UNIVERSITY OF TARTU
Faculty of Science and Technology
Institute of Computer Science
Computer Science Curriculum

Thi Song Huong Pham

Fault-Tolerant Distributed Database for Public Key Infrastructure

Master's Thesis (30 ECTS)

Supervisor(s): Pelle Jakovits, PhD
Matti Siekkinen, PhD
Advisor(s): Bruno Duarte Coscia, M.Sc

Tartu 2024

Fault-Tolerant Distributed Database for Public Key Infrastructure

Abstract:
This thesis presents a practical study focusing on improving service level availability (SLA) and reducing maintenance costs of distributed SQL database clusters by migrating from a traditional manually-managed virtual machine (VM) architecture to a Kubernetes-based one. The original architecture suffers from complex manual recovery processes during node malfunctions, leading to increased downtime. Three failure modes inherent to the original architecture are investigated: 1. primary node failure; 2. replica node failure; 3. split-brain mode.

The proposed architecture leverages Kubernetes to automate cluster recovery from primary and replica node failures, significantly reducing downtime compared to manual processes. Furthermore, Kubernetes' architectural features eliminate split-brain mode entirely.

Keywords:
Kubernetes, Cloud Computing

CERCS: P170

Tõrketaluv hajutatud andmebaas avaliku võtme infrastruktuuri jaoks

Lühikokkuvõte:

See väitekiri esitab praktilise uuringu, mis keskendub teenindustaseme kättesaadavuse (SLA) parandamisele ja hajutatud SQL andmebaasiklasterite hoolduskulude vähendamisele, migreerides traditsiooniliselt käsitsi hallatavast virtuaalmasina (VM) arhitektuurist Kubernetesel põhinevale. Varasemalt kasutusel oleva arhitektuuri peamine probleem on kererukate käsitsi tehtavate taastamisprotsesside vajadus, mis viib suurte seisakuaegadeni. Uuritakse kolme algsele arhitektuurile omast rikete tüüpi: 1. peamise sõlme rike; 2. koopia sõlme rike; 3. lõhestunud aju (split-brain) olekut.

Töös välja pakutud arhitektuur kasutab Kubernetest, et automatiseerida klastri taastumist peamise või replikaat sõlme riketest, oluliselt vähendades seisakuaega võrreldes käsitsi tehtavate protsessidega. Lisaks kõrvaldab Kubernetese arhitektuur täielikult lõhestunud aju oleku tekkimise võimaluse.

Võtmesõnad:
Kubernetes, pilvearvutus

CERCS: P170

Contents

1 Introduction

Public Key Infrastructure PKI is an operational system that employs asymmetric cryptography, information technology, operating rules, physical and logical security, and legal matters. Big organizations have their private PKI servers in order to sign their own applications and issue internal certificates. For such organizations, PKI is one of their core services since its failure could risk downtime of other services, compromising business continuity. Moreover, exposure of private keys could allow attackers to impersonate an organization's identity, for example to fraudulently sign applications. Therefore, PKI components need to have strict security protection. The 3 relevant segments of PKI sec are Confidentiality, Integrity, Availability, which is also CIA triad [19].

This thesis initially highlights an issue within the PKI architecture of a specific organization, specifically addressing its impact on the fault tolerance of the system. For instance, it explores challenges like the unavoidable occurrence of split-brain scenarios in organizations with only two sites, as well as the complications associated with manual intervention during primary switching events.

Since the database is a key component of the PKI system, this thesis focuses on the benefits of transitioning databases from virtual machines to Kubernetes, an orchestrated environment. This analysis is relevant to all high-availability systems.

2 Motivation

The research is driven by the emergence of Kubernetes, a cluster orchestration tool, and its operator resource, which offers the potential to automatically recover clusters without manual intervention during node failures. Additionally, it presents an opportunity to improve the split-brain scenario within the organizational use-case.

Split-brain is the scenario when a network partition causes the database to become partitioned into multiple independent instances that are unaware of each other's state. This situation can occur in distributed database systems where multiple nodes are interconnected, and communication between them is interrupted. In the organization case, split-brain scenario happens to the organization because it only has 2 sites.

Unlike traditional methods that depend on quorum for primary node selection, Kubernetes operator facilitates centralized election management, thereby preventing the occurrence of multiple primary pods within a cluster simultaneously. This study explores whether Kubernetes operator based approach would be capable of addressing this issue, as well as conducts experiments to evaluate the cluster's ability to self-heal following primary and replica node failures.

3 Research question and contribution

This thesis identifies challenges within the organization's specific use-case and examines state-of-the-art potential solutions. It progresses to assess whether Kubernetes can effectively address these challenges through a detailed study of Kubernetes and its operators, subsequently proposing a Kubernetes-based architecture for the application. To validate the efficacy of this design, the thesis conducts experiments designed to answer the following research questions:

1. How Kubernetes (operator) improves the availability of the cluster during database node failure. Particularly, what is the recovery time for read and write operations in the event of:
   (a) a primary node failure
   (b) a replica node failure
2. How can the split-brain problem in high-availability systems be alleviated with Kubernetes?

This thesis is structured as follows: it begins with a background study of Kubernetes, followed by an examination of the current architecture's challenges. Subsequently, it presents the proposed architecture leveraging Kubernetes. The thesis then evaluates the effectiveness of this proposed architecture through experimentation. Finally, it concludes with a discussion on potential future work aimed at enhancing the operator.

The thesis findings reveal that the new architecture, utilizing a Kubernetes operator, allows the cluster to automatically recover, eliminating the manual intervention required in the current setup. Additionally, the operator inherent mechanism of active/active configuration effectively addresses the critical issue of split-brain failure mode.

4 Background

4.1 Kubernetes

As the demand for running containers not only for development purposes but also for production increases, so is the need for a tool to manage these containers in a large scale manner. Kubernetes is designed to solve this problem and while there are several approaches to this problem such as Docker Swarm, Kubernetes is a success because of its design principles that allow easy integration (migrating to Kubernetes doesn't require code rewrite) and customization (users could write operators for their own applications).

4.1.1 Kubernetes design principles

Kubernetes API are declarative rather than imperative — In Docker, the deployment process is done by the developers gaining remote access to the server, starting the container and deploying it by himself. Hence, the process has to be repeated by the developer when he wants to deploy the container again. This process is called imperative. In Kubernetes, the developer only needs to define the desired state and lets Kubernetes find out how to reach that state. This style of configuration is called declarative and is similar to the way infrastructure-as-code tools such as Ansible and Terraform works. This not only saves developers a lot of time, but also better guarantees that the container is deployed correctly.

There are no internal hidden APIs in Kubernetes control plane. — We tend to think the control plane is the brain of the cluster, so it requires a lot of complex logic in order to make all decisions. However, giving the control plane too many responsibilities will introduce a central point of failure, as the cluster couldn't function correctly if the master node isn't available. Instead, what they do in Kubernetes is to store only the desired states of pods inside the master node. Worker nodes constantly watch the API server to see compare between their current state and the desired state. If the desired state is different, the worker node will work to reach that state. Additionally, all components monitor their health and report to the master node.

This approach where the system operates to reach a state is called level-triggering. Compared to edge-triggering, where a system responds to events so when the system doesn't work correctly, developers are responsible to send the events again, level-triggering has self-healing benefits. If a component crashes, when coming back up, it can look at the API server to restore the current state. Hence, there is no missing events issues.

The same API is used by internal components, is exposed to the outside. Hence, when writing an application, developers could fetch secret object, configmap

information, or information about the pod that the application is running on from API server.

Meet the user where they are — Kubernetes creators understand that not every organization is open to make modifications to their applications in order to accommodate new technologies. For example, for applications that are old but critical to the organizations such as PKI, requiring modifications in order to integrate with Kubernetes API is a major hinder to adoption because the cons of breaking the application are much greater than the pros of successfully migrating it. Hence, in order to encourage transition, it's possible to consume objects such as secrets, configmaps, downward API as files or environment variables within the containers. Information about dependent objects could be specified under pod definition, which Kubernetes will make sure to be mounted as a file or as an environment variable to the containers and hence the applications only need to be able to read from files or from environment variables.

Workload portability — Kubernetes is comparable to an operating system OS in infrastructure level. In the old days, applications were written to work on specific hardware. Then OS came along to remove the trouble of considering hardware specification for developers by providing interfaces. The similar case is happening in distributed system world, where instead of deploying applications to fit certain cluster implementations, Kubernetes exposes its API so that applications could be deployed against it, making Kubernetes an abstraction layer for application deployment. This approach successfully separates application development from cluster implementation.

4.1.2 Kubernetes components

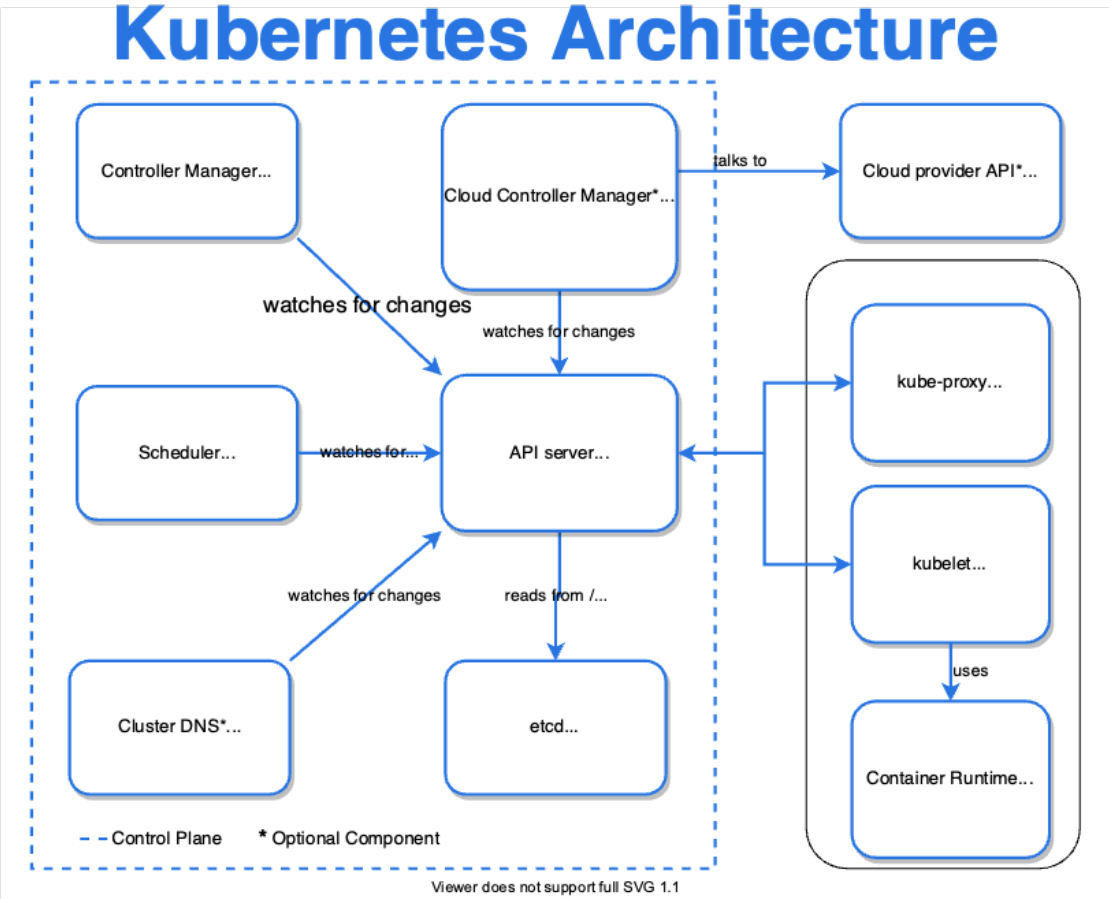

Figure 1. Kubernetes architecture [18]

Core components of a Kubernetes cluster:

etcd — key-value store inside control plane to store the components' states.

api-server — the most important component of the cluster because all interactions from outside or from other components to the cluster go through api-server.

scheduler — the scheduler monitors the kube api server looking for unscheduled pods. These are API objects of type pods that don't have the Node field filled in. So it starts to look for the best placement for that object, then update the API server accordingly.

controller-manager — there needs to be a component to continuously watch the API server for the shared states in order to bring the current state towards the desired

ones. Controller manager is such component, that operates as a non-terminating loop to regulate the state of the system. [14]

kubelet — exists in every node to act like a communication point between node and the cluster. Kubelet is responsible to self-report node health and current states to API server for monitoring.

kube-proxy — component that runs on each node for maintaining the network rules. These rules allow communication to pods from inside or outside network.

API server — is the central component and is the only component that talks to Etcd to store and retrieve cluster states. Other components such as kube-proxy, scheduler, controller manager and kubelet are API server clients. API-server, scheduler, controller manager, and Etcd are control plane components while kube-proxy and kubelet belong to worker. [15]

4.1.3 Kubernetes Storage

In traditional bare-metal setups, storage provisioning and management were typically manual processes. Administrators had to manually allocate and configure storage volumes for individual servers, which was time-consuming, error-prone, and lacked automation. Managing persistent storage for containerized workloads on bare-metal servers required custom scripting and integration efforts. Furthermore, many storage solutions designed for cloud environment are not easily adaptable to the bare-metal environment. These problems prevent bare-metal environment to become production-ready easily. Longhorn is a storage provisioner that mitigates such problem by enabling Kubernetes to provision storage dynamically.

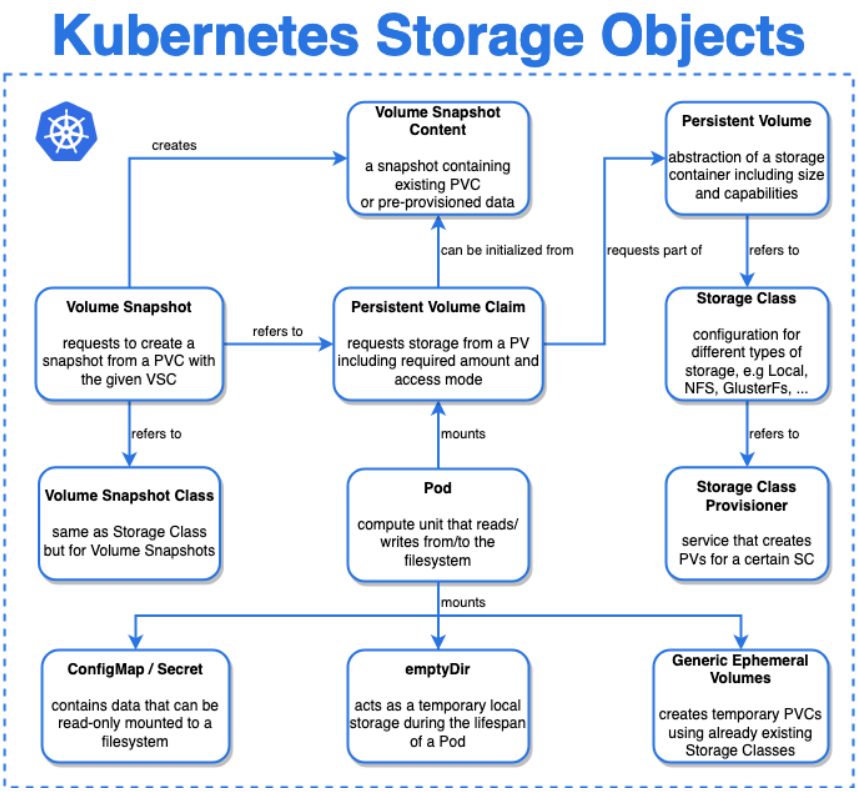

Figure 2. Kubernetes storage [17]

Figure 2 shows the relationship between Storage Provisioner (here is Longhorn) with the Persistent Volume (PV) and Persistent Volume Claim (PVC). Through the CSI integration, when a PVC is created in Kubernetes, Longhorn dynamically provisions a PV to meet the claim. This process involves creating a new volume within Longhorn, which automatically becomes available to Kubernetes for binding to the PVC.

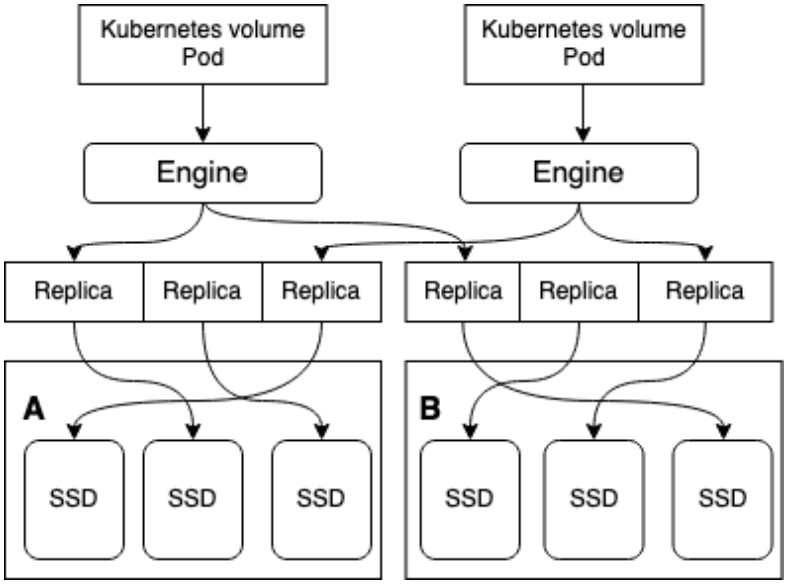

Figure 3. Longhorn data replication [1]

For each volume, Longhorn creates multiple replicas across different nodes based on the specified replication factor. This replication ensures that even if a node fails, the volume can continue operating using the remaining replicas. Longhorn's engine manages the replication logic, keeping the data consistent across replicas.

Longhorn allows for creating volume snapshots, which are point-in-time copies of the volume. Snapshots can be used for quick recovery and cloning volumes. They are stored locally with the volume's data.

Unlike snapshots, backups are stored in secondary storage, which can be off-cluster (like S3 or NFS). Backups provide a way to recover data in case of severe failures, including total cluster loss.

If a replica fails (due to node failure or disk issues), Longhorn automatically initiates a rebuilding process for the affected volume by creating a new replica on a healthy node. This process ensures that the volume returns to its desired state of replication and availability without manual intervention.

4.1.4 Database Replication

One motivating factor for transitioning to Kubernetes is its robust replica storage capabilities. While lightweight virtualization technologies, such as containerization, address the performance-reliability trade-off in fog computing and offer built-in fault tolerance mechanisms, they often fall short in providing fault-tolerant persistent storage [3]. Initial reservations about running databases in Kubernetes arose due to perceived limitations in handling stateful storage. Several potential failures that could cause state loss [9]:

- **Node Failure**: A node can completely fail due to disconnection or any hardware failure. [9]
- **Application Execution Failure**: The application stops working due to an undefined internal reason.[9]
- **Application Deployment Failure**: The application is not correctly initially deployed or re-deployed. Several reasons are possible, such as no available resources, losing communication to available nodes, or improper resource management.[9]
- **File Access Failure**: Applications fail to access required files, states, and other necessary data to continue working. This encompasses the loss of volumes in pods, e.g., due to Node Failure.[9]
- **Management failure**: This includes all failures caused by functions required to run a container environment.[9]

In database, state persistence needs to be replicated in multiple locations by either following strategies:

1. **storage-level replication** Kubernetes native database replaces cloud native database [13]

   In the past, distributed storage could only achieve by using cloud environment such as Amazon S3 or vSAN. With Kubernetes, it's now possible to have distributed block storage for bare-metal environment. Hence, organizations could now enjoy the high availability that cloud databases offer while still having the fault tolerance of distributed block storage inside their premises. Because of this feature, the rise of Kubernetes native databases that used to live outside Kubernetes such as Apache Cassandra and DynamoDB are now adopting to the new environment.

2. **application-level replication** Differ ent databases have different ways to achieve high availability through replication. Here, CloudnativePG way is focus as it's the solution used for the implementation.

   Physical data replication is Postgres database main way to achieve high availability. Replication also allows scale of read-only workloads to offload work from the primary node. [7]

4.2 Kubernetes Operators

Although Kubernetes offers the advantage of scalability to users, this also introduces increased complexity and a heavier maintenance burden [8]. Moreover, the distributed nature of these workloads inherently increases the number of potential points of failure, which can result in costly and time-consuming repairs [8].

The introduction of Kubernetes operators has shifted this paradigm, enabling operational knowledge that was confined to administrators, shell scripts, and declarative configuration tools like Ansible and Terraform to be embedded directly into software. An operator extends Kubernetes by automating the entire life cycle management of a particular application, including monitoring, maintenance, recovery, and upgrades. They serve as a sophisticated packaging mechanism for distributing applications on Kubernetes, ensuring that applications are not only deployed but also continuously optimized according to the dynamic conditions of the cloud environment. This integration of operators into Kubernetes represents a significant evolution in the deployment of distributed applications on containerized platforms. It underscores the widespread acceptance and adoption of such tools in state-of-the-art cloud orchestration environments, marking a pivotal shift towards more autonomous cloud management solutions [11].

The operator framework defines five levels of operator maturity like below.

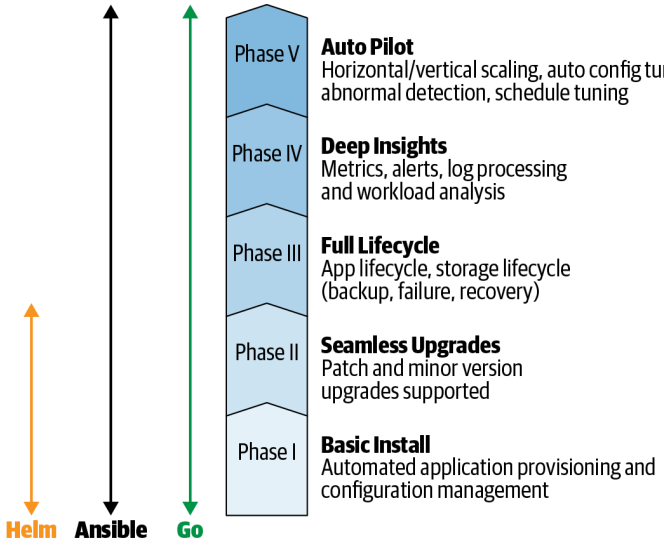

Figure 4. Operator capability level [10]

4.2.1 CloudNativePG operator

Cloudnative-PG represents a Kubernetes-native database operator that could potentially replace traditional cloud-native databases. In terms of operator capability, it has reached level 5 of autopilot. Key features relevant to this study include replication and automatic failover

1. Replication

   Synchronous replication: To further prevent data inconsistencies, CloudNativePG supports configuring synchronous replication. In synchronous replication, the primary waits for one or more replicas to confirm that they have received and

applied a write operation before the operation is considered complete. This ensures data consistency but may impact write performance.

2. Automatic Failover [6]

   When the readiness probe on the primary fails, the controller's reconciliation loop automatically detects it and the operator immediately elects the new primary among the available replicas. The operator ensures that all nodes are aware of the new primary to prevent multiple primaries from emerging.

   When the old primary comes back, the instance manager detects it, preventing split-brain from happening. Once the pod is ready, the operator assigns it to the read-only replica.

   In the event of unexpected errors lasting longer than the specified failover delay, the PostgreSQL cluster initiates failover. This could occur due to disk failure, pod deletion, or sustained failures in the primary pod's PostgreSQL container. During failover, the readiness probe for the primary pod fails, triggering the controller's reconciliation loop.

   The failover process occurs in two steps:

   (a) Pending marking of target primary: The controller marks the target primary as pending, forcing the primary pod to shut down. This ensures the Write-Ahead Logging (WAL) receivers on replicas stop. The cluster enters the failover phase.

   (b) Leader election and promotion: Once WAL receivers stop, a leader election takes place, and a new primary is named. The chosen instance initiates promotion to primary, and normal cluster operations resume. The former primary pod restarts as a replica.

   During the shutdown of the failing primary:

   (a) A fast shutdown is attempted with a specified switchover delay, attempting to archive pending WALs.

   (b) If fast shutdown fails or times out, an immediate shutdown is initiated, aborting all PostgreSQL server processes immediately.

   During the primary's failure and before failover starts, queries, WAL writes, and checkpoints may fail. Fast shutdown stops connections but preserves data. If it fails, immediate shutdown may lead to data loss. While the new primary hasn't started, the cluster operates without a primary, impairing but not losing data. The switchover delay affects recovery time objective (RTO) and recovery point

objective (RPO). A low value may prioritize RTO but risk data loss, while a high value minimizes data loss but delays normal operation.

A delayed failover option exists to postpone failover after detecting primary unhealthiness, preventing premature failover due to short-lived network or node instability. The default failover delay is set to 0, triggering immediate failover.

4.3 Organization use case

A specific organization relies on a critical application where consistency and availability are crucial. To enhance resilience, the organization deploys a database cluster spanning two sites, each hosting three instances. When a host returns from a shutdown state, another host aids in synchronizing its state, while the remaining one continues serving requests.

To enable an instance to handle read or write requests, it must confirm its ability to communicate with a majority of nodes in the current cluster. Nodes stay in touch, determining the size of the network partition they can reach. This strategy ensures that if one site is lost, the other can function normally as long as it has the majority of instances. However, if both sites lose connection, both sites continue accepting requests, leading to the split-brain problem.

Figure 5 shows the present setup of the application. Each site contains 2 application instances and 3 database instances, which synchronize via a high bandwidth connection.

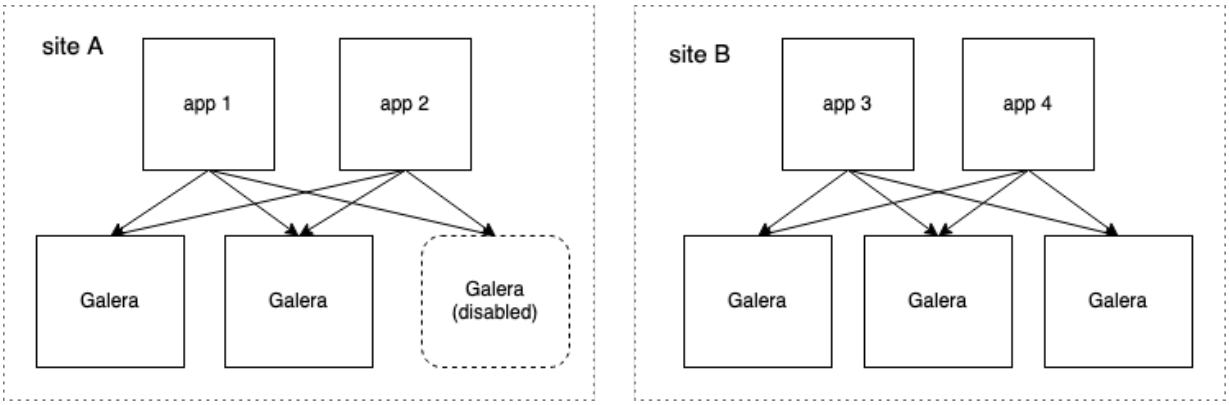

Figure 5. Current use case

To prevent split-brain in this application, the organization decides to disable one instance on one site. Yet, during an outage or maintenance in site A, site B can't automatically assume the primary role as it has only two out of five instances. In order for site B to assume the primary role, the organization needs to flip the secondary and primary states manually by turning off one Galera node on the primary site and activating

the spare Galera node on the secondary site. This process effectively prevents split-brain, albeit requiring manual intervention.

Hence, during site A's outage, the service remains non-operational until manual intervention occurs. While this setup mitigates the split-brain problem by preventing outdated data from being served, it compromises service availability.

4.3.1 Galera and split-brain failure mode

Galera is implemented as a multi-master cluster for MySQL and MariaDB [2]. In this architecture, all nodes are fully operational, enabling them to handle write requests independently. This feature enhances high availability by eliminating single points of failure and enabling the scaling of write operations.

The split-brain failure mode arises when database instances operate independently of one another. In such scenarios, data corruption can occur as two separate database nodes may make conflicting updates to the same data row in a table. As is the case with any quorum-based system, Galera Cluster is subject to these conditions when the quorum algorithm fails to select a primary component [12].

To prevent split-brain issues, Galera requires that each transaction receive a majority of acknowledgments from other nodes before it is committed. If a network partition occurs such that no single group holds a majority, the cluster will halt all write operations to maintain data consistency until a majority is reestablished.

5 Proposed solution

5.1 General approach

The main problem that this work aims to address is that the current design requires manual intervention to address node failures. This chapter outlines how Kubernetes and related technologies can be efficiently applied to automate most of the failure recovery scenarios that are of interest to the application at hand.

The split-brain failure mode that affects the existing cluster design is addressed with the help of the CloudNativePG postgres database operator that is able to seamlessly hand off the primary role from one node in the cluster to another such that the handover process is transparent to the client.

The delegation of the primary election function to the CloudNativePG operator is not expected to affect the reliability of the cluster as from the standpoint of Kubernetes itself, the operator itself is an ordinary application executed in a separate Kubernetes pod. In the event of a failure of the pod that executes the operator, the self-healing capabilities of Kubernetes will automatically reschedule the operator on a new pod, thus avoiding the single point of failure in the cluster[1].

Failure modes related to a cluster node failure (i.e., disappearance of a primary or a secondary node from the database cluster) are addressed by the aforementioned self-healing capabilities similarly, with the exception that the "healing" action is performing not by the Kubernetes core itself but by the CloudNativePG operator.

The system outlined above is extended with several auxiliary components that are inessential for the purposes of this high-level overview. The resulting capabilities include instantaneous and fully automatic recovery from all relevant failure modes. The following sections provide an in-depth technical review of the resulting system.

5.2 Cluster architecture

This section aims to describe the optimal architecture for a two-zone setup given the present conditions. It will cover recommended configurations for CloudNativePG (the database operator used in the experiment) to balance operational efficiency with robust disaster recovery strategies.

> I not only think that we can run databases on Kubernetes because "it's fundamentally the same as running a database on a VM", but I believe that through the CloudNativePG operator we all can run Postgres databases better than on a VM [4].
>
> *Gabriele Bartolini - VP of Cloud Native at EDB*

[1] Kubernetes itself is a distributed fault-tolerant peer system devoid of a single point of failure; see chapter 4 for the background.

Kubernetes is designed to run across data centers given that these centers are linked by high bandwidth, low latency, and redundant networking [4]. In this case of stretched cluster, the recommendation is that the latencies between clusters shouldn't exceed 5 milliseconds (ms) round-trip time (RTT) between nodes in different location, with a maximum round-trip time (RTT) of 10 ms [16].

It's recommended to use a single Kubernetes cluster across multiple availability zones instead of isolating resources in separating data centers within distinct Kubernetes clusters [4].

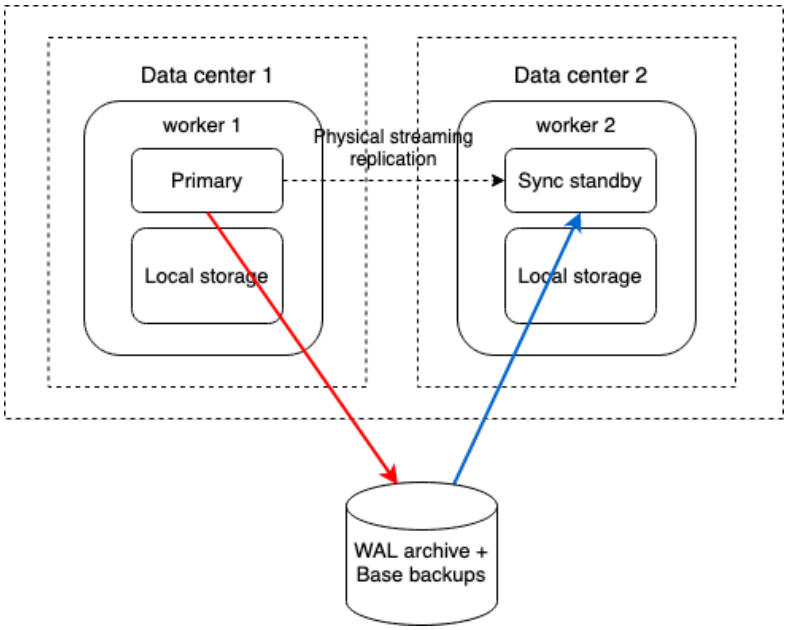

Figure 6. Single cluster over two zones [4]

The primary PostgreSQL database streams its transaction logs (WAL) to the synchronous standby in a different availability zone to ensure data consistency and high availability. In case the primary database fails, the standby can use the WAL archives and base backups from the object store to rebuild the database state up to the last consistent transaction. This helps ensure that the data on the standby is as current as possible, minimizing data loss and recovery time.

Another aspect to consider is the sharing of workloads and storage between nodes. It's recommended that the database has its own dedicated nodes instead of sharing storage with applications. This allows separation between shared storage for applications and local storage for PostgreSQL. This approach is also known as shared-nothing architecture under distributed systems [4].

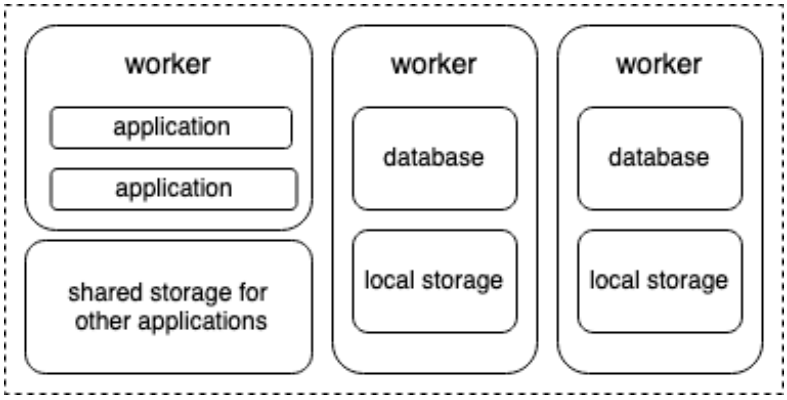

Figure 7. Shared-nothing architecture [4]

If data centers do not qualify for a stretched cluster, the last resort is that one data center serves as the primary cluster, while the other acts as disaster recovery. In such case, the operator's replica cluster is used to create a symmetrical PostgreSQL cluster in the secondary data center, which can be manually promoted to be primary when necessary [4]. One disadvantage of this setup is the lack of automatic failover, unlike stretched cluster.

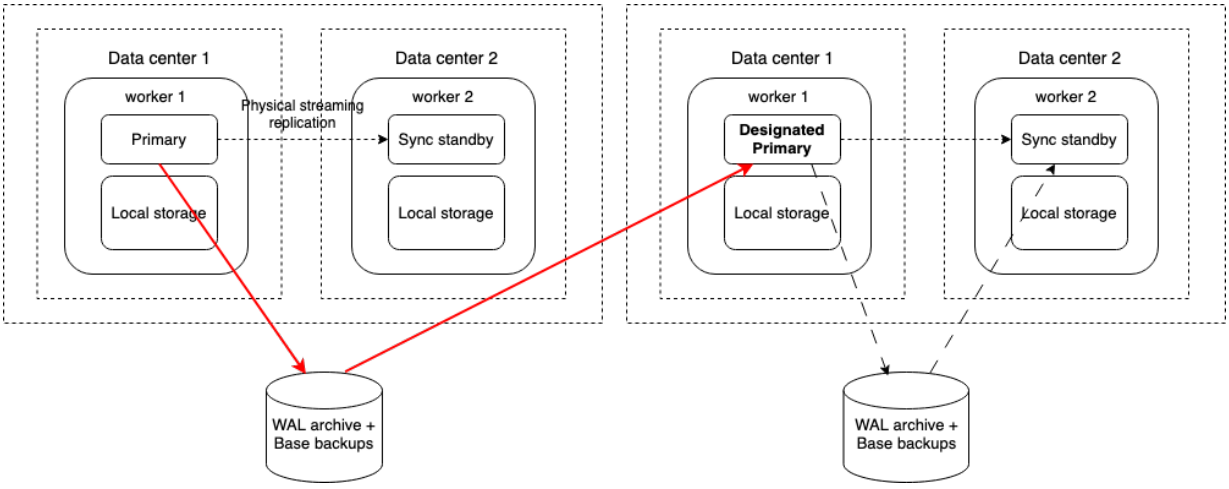

Figure 8. Replica cluster [4]

In this setup, the backup (replica) cluster mirrors the architecture of the main cluster. It includes a designated primary, which is a standby ready to take over in case of main cluster failure. Synchronization with the main cluster is maintained by retrieving WAL archives and base backups from the origin's object store, allowing a recovery point objective (RPO) of approximately 5 minutes for cross-regional disaster recovery. By

enabling streaming replication between the primary and replica clusters, the RPO can potentially be reduced to nearly zero across regions [4].

Similarly, a replica cluster could serve as an additional layer of backup for a stretched cluster scenario over multiple regions.

It's crucial to be aware that adding storage level replication doesn't increase business continuity but degrading performance because it introduces write amplification, where the number of write operations increases.

In summary, for an optimal two-zone setup, it's recommended to employ a stretched cluster across two data centers with latency not exceeding 10ms. Additionally, databases should have their own dedicated nodes to maintain isolation from application.

5.3 Comparison between organization use case and proposed architecture

This subsection compares the use of active/active setup in Galera with the active/passive setup of Postgres operator as used in the proposed architecture.

Active/active refers to Galera multi-master property, where all nodes accept write requests while active/passive refers to Postgres one primary node active at a time for writing operations.

In Galera, even though it operates in an active/active configuration, the changes made are not immediately committed across the database until a sufficient quorum is achieved. This quorum mechanism is what protects the cluster from having split-brain failure. However, this also means that if the number of nodes falls below the quorum threshold, the cluster becomes inaccessible until enough nodes are available again to meet the quorum requirement.

In Postgres operator, because it's an active/passive setup, the failover process is initiated when the liveness probe detects a failure in the master node. This also means that even if only one node remains operational, the cluster can continue functioning by promoting the surviving replica to become the new master.

Another difference is that Galera utilizes a proprietary method called certification-based replication, while PostgreSQL relies on its native streaming replication. When PostgreSQL is deployed within Kubernetes using CloudnativePG, it also benefits from Raft, which helps ensure consistent orchestration and scheduling of pods, services, and other Kubernetes resources.

6 Evaluation

6.1 Experiment design

In this setup, a Kubernetes cluster consisted of 1 master and 3 workers was deployed in a private cloud environment. The host information of the VMs are presented in table 1, while table 2 presents the chosen technology and their purposes.

Host Name	Role	System	Core
master1	control plane, etcd	CentOS 9, 100GB, 8G Memory	4 vCPUs
worker1	worker	CentOS 9, 100GB, 8G Memory	4 vCPUs
worker2	worker	CentOS 9, 100GB, 8G Memory	4 vCPUs
worker3	worker	CentOS 9, 100GB, 8G Memory	4 vCPUs

Table 1. Hosts information

Technology	Purpose
CloudNativePG Operator	Highly available Postgres database
Cilium	Container networking
Longhorn	Block storage for persistent storage
MetalLB	L2/L3 load balancing
Traefik	Reverse proxy also known as ingress controller
kubeadm	Provision Kubernetes master nodes

Table 2. Technology and its purposes

For these experiments, the Longhorn storage operator's replica count is adjusted to 1 instead of the default of 3. As the experiment will involve intentionally create database failure, having 3 replicas as default could slow down the time of the cluster responding to failure because Longhorn needs to delete and recreate 3 times number of storage for each instance.

In practice, the number of copies of data depends on:

1. replication at CSI layer (Longhorn)
2. replication at database layer (CloudnativePG)
3. RAID or regional replication at the block storage layer

The Kubernetes manifests for Postgres cluster and Longhorn adjustment could be found in Appendix B, while the python script used to connect to the cluster is described in Appendix C.

```
# kubectl get pods
NAME                  READY   STATUS    AGE   IP           NODE
pod/cnpg-cluster-1    1/1     Running   12d   10.0.3.51    worker3
pod/cnpg-cluster-2    1/1     Running   12d   10.0.2.64    worker2
pod/cnpg-cluster-3    1/1     Running   12d   10.0.1.199   worker1

# kubectl get endpoints
NAME              ENDPOINTS                                          AGE
cnpg-cluster-r    10.0.1.199:5432,10.0.2.64:5432,10.0.3.51:5432      12d
cnpg-cluster-ro   10.0.1.199:5432,10.0.2.64:5432                     12d
cnpg-cluster-rw   10.0.3.51:5432                                     12d
kubernetes        172.17.89.167:6443                                 45d
```

6.1.1 Working mechanism overview

The purpose of this subsection is to explain how the Kubernetes components used in the experiment works together. The photo below describes a working cluster composed of the similar components

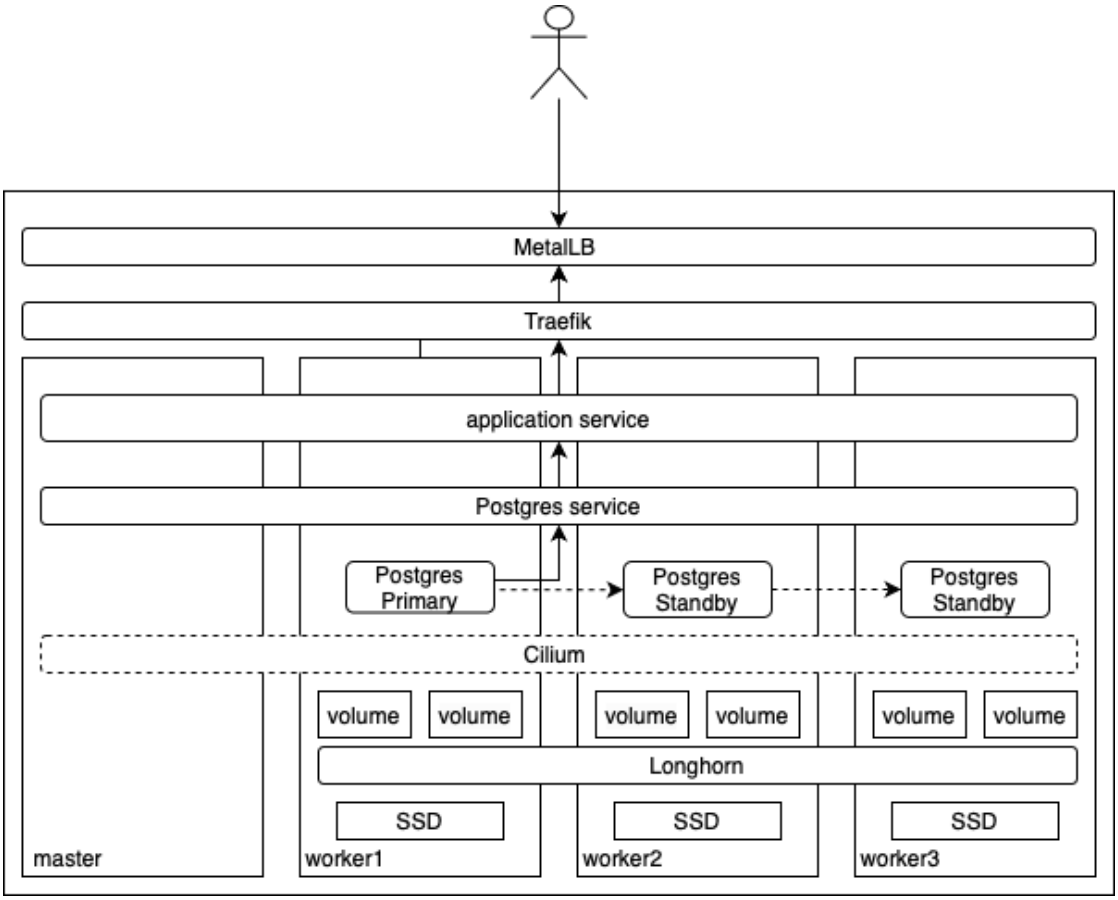

Figure 9. Experiment implementation

On the master node, the cluster is first created with kubeadm. Kubeadm is the command line tool that helps initialize a production-ready Kubernetes cluster by installing basic components of the master node as described in Background Kubernetes core components part 4.1.2.

After creating the primary components, the cluster needs a networking layer to serve the internal communication between them. Cilium serves as the layer 2 technology facilitating such communication.

Next, the cluster needs a storage provisioner to persist the application data and provide replication for the distributed system. Longhorn is a distributed block storage solution for bare-metal cluster. Previously, it has been a challenge to host an application for bare-metal setup. Longhorn's ability to provide replication and dynamic storage provisioner helps bare-metal environment achieve fault-tolerance just like cloud solutions. More about Longhorn replication is described in 4.1.3.

After installing the networking and storage layers, the worker nodes could then join the cluster by kubeadm. All the commands and configuration file for the cluster is provided in the Appendix A.

MetalLB serves as the network load balancer, acting as a reverse proxy for the cluster and exposing the service of a bare-metal cluster to the network. When a client sends a request to the application's external IP, MetalLB forwards the request to the Traefik ingress controller, which acts as an application-level load balancer for the cluster by routing requests to the correct endpoints.

As the request traverses the cluster's networking layer, it encounters Cilium, which provides networking, security, and observability features for Kubernetes. Cilium may enforce network policies, such as allowing or denying traffic based on predefined rules, before allowing the request to proceed further.

If the request involves storage operations, such as accessing or updating persistent data, Longhorn comes into play. Longhorn provides persistent storage for stateful applications running within the Kubernetes cluster, ensuring data replication and availability across multiple nodes for resilience and fault tolerance.

For requests involving interaction with a database managed by a Kubernetes operator, such as a database operator, the operator manages the lifecycle of the database instance. It ensures the database service is running, scales it as needed, and handles tasks like backups, restores, and configuration changes.

Finally, the request reaches the appropriate service pods within the Kubernetes cluster, where the actual application logic resides.

Upon generating a response, the response is routed back through the reverse proxy Traefik. Traefik ensures the response is sent back to the user through the appropriate route, based on the initial request's ingress configuration.

MetalLB also participates in distributing the response traffic back to the user, particularly if there are multiple replicas of the application pods across different nodes in the

cluster.

Ultimately, the response exits the Kubernetes cluster through the internet gateway and traverses back to the user.

6.1.2 Monitoring and visualization

The following tools have been used during the experiment in the attempt to monitor the behavior of the experiment. In production, these tools could also be used for monitoring and alerting of the cluster and application.

1. Prometheus

   For example, here promQL query below is used to capture the pod switching label from replica to primary label during primary database instance failure experiment

```
sum(kube_pod_status_phase{namespace=~"default", phase="Running"} * on(uid)  \
group_left(label_cnpg_io_instance_role) kube_pod_labels{}) \
by (label_cnpg_io_instance_role)
```

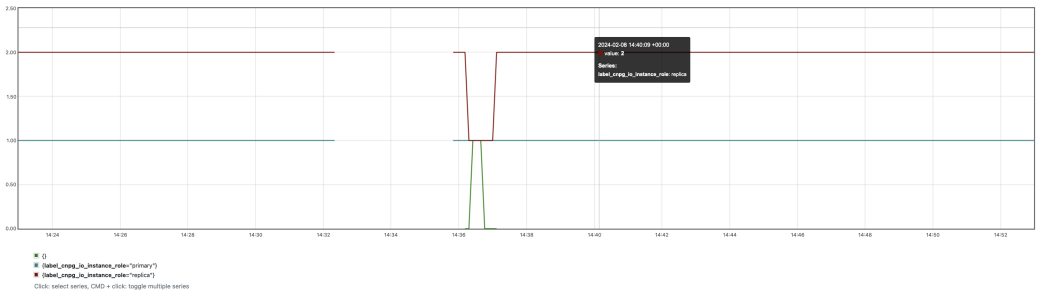

Figure 10. Pod switching labels

2. Grafana dashboard helps visually present the data originating from Prometheus. Dismissing the need for typing promQL queries to learn about the monitored target, Grafana provides preconfigured dashboards, encapsulating metrics from custom queries. For example, kube-state-metrics and CloudNativePG dashboards could be used in the case of the implementation cluster.

3. Kubernetes Dashboard is the hardest to access as it requires Role Based Access Control (RBAC) to be set up. Nonetheless, compared to Prometheus and Grafana, the Dashboard setup is quicker and demands minimal configuration, primarily RBAC for authentication.

6.2 Experiments

In order to prove that the proposed solution is more fault-tolerant than the original use case, the following experiments are designed to investigate how the cluster responds during failures. The purpose of the first two experiments is to demonstrate the recovery speed of the application in response to anticipated failures, specifically those involving primary and replica pod failures. The final experiment aims to simulate a split-brain scenario by writing to the primary pod and reading from the replica during the primary failure.

6.2.1 Scenario 1: Primary instance failure

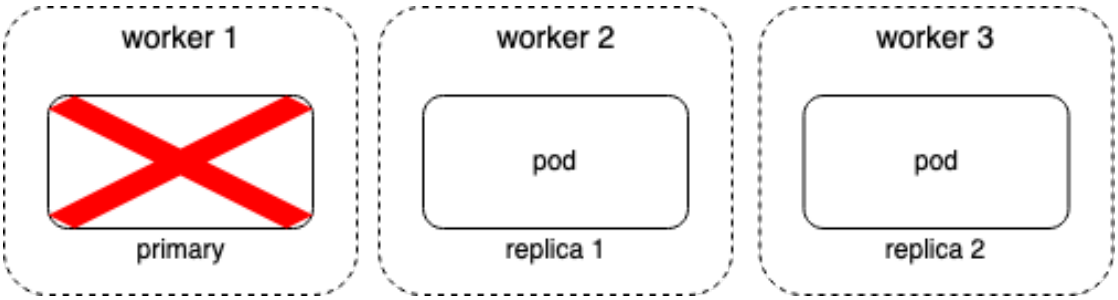

Figure 11. Experiment 1

The primary pod is deleted by the following command

```
kubectl delete pod -l role=primary --grace-period=1
```

Here, the grace period signifies that the kubelet will wait 1 second before forcibly terminating the pod, which simulates a scenario similar to an unexpected primary instance failure. In this case, the database operator is expected to perform automatic failover, as detailed in 4.2.1.

The experiment showed that it took about 133 seconds for the cluster to resume functionality. This represents a substantial enhancement in self-recovery from primary pod failure compared to the original use case. The script and results of this test are documented in Appendix C.

6.2.2 Scenario 2: Replica instance failure

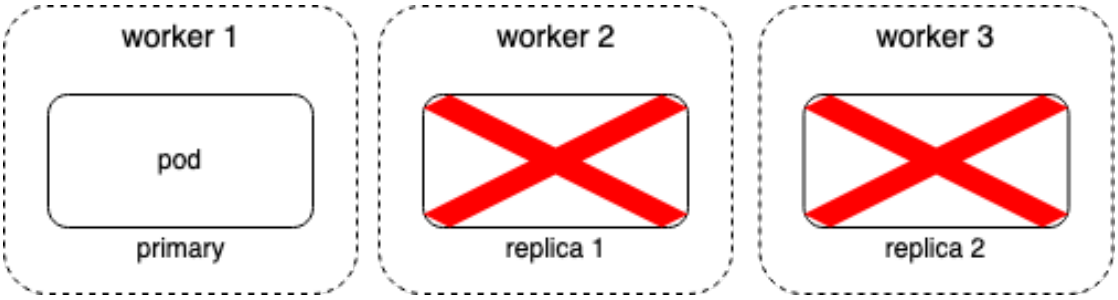

Figure 12. Experiment 2

The second experiment involves all the replicas that the script is connected to. The following command will delete all the replicas so that the replica endpoint couldn't function completely

.

```
kubectl delete pod -l role=replica --grace-period=1
```

The outcome shows that recovery from the failure took approximately 132 seconds. This represents a significant enhancement in self-recovery from replica pod failure compared to the original use case. The script and output for this test are documented in Appendix D.

6.2.3 Scenario 3: Reading from replica when removing primary

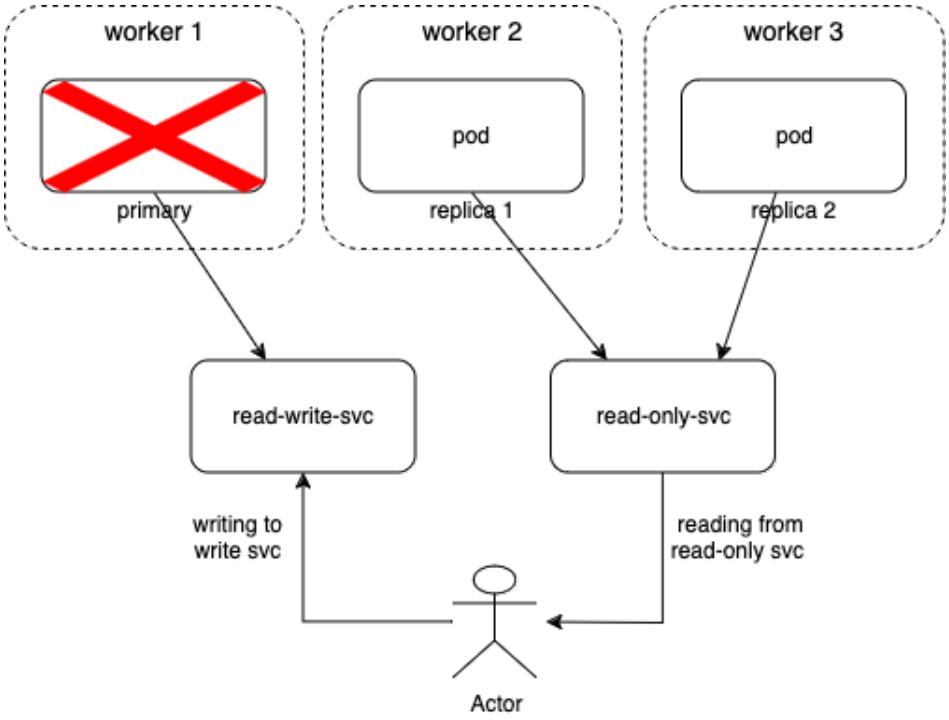

Figure 13. Experiment 3

The goal of this experiment is to simulate the split-brain issue, where the primary risk is that the client may receive incorrect data due to reading from a replica while the primary is experiencing an unexpected failure.

The testing procedure involves initially connecting to the primary (rw endpoint) of the cluster to write a random number. Subsequently, it connects to the replica (ro endpoint) to read the value. If the written and read values differ, the database has failed to maintain the cluster's state. Conversely, if the values match, it indicates that the database successfully blocked read and write operations during the primary's failure and replicated data to all standbys using synchronous replication.

Details of the script and output are provided in Appendix E. The test results show that recovery took about 135 seconds. This indicates that the application prevents any operations until the primary is restored, effectively avoiding split-brain failures.

6.2.4 Summary

These experiments demonstrate that the new setup significantly enhances recovery times, thereby reducing the need for on-call interventions. With optimal configuration, it is conceivable that the cluster could recover in under two minutes.

7 Conclusions

The primary objective of the thesis is to explore whether a Kubernetes-based architecture can enhance the recovery time compared to the existing VM-based architecture. Initially, it provides a detailed overview of Kubernetes and the Kubernetes operator. The thesis then proposes a new architecture based on Kubernetes and conducts experiments utilizing this architecture to evaluate recovery times following a range of unexpected failures.

The thesis identifies a problem in the current architecture, which requires manual intervention when switching to the primary to recover from a failure. Additionally, since the architecture is based on a 2-site setup, the split-brain failure mode becomes an issue.

Subsequently, the thesis explains how employing a Kubernetes operator resolves these issues. The application not only benefits from Kubernetes' self-healing capability, but also from the operator's inherent characteristic of avoiding split-brain failure. The primary/standby architecture ensures the cluster has only one primary at a time.

The narrative continues with experiments demonstrating how the cluster behaves during unexpected primary and replica failures. These experiments reveal that the cluster takes approximately 120 seconds to recover from such failures. The final experiment involves reproducing a split-brain situation. The result indicates that the cluster's service remains inaccessible until the primary node is restored and data is replicated to all standbys. This experiment is conducted by writing to the read-write endpoints while reading from the replicas to discern any differences between the writes and responses.

Further research should investigate methods to enhance the cluster's recovery speed to less than two minutes. According to the cloudnativePG documentation, reducing the switchoverDelay parameter could enable quicker recovery by minimizing the delay in the cluster failover process, thus not exceeding the permissible outage duration.

Additionally, the integration of Hardware Security Modules (HSM) into a Kubernetes cluster presents another challenge. It is crucial to explore how these security components can be effectively integrated into Kubernetes. This includes assessing their compatibility with database operators to ensure seamless functionality within the cluster.

References

[1] Longhorn documentation. https://longhorn.io/, 2024. Accessed: 15 April 2024.

[2] ADITYA, B., AND JUHANA, T. A high availability (ha) mariadb galera cluster across data center with optimized wrr scheduling algorithm of lvs-tun. In *2015 9th International Conference on Telecommunication Systems Services and Applications (TSSA)* (2015), IEEE, pp. 1–5.

[3] BAKHSHI, Z., AND RODRIGUEZ-NAVAS, G. *Lightweight Persistent Storage for Industrial Applications*. PhD thesis, Malardalen University (Sweden), 2023.

[4] BARTOLINI, G. Recommended architectures for postgresql in kubernetes. *Cloud Native Computing Foundation Blog* (September 2023).

[5] Cilium installation documentation. https://docs.cilium.io/en/stable/gettingstarted/k8s-install-default/. Accessed: 07 March 2024.

[6] Cloudnativepg automatic failover. https://cloudnative-pg.io/documentation/1.22/instance_manager/#failover. Accessed: 07 March 2024.

[7] Cloudnativepg replication documentation. https://cloudnative-pg.io/documentation/1.18/replication/. Accessed: 07 March 2024.

[8] DAME, M. *The Kubernetes Operator Framework Book: Overcome complex Kubernetes cluster management challenges with automation toolkits*. Packt Publishing Ltd, 2022.

[9] DENZLER, P., RAMSAUER, D., PREINDL, T., KASTNER, W., AND GSCHNITZER, A. Comparing different persistent storage approaches for containerized stateful applications. In *2022 IEEE 27th International Conference on Emerging Technologies and Factory Automation (ETFA)* (2022), IEEE, pp. 1–8.

[10] DOBIES, J., AND WOOD, J. *Kubernetes operators: Automating the container orchestration platform*. O'Reilly Media, 2020.

[11] DUAN, R., ZHANG, F., AND KHAN, S. U. A case study on five maturity levels of a kubernetes operator. In *2021 IEEE Cloud Summit (Cloud Summit)* (2021), IEEE, pp. 1–6.

[12] GALERA DOCUMENTATION. Quorum components. https://galeracluster.com/library/documentation/weighted-quorum.html. Accessed: 24 April 2024.

[13] Kubernetes native database. https://thenewstack.io/the-rise-of-the-kubernetes-native-database/. Accessed: 07 March 2024.

[14] Kubernetes documentation. https://kubernetes.io/. Accessed: 07 March 2024.

[15] NASSIM KEBBANI, P. T., AND MCKENDRICK, R. *The Kubernetes Bible*. Packt Publishing, 2022.

[16] RED HAT, INC. *Configuring OpenShift Data Foundation Disaster Recovery for OpenShift Workloads*. Red Hat, Inc., April 2023. Accessed: 2024-04-14.

[17] Kubernetes storage. https://shipit.dev/posts/kubernetes-overview-diagrams.html. Accessed: 07 March 2024.

[18] Kubernetes architecture. https://shipit.dev/posts/kubernetes-overview-diagrams.html. Accessed: 07 March 2024.

[19] STAPLETON, J. J. *Security without obscurity: A guide to confidentiality, authentication, and integrity*. CRC press, 2014.

[20] Kubernetes cluster installation. https://k8s.cs.ut.ee/lab3/#create-a-kubernetes-cluster. Accessed: 07 March 2024.

A Node preparation

```
# Delete snapd (resource utilization)
sudo su
snap list
snap remove lxd
snap remove lxd
snap remove core20
snap remove snapd
systemctl stop snapd
systemctl stop snapd.socket
systemctl disable snapd
systemctl mask snapd
apt purge snapd -y
apt-mark hold snapd
cat << EOF | sudo tee /etc/apt/preferences.d/nosnap.pref
Package: snapd
Pin: release a=*
Pin-Priority: -10
EOF
rm -rf /snap
rm -rf /var/snap
rm -rf /var/lib/snapd

# Disable swap
swapoff -a

# Install containerd
apt-get install ca-certificates curl gnupg
apt install -m 0755 -d /etc/apt/keyrings
curl -fsSL https://download.docker.com/linux/ubuntu/gpg \
| gpg --dearmor -o /etc/apt/keyrings/docker.gpg
chmod a+r /etc/apt/keyrings/docker.gpg
echo "deb [arch=$(dpkg --print-architecture) \
signed-by=/etc/apt/keyrings/docker.gpg] \
https://download.docker.com/linux/ubuntu \
$(. /etc/os-release && echo "$VERSION_CODENAME") stable" \
| sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
apt-get update
apt-get install containerd.io
systemctl start containerd
systemctl enable containerd

## Disable Ubuntu caching DNS resolver
systemctl disable systemd-resolved.service
systemctl stop systemd-resolved
rm -fv /etc/resolv.conf
cat > /etc/resolv.conf << EOF
nameserver 1.1.1.1
```

```
nameserver 8.8.8.8
EOF

## Change timezone to UTC
timedatectl set-timezone UTC

## Open firewall ports
ufw enable
ufw allow 4240/tcp # for CNI
ufw allow 6443/tcp # for control plane communications
ufw allow 8472/udp # for CNI
ufw allow 10250/tcp # for kubelet communication
ufw allow 30000-32767/tcp  # for publishing services

## Load kernel modules
cat << EOF | sudo tee /etc/modules-load.d/modules.conf
br_netfilter overlay
EOF
modprobe overlay
modprobe br_netfilter

## Enable sysctl settings
cat << EOF | sudo tee /etc/sysctl.d/99-sysctl.conf
net.bridge.bridge-nf-call-iptables = 1
net.bridge.bridge-nf-call-ip6tables = 1
net.ipv4.ip_forward = 1
EOF
sysctl -p

Install kubernetes tools (Kubeadm, kubelet and kubectl)

## Install Kubectl, kubelet, kubeadm
apt-get update
apt-get install -y apt-transport-https ca-certificates curl

curl -fsSL \
https://pkgs.k8s.io/core:/stable:/v1.29/deb/Release.key | \
sudo gpg --dearmor -o /etc/apt/keyrings/kubernetes-apt-keyring.gpg

echo 'deb [signed-by=/etc/apt/keyrings/kubernetes-apt-keyring.gpg] \
https://pkgs.k8s.io/core:/stable:/v1.29/deb/ /' \
| sudo tee /etc/apt/sources.list.d/kubernetes.list

apt-get install -y kubelet kubeadm kubectl
apt-mark hold kubelet kubeadm kubectl

---
# Cluster initialization with kubeadm [20]
cat << EOF | sudo tee cluster-init.yaml
```

```
apiVersion: kubeadm.k8s.io/v1beta3
kind: ClusterConfiguration
controlPlaneEndpoint: "193.40.103.161:6443"
networking:
  dnsDomain: cluster.local
  podSubnet: "10.217.0.0/16"
  serviceSubnet: "10.96.0.0/12"
apiServer:
  extraArgs:
    advertise-address: "193.40.103.161"
---
apiVersion: kubeadm.k8s.io/v1beta3
kind: InitConfiguration
nodeRegistration:
  name: "master1"
  criSocket: "unix:///run/containerd/containerd.sock"
---
kind: KubeletConfiguration
apiVersion: kubelet.config.k8s.io/v1beta1
cgroupDriver: "systemd"
shutdownGracePeriod: 5m0s
shutdownGracePeriodCriticalPods: 5m0s
EOF

kubeadm init --config=cluster-init.yaml

# Install cilium [5]
CILIUM_CLI_VERSION=$(curl -s
↪  https://raw.githubusercontent.com/cilium/cilium-cli/main/stable.txt)
CLI_ARCH=amd64
if [ "$(uname -m)" = "aarch64" ]; then CLI_ARCH=arm64; fi
curl -L --fail --remote-name-all
↪  https://github.com/cilium/cilium-cli/releases/download/${CILIUM_CLI_VERSION}/cilium-linux-${CLI_A
sha256sum --check cilium-linux-${CLI_ARCH}.tar.gz.sha256sum
sudo tar xzvfC cilium-linux-${CLI_ARCH}.tar.gz /usr/local/bin
rm cilium-linux-${CLI_ARCH}.tar.gz{,.sha256sum}

# Install Longhorn
apt-get install open-iscsi
modprobe open-iscsi

# Check NFSv4.1 support is enabled in kernel
cat /boot/config-`uname -r`| grep CONFIG_NFS_V4_1

# Check NFSv4.2 support is enabled in kernel
cat /boot/config-`uname -r`| grep CONFIG_NFS_V4_2

apt-get install nfs-common
```

B Kubernetes Manifests

```yaml
# cnpg-cluster.yaml

apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cnpg-cluster
spec:
  maxSyncReplicas: 2
  minSyncReplicas: 2
  enableSuperuserAccess: true
  superuserSecret:
    name: superuser-secret
  monitoring:
    enablePodMonitor: true
  instances: 3
  bootstrap:
    initdb:
      database: app
      owner: app
      secret:
        name: app-secret
  primaryUpdateStrategy: unsupervised
  storage:
    storageClass: longhorn
    size: 1Gi
  walStorage:
    size: 512Mi
  # replicationSlots:
  #   highAvailability:
  #     enabled: true
  # postgresql:
  #   parameters:
  #     max_slot_wal_keep_size: '500MB'
  #     hot_standby_feedback: 'on'
---
apiVersion: v1
data:
  username: cG9zdGdyZXM=
  password: cGFzc3dvcmQ=
kind: Secret
metadata:
  name: superuser-secret
type: kubernetes.io/basic-auth
---
apiVersion: v1
data:
  username: YXBw
```

```
  password: cGFzc3dvcmQ=
kind: Secret
metadata:
  name: app-secret
type: kubernetes.io/basic-auth


---
apiVersion: v1
data:
  username: YXBw
  password: cGFzc3dvcmQ=
kind: Secret
metadata:
  name: app-secret
type: kubernetes.io/basic-auth

# Prometheus manifest
---
apiVersion: v1
kind: ServiceAccount #
metadata:
  name: prometheus
  namespace: monitoring
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole #
metadata:
  name: prometheus
rules:
- apiGroups: [""]
  resources:
  - nodes
  - services
  - endpoints
  - pods
  - nodes/metrics
  - nodes/proxy
  - nodes/stats
  verbs: ["get", "list", "watch"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding #
metadata:
  name: prometheus
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: prometheus
```

```
subjects:
- kind: ServiceAccount
  name: prometheus
  namespace: monitoring
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: prometheus-config
data: #
  prometheus.yml: |
    global:
      scrape_interval: 15s
      evaluation_interval: 15s
    scrape_configs:
      - job_name: 'postgres-cluster'
        kubernetes_sd_configs:
          - role: pod
        relabel_configs:
          - source_labels: [__meta_kubernetes_namespace]
            action: keep
            regex: default
          - source_labels: [__meta_kubernetes_pod_label_cnpg_io_instanceRole]
            action: replace
            target_label: instance_role
          - source_labels: [__meta_kubernetes_pod_container_port_number]
            action: keep
            regex: 9187
      - job_name: 'kubernetes-node-exporter'
        kubernetes_sd_configs:
          - role: pod
        relabel_configs:
          - source_labels: [__meta_kubernetes_pod_label_app]
            action: keep
            regex: node-exporter
      - job_name: 'kubernetes-kube-state-metrics'
        scrape_interval: 1s
        kubernetes_sd_configs:
          - role: pod
        relabel_configs:
          - source_labels: [__meta_kubernetes_pod_label_app_kubernetes_io_name]
            action: keep
            regex: kube-state-metrics
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: prometheus
spec:
```

```
  serviceName: prometheus
  replicas: 1
  selector:
    matchLabels:
      app: prometheus
  template:
    metadata:
      labels:
        app: prometheus
    spec:
      serviceAccountName: prometheus #
      initContainers:
      - name: prometheus-data-permission-setup
        image: busybox
        command: ["/bin/chown","-R","65534:65534","/data"]
        volumeMounts:
        - name: data
          mountPath: /data
      containers:
      - name: prometheus
        image: prom/prometheus:latest
        args:
        - "--config.file=/etc/prometheus/prometheus.yml"
        - "--storage.tsdb.path=/data"
        - "--web.enable-lifecycle"
        ports:
        - containerPort: 9090
          name: http-metrics
        volumeMounts:
        - name: config
          mountPath: /etc/prometheus
        - name: data
          mountPath: /data
      volumes:
      - name: config
        configMap:
          name: prometheus-config
  volumeClaimTemplates:
  - metadata:
      name: data
    spec:
      accessModes:
      - ReadWriteOnce
      resources:
        requests:
          storage: 2Gi
---
```

C Primary failure testing script

C.1 Script

```python
import psycopg2
from psycopg2 import OperationalError
import time
import random

postgres_rw, postgres_ro = "10.96.188.11", "10.101.123.167"
dbname, username, password = "app", "app", "password"

def connect_to_database():
    # Modify these parameters with your PostgreSQL database information
    try:
        conn = psycopg2.connect(
            dbname=dbname,
            user=username,
            password=password,
            host=postgres_rw,
            port="5432"
        )
        return conn
    except OperationalError as e:
        return None

def execute_write_read_query(conn):
    try:
        cursor = conn.cursor()

        try:
            # Create a table if not exists
            create_table_query = """
            CREATE TABLE IF NOT EXISTS test_table (
                id SERIAL PRIMARY KEY,
                column_name INTEGER
            )
            """
            cursor = conn.cursor()
            cursor.execute(create_table_query)
            conn.commit()

            # # Generate a random value
            random_value = random.randint(0, 1000000)

            # # Execute write query
            start_time = time.monotonic()
            cursor.execute("INSERT INTO test_table (column_name) VALUES (%s)",
            ↪  (random_value,))
```

```
            conn.commit()
            end_time = time.monotonic()
            write_latency = end_time - start_time

            # Execute read query
            start_time = time.monotonic()
            cursor.execute("SELECT column_name FROM test_table ORDER BY id DESC
            ↪  LIMIT 1")
            result = cursor.fetchone()
            end_time = time.monotonic()
            read_latency = end_time - start_time

            # Check if read is successful
            if result is not None:
                read_success = True
            else:
                read_success = False

            # Check if write is successful (compare the written value with the read
            ↪  value)
            if result is not None and result[0] == random_value:
                write_success = True
            else:
                write_success = False

            return write_success, write_latency, read_success, read_latency

        except psycopg2.Error as e:
            conn.rollback()
            print(f"{e}")
            return False, None, False, None
        finally:
            cursor.close()
    except:
        print("no connection")
        return False, None, False, None

def main():
    while True:
        conn = connect_to_database()

        if conn is None:
            print("Lost connection to the database")

            # Wait until the connection is re-established
            while True:
                print("Attempting to reconnect...")
                conn = connect_to_database()
                if conn is not None:
```

```
                    break
            write_success, write_latency, read_success, read_latency =
            ↪  execute_write_read_query(conn)
            print(f"{time.monotonic()}; Write Success: {write_success}, Latency:
            ↪  {write_latency}; Read Success: {read_success}, Latency:
            ↪  {read_latency}")

if __name__ == "__main__":
    main()
```

C.2 Output

```
Current time: 4363537.877782201; Write Success: True, Latency:
↪  0.009144876152276993; Read Success: True, Latency: 0.0014316439628601074
Current time: 4363537.910006191; Write Success: True, Latency:
↪  0.010425306856632233; Read Success: True, Latency: 0.003867696039378643
Current time: 4363537.950310707; Write Success: True, Latency:
↪  0.011125799268484116; Read Success: True, Latency: 0.0015683574602007866
Lost connection to the database
Attempting to reconnect...
Attempting to reconnect...
Attempting to reconnect...
cluster takes 132.62531081866473 to recover
Current time: 4363670.610451287; Write Success: True, Latency:
↪  0.018396921455860138; Read Success: True, Latency: 0.002729536034166813
Current time: 4363670.639573601; Write Success: True, Latency: 0.00948651134967804;
↪  Read Success: True, Latency: 0.002178587019443512
Current time: 4363670.668656484; Write Success: True, Latency:
↪  0.009290928021073341; Read Success: True, Latency: 0.0015430273488163948
```

D Replica failure testing script

D.1 Script

```python
import psycopg2
from psycopg2 import OperationalError
import time

REPLICA_IP, dbname, username, password = "10.101.118.101", "app", "app", "password"

def connect_to_database():
    # Modify these parameters with your PostgreSQL database information
    try:
        conn = psycopg2.connect(
            dbname=dbname,
            user=username,
            password=password,
            host=REPLICA_IP,
            port="5432"
        )
        return conn
    except OperationalError as e:
        return None

def execute_read_query(conn):
    try:
        cursor = conn.cursor()

        try:
            # Execute read query
            start_time = time.monotonic()
            cursor.execute("SELECT column_name FROM test_table ORDER BY id DESC
            ↪  LIMIT 1")
            result = cursor.fetchone()
            end_time = time.monotonic()
            read_latency = end_time - start_time

            # Check if read is successful
            if result is not None:
                read_success = True
            else:
                read_success = False

            return read_success, read_latency

        except psycopg2.Error as e:
            conn.rollback()
            print(f"{e}")
            return False, None
```

```
        finally:
            cursor.close()
    except:
        print("no connection")
        return False, None

def main():
    while True:
        conn = connect_to_database()

        if conn is None:
            print("Lost connection to the database")
            failure_start_time = time.monotonic()
            # Wait until the connection is re-established
            while True:
                print("Attempting to reconnect...")
                conn = connect_to_database()
                if conn is not None:
                    print(f"cluster takes {time.monotonic() - failure_start_time} to
                    ↪ recover")
                    break
        read_success, read_latency = execute_read_query(conn)
        print(f"{time.monotonic()}; Read Success: {read_success}, Latency:
        ↪ {read_latency}")

if __name__ == "__main__":
    main()
```

D.2 Output

```
4360728.066286806; Read Success: True, Latency: 0.002768927253782749
4360728.089979115; Read Success: True, Latency: 0.005216777324676514
Lost connection to the database
Attempting to reconnect...
Attempting to reconnect...
cluster takes 132.5847332375124 to recover
4360860.731599201; Read Success: True, Latency: 0.011759408749639988
4360860.745601946; Read Success: True, Latency: 0.002377951517701149
```

E Split-brain testing script

E.1 Script

```python
import psycopg2
from psycopg2 import OperationalError
import time
import random

postgres_rw, postgres_ro = "10.0.3.138", "10.0.1.104"
dbname, username, password = "app", "app", "password"

def connect_to_database(host):
    # Modify these parameters with your PostgreSQL database information
    try:
        conn = psycopg2.connect(
            dbname=dbname,
            user=username,
            password=password,
            host=host,
            port="5432"
        )
        return conn
    except OperationalError as e:
        return None

def execute_write_read_query(primary_conn, replica_conn):
    primary_cursor, replica_cursor = primary_conn.cursor(), replica_conn.cursor()

    try:
        # Create a table if not exists
        create_table_query = """
        CREATE TABLE IF NOT EXISTS test_table (
            id SERIAL PRIMARY KEY,
            column_name INTEGER
        )
        """
        primary_cursor.execute(create_table_query)
        primary_conn.commit()

        # # Generate a random value
        random_value = random.randint(0, 1000000)

        # # Execute write query
        start_time = time.monotonic()
        primary_cursor.execute("INSERT INTO test_table (column_name) VALUES (%s)",
        ↪  (random_value,))
        primary_conn.commit()
        end_time = time.monotonic()
```

```
        write_latency = end_time - start_time

        # Execute read query
        start_time = time.monotonic()
        replica_cursor.execute("SELECT column_name FROM test_table ORDER BY id DESC
        ↪  LIMIT 1")
        result = replica_cursor.fetchone()
        end_time = time.monotonic()
        read_latency = end_time - start_time

        # Check if read is successful
        if result is not None:
            read_success = True
        else:
            read_success = False

        # Check if write is successful (compare the written value with the read
        ↪  value)
        if result is not None and result[0] == random_value:
            write_success = True
        else:
            write_success = False
            print(f"random_value is {random_value}, result[0] is {result[0]}")

        return write_success, write_latency, read_success, read_latency
    except psycopg2.Error as e:
        print(f"{e}")
        return False, None, False, None
    finally:
        primary_cursor.close()
        replica_cursor.close()

def main():
    while True:
        primary_conn, replica_conn = connect_to_database(postgres_rw),
        ↪  connect_to_database(postgres_ro)
        if primary_conn is None or replica_conn is None:
            print("Lost connection to the database")

            # Wait until the connection is re-established
            while True:
                print("Attempting to reconnect...")
                primary_conn, replica_conn = connect_to_database(postgres_rw),
                ↪  connect_to_database(postgres_ro)
                if primary_conn is not None and (replica_conn is not None):
                    break

        else:
```

```
            write_success, write_latency, read_success, read_latency =
            ↪  execute_write_read_query(primary_conn, replica_conn)
            print(f"{time.monotonic()}; Write Success: {write_success}, Latency:
            ↪  {write_latency}; Read Success: {read_success}, Latency:
            ↪  {read_latency}")

if __name__ == "__main__":
    main()
```

E.2 Output

```
4362370.219541379; Write Success: True, Latency: 0.01057056151330471; Read Success:
↪  True, Latency: 0.0023303302004933357
4362370.273977052; Write Success: True, Latency: 0.010082093998789787; Read
↪  Success: True, Latency: 0.0029499325901269913
4362370.333786897; Write Success: True, Latency: 0.01132146641612053; Read Success:
↪  True, Latency: 0.0023874007165431976
Lost connection to the database
Attempting to reconnect...
Attempting to reconnect...
cluster takes 132.8893731413409 to recover
4362503.311456433; Write Success: True, Latency: 0.019097312353551388; Read
↪  Success: True, Latency: 0.0028968779370188713
4362503.36329722; Write Success: True, Latency: 0.010089441202580929; Read Success:
↪  True, Latency: 0.003241436555981636
4362503.416680352; Write Success: True, Latency: 0.012647973373532295; Read
↪  Success: True, Latency: 0.005675826221704483
4362503.464382069; Write Success: True, Latency: 0.009269298054277897; Read
↪  Success: True, Latency: 0.0037214430049061775
4362503.508715628; Write Success: True, Latency: 0.011828869581222534; Read
↪  Success: True, Latency: 0.0033265361562371254
```

F Useful commands

```
# List all pods sort by node name
kubectl get pods -A -o wide --sort-by=.spec.nodeName

# How to join second node as control plane node
## In the first master node run:
kubeadm init phase upload-certs --upload-certs

## Take note of the certificate key
## Add the certificate key to parameter --certificate-key
## to the end of the output of
kubeadm token create --print-join-command

## Example of a complete command:
kubeadm join 193.40.103.161:6443 \
--token puoa01.a5oxbafba7k2pyns \
--discovery-token-ca-cert-hash \
sha256:5c185a27adfba2940f50d247c66de45b9e5bd35ed00d709c59ec40a5591b5bf0 \
--control-plane --certificate-key \
8d3c989c8fd311a1ed7640992648bbec729b267103b4a62cc6cb33fa85b79035

# Remove taint on master nodes so that they could act as worker
kubectl taint nodes master1 node-role.kubernetes.io/control-plane:NoSchedule-
kubectl taint nodes master2 node-role.kubernetes.io/control-plane:NoSchedule-
kubectl taint nodes master3 node-role.kubernetes.io/control-plane:NoSchedule-

# Use Helm to generate Kubernetes resource files
helm template eppki traefik/traefik \
--create-namespace --include-crds -n traefik > traefik.yaml

# forward Longhorn service to access the UI
kubectl port-forward service/longhorn-frontend -n longhorn-system 2222:80
```

G Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, **Thi Song Huong Pham**,

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to

   reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

   Fault-tolerant Distributed Database for Public Key Infrastructure,

   supervised by Pelle Jakovits and Matti Siekkinen.

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Thi Song Huong Pham
07/03/2024