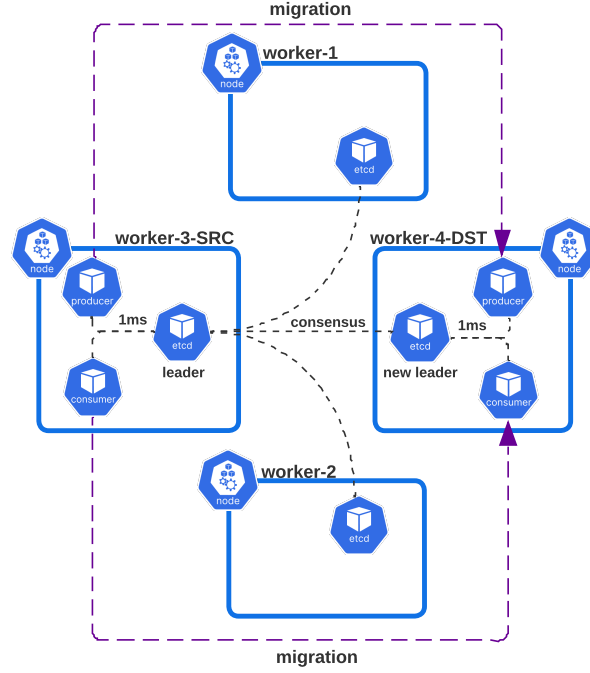


Figure 5.28: Proactive approach, migrating the producer from worker-3 to worker-4, no migration for etcd, all workers have etcd instance ready

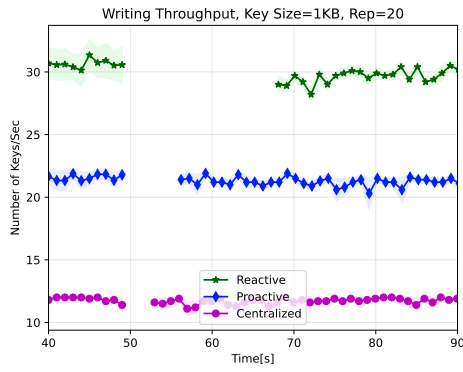
In the second case, the Reactive one produced more results, as we expected. The main reason is that both etcd and the producer/consumer worked at the same node, so there is no external delays that can affect their communication. In comparison to Proactive and Centralized approaches, this one takes more time for the migration duration, since they (etcd and microservices) have to migrate from one worker node to another one. As we can see in the plot, it took longer time for migration. Because the time is considered as several factors, such as etcd warm-up time, etcd transfer time, etcd container boot up, microservice boot up (near 20 seconds).

In the last case, the Proactive, the throughput is more than Centralized, but less than reactive, because in this case, four etcd instances are running in different worker nodes, so we need to consider the time of consensus among the instances. Consistency requires time for getting the quorum among all members, especially for operations like put and linearizable get. However, the service disruption is similar to the Centralized since there is no need to move the etcd between nodes.

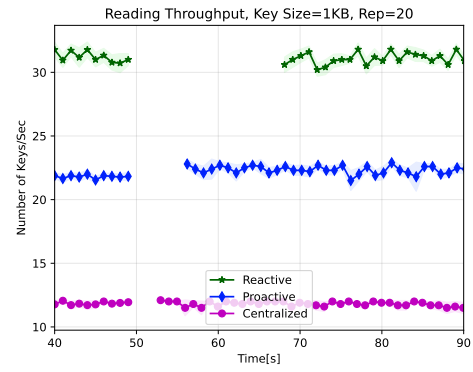
We implemented a realistic testing scenario to highlight their performance tradeoffs. In the **Centralized** approach, although the etcd is central in one host without needing to migrate, with less time disruption, there is a single point of failure. In **Reactive**, the etcd keeps inside the same host as the microservice, which is ideal for resource consumption, but migration time lasts longer. Because it takes time to boot up and warm up both

microservices and etcd, waiting for fully replicating etcd state. Lastly, in **Proactive**, we have many nodes having etcd instances running in each which is necessary without needing to migrate the state that guarantees quick migration like centralized, but we have high consensus overhead than the reactive one.

In each approach, we sacrifice one benefit to gain another one. If we need better migration duration time, centralized and proactive would be the main option. However, we have to consider the cons of a single point of failure, network latency, and high resource consumption respectively. On the other hand, if we need to have more throughput, the reactive would be the one option to achieve this, but we sacrifice the migration duration time.



(a) Throughput of PUT operation, generated by Producer microservice before and after migration



(b) Throughput of GET/L operation, generated by Consumer microservice before and after migration

Figure 5.29: Throughput of Read and Write operations





## Chapter 6

# Conclusion and Future Work

Since some microservices rely on the internal state, stateless migration leads to service disruption. The objective is to integrate the benefits of stateless migration with distributing etcd across the cluster. In this case, we are aware of the state; however, we do not care about the costly operations that happen in stateful migration, like involving CPU context state and network connections.

To accomplish this purpose as state distribution, we examine various techniques based on an insured level of consistency and propose a performance evaluation. In this case, we evaluate the performance of etcd as the popular solution to satisfy the state distribution across the cluster via providing a strong consistency level and then compare it with OrbitDB as a technique to ensure a weaker consistency level.

In terms of etcd and OrbitDB performance, based on our experiment, the number of keys has no essential effects on latency for put, get, and reset operations. In other words, the duration per key remains steady while the overall duration increases. However, in OrbitDB, time duration depends on the number of keys and state size, which shows an increasing trend. This trend happened in PUT and DELETE operations according to their natural behavior. However, duration per key for GET operation experienced no raising based on the State Size.

Therefore, we present three architectures as Centralized, Reactive, and Proactive approaches to implement our proposal of stateless migration with preserving the state. Then, we implemented a realistic testing scenario to highlight their performance trade-offs. In the Centralized approach, although the etcd is central in one host without needing to migrate, there is a single point of failure. In Reactive, the etcd keeps inside the same host as the microservice, which is ideal for resource consumption, but migration time lasts longer. Lastly, in Proactive, we have many nodes with CPU allocation because having etcd instances in each node is necessary without needing to migrate the state that guarantees quick migration, but we have high consensus overhead. Based on our outcome, our proposal is achievable, effective, and simple to implement using only the provided features by kubernetes. In the end, we measure service disruption duration caused by migration; for instance, in Centralized and Proactive approaches, this duration is very low compared to Reactive. Although our solution is straightforward to obtain, there is an impact on the throughput of microservice because of distributed state, we need

to consider realistic network conditions and the overhead introduced by the consensus algorithm to ensure state distribution.

The proposed solution can be modified to use CRDT (Conflict-free replicated data type) to enhance our throughput. Since etcd ensures consistency among members by using the RAFT algorithm, we experienced low throughput because of consensus overhead in operations like PUT and Get/L. Strongly consistent replication allows powerful consistency models such as serializable transactions and linearizability. Nonetheless, remaining for this coordination diminishes the performance of the system. The CRDT is used in systems with optimistic replication. It ensures that no matter what data modifications are made on different replicas, the data can always be merged into a consistent state. The desirable quality of using CRDT in the distribution system is to make the ordering less essential because of performing the replication as commutative operations. Providing an arbitrary order for the replication leads to reframing the race conditions in many distributed systems.

# Bibliography

- [1] A. Ahmed, A. Mohan, G. Cooperman, and G. Pierre. Docker container deployment in distributed fog infrastructures with checkpoint/restart. *IEEE*, 2020.
- [2] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Wareld. Live migration of virtual machines. *USENIX*, 2005.
- [3] Luca Conforti, Antonio Virdis, Carlo Puliafito, and Enzo Mingozzi. Extending the quic protocol to support live container migration at the edge. *IEEE*, 2021.
- [4] CRDT. About crdts. <https://crdt.tech/>.
- [5] CRIU. Tcp connection. [https://criu.org/TCP\\_connection](https://criu.org/TCP_connection), 2020.
- [6] CrownLabs. Crownlabs polito. <https://crownlabs.polito.it>.
- [7] R. Cziva, C. Anagnostopoulos, and D. P. Pazaros. Dynamic, latency-optimal vnf placement at the network edge. *IEEE*, pages 693–701, 2018.
- [8] docker. Use containers to build, share and run your applications. <https://www.docker.com/resources/what-container/>.
- [9] etcd. Kv api guarantees. [https://etcd.io/docs/v3.6/learning/api\\_guarantees/](https://etcd.io/docs/v3.6/learning/api_guarantees/).
- [10] etcd. etcd releases. <https://github.com/etcd-io/etcd/releases>, 2022.
- [11] etcd Maintenance. Maintenance. <https://etcd.io/docs/v3.2/op-guide/maintenance>.
- [12] etcd Performance. Performance. <https://etcd.io/docs/v3.2/op-guide/performance>.
- [13] J. Halpern and C. Pignataro. Service function chaining (sfc) architecture. *Internet Engineering Task Force*, October 2015.
- [14] Eric Harney, Sebastien Goasguen, Jim Martin, Mike Murphy, and Mike Westall. The efficacy of live virtual machine migrations over the internet. *IEEE*, 2007.
- [15] TianZhang He and Rajkumar Buyya. A taxonomy of live migration management in cloud computing. *Distributed, Parallel, and Cluster Computing*, 2021.
- [16] TianZhang He, Adel N. Toosi, and Rajkumar Buyya. Sla-aware multiple migration planning and scheduling in sdn-nfv-enabled clouds. *Systems and Software*, January 2021.
- [17] helm. Installing helm. <https://helm.sh/docs/intro/install/>.
- [18] Yun Chao Hu, Milan Patel, Dario Sabella, Nurit Sprecher, and Valerie Young. Mobile edge computing a key technology towards 5g. *European Telecommunications Standards Institute*, 2015.
- [19] iPerf3. What is iperf / iperf3 ? <https://iperf.fr>.
- [20] IPFS. Ipfs docs. <https://docs.ipfs.tech/>.

- [21] jepsen. Linearizability. <https://jepsen.io/consistency/models/linearizable>.
- [22] jepsen. Serializability. <https://jepsen.io/consistency/models/serializable>.
- [23] juniper. What is multi-access edge computing? <https://www.juniper.net/us/en/research-topics/what-is-multi-access-edge-computing.html>.
- [24] kubernetes. busy box. <https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale-walkthrough>.
- [25] Kubernetes. Kubernetes python client. <https://github.com/kubernetes-client/python>.
- [26] Kubernetes. Kubernetes. <https://kubernetes.io/docs/home>, 2022.
- [27] Edward A. Lee, Soroush Bateni, Shaokai Lin, Marten Lohstroh, and Christian Menard. Quantifying and generalizing the cap theorem. *Distributed, Parallel, and Cluster Computing*, 2021.
- [28] libp2p. libp2p documentation portal. <https://libp2p.io/>.
- [29] local-path provisioner. local-path-provisioner. <https://github.com/rancher/local-path-provisioner>.
- [30] V. Mann, A. Vishnoi, K. Kannan, and S. Kalyanaraman. Crossroads: Seamless vm mobility across data centers through software defined networking. *IEEE*, 2012.
- [31] Linux manual page. Linux manual page. <https://man7.org/linux/man-pages/man8/tc.8.html>.
- [32] OrbitDB. Orbitdb. <https://github.com/orbitdb/orbit-db>.
- [33] Etcd packaged by Bitnami. Etcd packaged by bitnami. <https://artifacthub.io/packages/helm/bitnami/etcd>.
- [34] Raft. The raft consensus algorithm. <https://raft.github.io/>.
- [35] redis. Getting started with redis. <https://redis.io/docs/getting-started/>.
- [36] M. Satyanarayanan. The emergence of edge computing. *IEEE*, 2017.
- [37] P. Sharma, L. Chaufournier, P. Shenoy, and Y. C. Tay. Containers and virtual machines at scale: A comparative study. *ACM*, November 2016.
- [38] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE*, 2016.
- [39] Hiroki Watanabe, Ryo Yasumori, Takao Kondo, Ken Kumakura, Keisuke Mae-sako, Liang Zhang, Yusuke Inagaki, and Fumio Teraoka. Contmec: An architecture of multi-access edge computing for offloading container-based mobile applications. *IEEE*, 2022.
- [40] wikipedia. Persistent data structure. [https://en.wikipedia.org/wiki/Persistent\\_data\\_structure](https://en.wikipedia.org/wiki/Persistent_data_structure).
- [41] wtcd kluster. etcd-cluster. <https://etcd.io/docs/v3.4/op-guide/clustering>.
- [42] xenonstack. Stateful and stateless applications and its best practices. <https://www.xenonstack.com/insights/stateful-and-stateless-applications>.