Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Master Thesis

Orchestrating data governance workloads as stateful services in cloud environments using Kubernetes Operator Framework

Xiaomin Wang

Course of Study: Information Technology

Examiner: Prof. Dr.-Ing. Bernhard Mitschang

Supervisor: Dipl.-Phys. Cataldo Mega

Commenced: June 9, 2022

Completed: December 9, 2022

Abstract

Data is becoming the core corporate asset that will determine the business's success. As a result, it is critical for governing enterprise data. Previously, the Enterprise Content Management (ECM) system was employed by many companies to manage and process their enterprise data, which is a monolithic data governance application. As the ECM system is typically deployed on bare metal or at most in a virtualized IT infrastructure, it lacks the flexibility to support Continuous Integration (CI) and Continuous Delivery (CD) cost-effectively. Cloud computing has gained popularity as a powerful platform for application deployment, owing to perceived benefits such as elasticity to fluctuating load and reduced operational costs as compared to running in traditional data centers. Therefore, it is promising to migrate the legacy ECM system into the cloud. The goal of this thesis is to orchestrate stateful database workloads in *Kubernetes* that are typical for *ECM* systems. For our concept verification, we included a comparison and analysis between traditional and comparable cloud native Relational Database Management System (RDBMS) using IBM DB2, PostgreSQL, CockRoachDB and Google Spanner. We proposed an implementation of the Monitor-Analyze-Plan-Execute (MAPE) concept using the *Kubernetes Operator* framework. With our prototype implementation, we proved that the *Kubernetes* operator is able to deploy a cluster for DB2 consisting of a read/write primary and up to three read-only members. Various experiments carried out on the prototype have evidenced its High Availability (HA), Disaster Recovery (DR) features as well as read scalability.

Contents

1 Introduction 17
1.1 Motivation . . . 17
1.2 Organization . . . 18

2 Background 19
2.1 Traditional RDBMS Characteristics . . . 19
2.2 Cloud Native RDBMS Characteristics . . . 22
2.3 Comparison Between Traditional RDBMS and Cloud Native RDBMS . . . 24

3 Foundations 27
3.1 Kubernetes . . . 27
3.2 Kubernetes Stateful Architecture Resources . . . 29
3.3 Operator . . . 31

4 Related Work 35
4.1 Design Changes for Decomposing Monolithic ECM Systems . . . 35
4.2 Deploying ECM Workloads in Cloud Environments Based on Kubernetes and Docker . . . 36

5 The MAPE Workload Management Concept 37
5.1 MAPE Concept . . . 37
5.2 MAPE Implemented by Kubernetes Control Loop . . . 38

6 Prototype 41
6.1 Design Approach . . . 41
6.2 Implementation . . . 46
6.3 Test and Evaluation . . . 67

7 Conclusion and Outlook 81

Bibliography 83

List of Figures

2.1 High Availability Disaster Recovery (HADR) Synchronization Mode [DB2hadr] . 20
2.2 PostgreSQL File-based Log Shipping with Streaming Replication [posRep18] . . 21
2.3 CockRoachDB HA and DR Features [RM17] 22
2.4 New Nodes Connection/Communication via Gossip [RM17] 23
2.5 Comparison Between Traditional and Cloud Native Database About HA and DR . 25
2.6 Static Scaling and Elastic Scaling for Unpredictable Workload Changes [FLR+14] 26

3.1 Kubernetes Architecture [MSK19] 27
3.2 General Structure of an Operator [OTH+21] 32
3.3 Kubernetes Operator Architecture and Mechanism 33

4.1 Topology of ECM System Developed by Shao [Sha20] 35
4.2 Topology of ECM System Inside a Kubernetes Cluster Developed by Trybek [Try21] 36

5.1 System Topology Applying the MAPE Loop Concept [RMM12] 37
5.2 ECM System Topology Refactored by Shao [Sha20] and Trybek [Try21] Applying a Cluster Manager for DB2 Based on the MAPE Loop Concept in *Kubernetes* . . 38

6.1 Topology of Stateful DB2 Database Service Prototype Inside a Kubernetes Cluster 41
6.2 Component Diagram of a Cluster for DB2 42
6.3 State Diagram of *Pods* Related to Failover 43
6.4 Structure of a *Kubernetes* Operator for DB2 45
6.5 Reconciling Logic of the *Kubernetes* Operator for DB2 64
6.6 Example of a Client Accessing the Primary DB2 Database Service 70
6.7 Example of a Client Accessing the Read-only DB2 Database Service 71
6.8 HADR Details of Each *Pod* Before Failover 72
6.9 Process of Label Changes for *Pods* in Failover 73
6.10 HADR Details of Each *Pod* After Failover 73
6.11 Availability Metrics 74
6.12 Definition and Differences Between Recovery Point Objective (RPO) and Recovery Time Objective (RTO) 75
6.13 HADR Details of Each *Pod* Before a Disaster 76
6.14 HADR Details of Each *Pod* After a Disaster 77
6.15 Average Response Time for 1000 Read Requests Per Client in Different Clusters for DB2 78
6.16 Transaction Distribution in Different Clusters for DB2 with Different Numbers of Clients 79

List of Tables

6.1 Implications of HADR Configuration Parameters 49
6.2 HADR Configuration Parameters for the Primary 50
6.3 HADR Configuration Parameters for the Principal Standby 50
6.4 HADR Configuration Parameters for the Auxiliary Standby1 51
6.5 HADR Configuration Parameters for the Auxiliary Standby2 51
6.6 Environment Variables of the Container Launched in *StatefulSet's Pods* 63
6.7 HADR State and HADR Connect Status of DB2 Databases 65
6.8 Results of 10 Experiments Measuring Availability 75

List of Algorithms

6.1 Setting up HADR on DB2 Instances with Different HADR Roles 48
6.2 Failover Algorithm of Governor 54
6.3 Failover Management of Reconciler in the *Kubernetes* Operator for DB2 65

List of Listings

6.1 Example of */hadr/hadr.cfg* . . . 47
6.2 HADR Service Ports in */etc/services* . . . 49
6.3 Example of *db2.yml* . . . 52
6.4 Dockerfile for Building a Custom DB2 Image . . . 55
6.5 Defining DB2Cluster Resource API in *api/v1/db2cluster_types.go* . . . 57
6.6 CR Configuration File Format . . . 57
6.7 Specifying Resources Watched by the Controller in *controllers/db2cluster_controller.go* 58
6.8 Primary Service YAML File . . . 59
6.9 Read-only Service YAML File . . . 59
6.10 YAML File of PVC for Generating PV to Mount HADR Configuration File . . . 60
6.11 YAML File of StatefulSet Deployed in the Cluster for DB2 . . . 61
6.12 HAProxy Configuration File . . . 68
6.13 CR Configuration File . . . 69
6.14 Commands for Setting up HADR on *db2-2* as Primary . . . 77
6.15 Commands for Setting up HADR on *db2-3* as Principal Standby . . . 77

Acronyms

ACID Atomicity, Consistency, Isolation, Durability. 19

ACK ACKnowledgement. 20

ASYNC Asynchronous. 20

CD Continuous Delivery. 3

CI Continuous Integration. 3

CLI Command Line Interface. 56

CR Custom Resource. 32

CRD Custom Resource Definition. 32

CRUD Create, Retrieve, Update, Delete. 24

DNS Domain Name System. 29

DR Disaster Recovery. 3

ECM Enterprise Content Management. 3

ER Model Entity–Relationship Model. 19

FQDN Fully Qualified Domain Name. 29

HA High Availability. 3

HADR High Availability Disaster Recovery. 7

MAPE Monitor-Analyze-Plan-Execute. 3

NEARSYNC Near Synchronous. 20

NFS Network File System. 59

OLTP Online Transaction Processing. 19

PV Persistent Volume. 30

PVC Persistent Volume Claim. 30

QA Quality Assurance. 29

RBAC Role-based access control. 46

RDBMS Relational Database Management System. 3

RPM Resource Provisioning Manager. 38

RPO Recovery Point Objective. 7

RTO Recovery Time Objective. 7

SQL Structured Query Language. 19

SUPERASYNC Super Asynchronous. 20

SYNC Synchronous. 20

TCP Transmission Control Protocol. 20

TTL Time to live. 52

VM Virtual Machine. 32

WAL Write-Ahead Log. 21

1 Introduction

1.1 Motivation

Data is subject to data governance as it is a strategic asset for any organization. The goal of data governance is to manage business-relevant data strategically throughout its full lifecycle, from generation to destruction. In addition, data governance defines who can take what action, upon what data, in what situations and using what methods, and includes business processes, corporate policies and regulatory compliance for ensuring effective data management. In the past, companies stored unstructured data using homegrown monolithic data governance applications called ECM systems on top of distributed content repositories. ECM systems are typically deployed on on-premise baremetal-infrastructure which requires a professional team or technicians to manage and maintain. Although the existing ECM systems are able to provide reliable and performant content services, they lack the potential of CI as well as CD in a cost-effective way. Cloud computing technology has become popular of late due to its economic and technical advantages, such as pay-per-use pricing models, scalability, flexibility as well as elasticity. Migrating the legacy ECM system to the cloud allows it to better support automated CI/CD by taking advantage of inherited built-in features of the cloud.
In previous master theses, Shao [Sha20] split monolithic ECM applications into smaller self-contained components, and Trybek [Try21] developed a cloud deployment model for the ECM system but only the set of stateless services. Thereby, we aim to design and implement a deployment model including stateful services, i.e. stateful database services. For our prototype, we looked at the traditional database IBM DB2 as it is utilized to offer database services in legacy ECM systems. Traditional database services are challenging to move to the cloud given that their deployment architecture is not natively suitable for the cloud environment. Furthermore, considering *Kubernetes* was originally designed to handle stateless workloads autonomously, deploying a stateful database application within a *Kubernetes* cluster is a significant challenge that requires a more complex database specific deployment design. We investigated the difference between traditional RDBMS and cloud native RDBMS in terms of architecture, HA, DR features as well as horizontal scalability. Based on our analysis, we designed a HADR cluster for DB2 to provide high available stateful database services in *Kubernetes*. We constructed and implemented a *Kubernetes* custom operator that can automatically deploy a cluster for DB2 employing the MAPE lifecycle as proposed by Ritter et al. [RMM12] utilizing the *Kubernetes Operator* framework. In addition, in order to verify and evaluate our approach, we developed a set of function verification tests for HA, DR features and read scalability of the cluster for DB2 and performed various tests on it which we show later in this thesis.

1.2 Organization

The remainder of this thesis is organized as follows. In Chapter 2, we compare traditional RDBMS with cloud native RDBMS and analyze their differences regarding architecture, HA and DR features as well as horizontal scalability. In Chapter 3, we discuss the necessary terms and technologies utilized in this work. We introduce the previous work related to migrating the ECM system into the cloud environment in Chapter 4. Chapter 5 presents the high-level concept of this work, including what the MAPE concept is and how it relates to the implementation of our *Kubernetes* operator for IBM DB2. Chapter 6 describes the design, implementation, and function verification tests of our stateful database service prototype. Finally, we conclude this thesis and discuss future directions in Chapter 7.

2 Background

For the past 30 to 40 years, businesses and organizations have primarily stored and analyzed their data using traditional database systems, such as RDBMS. They are based on Entity–Relationship Model (ER Model), which means data is organized in structured forms using tables and relations. Structured Query Language (SQL) is the standard language used to manage and access data in tables. They have strong Atomicity, Consistency, Isolation, Durability (ACID) capabilities in support of Online Transaction Processing (OLTP) workloads in secure environments.
The wide adoption of clouds has motivated a new RDBMS architecture design and technology. In the context of Cloud, RDBMS should meet the following requirements:

- In terms of the cloud characteristic massive multi-tenancy, a large number of databases are assigned to a large number of customers.
- There is no limit to horizontal scaling as such a system can be made out of inexpensive legacy hardware.
- Since HA and DR in cloud environments mean high complexity, managing a very large and complex RDBMS system must be done in an automated fashion. Manual administration does not work any longer.
- Geographically distributed systems require even more complex logic and monitoring.
- On the data side, big data needs scalable database systems to produce valuable results quickly at an acceptable cost.

In order to achieve these requirements, a new kind of database system called *Cloud Native Databases* was introduced. These systems have built-in distributed system logic and run natively on cloud platforms. They also adopt transaction-oriented SQL to manage and access data, which supports ACID logic.
In this chapter, we first introduce the characteristics of traditional RDBMS and cloud native RDBMS separately that relate to this work. Then we compare these two kinds of RDBMS in terms of these key characteristics, namely HA, DR and horizontal scalability.

2.1 Traditional RDBMS Characteristics

In this section, we take IBM BD2 [1] and PostgreSQL [2] as examples of traditional RDBMS to study HA, DR and horizontal scalability.

[1] IBM BD2: https://www.ibm.com/db2
[2] PostgreSQL: https://www.postgresql.org/

2.1.1 IBM DB2

- **High Availability and Disaster Recovery Design and Implementation**
The IBM DB2 Server, the HADR framework provides a high availability solution for both partial and complete production site failures. In a HADR environment, log data is shipped continuously from a primary database to one or up to three standby databases and reapplied to the standby databases. When the primary database fails, applications are redirected to a standby database that automatically takes over the role of the primary database [BDK+12]. There are four synchronization modes in the HADR framework: Synchronous (SYNC) mode, Near Synchronous (NEARSYNC) mode, Asynchronous (ASYNC) mode and Super Asynchronous (SUPERASYNC) mode, which are presented in Figure 2.1. For SYNC and NEARSYNC modes, the primary will wait for an ACKnowledgement (ACK) message from the standby to confirm that the logs have been received and written to disk on the standby (SYNC mode) or have been received on the standby (NEARSYNC mode). In terms of ASYNC mode, replication is considered done as soon as the logs are delivered to the Transmission Control Protocol (TCP) layer of the primary host machine. Transactions on the primary do not wait for replication of logs to the standby in SUPERASYNC mode [DB2hadr].

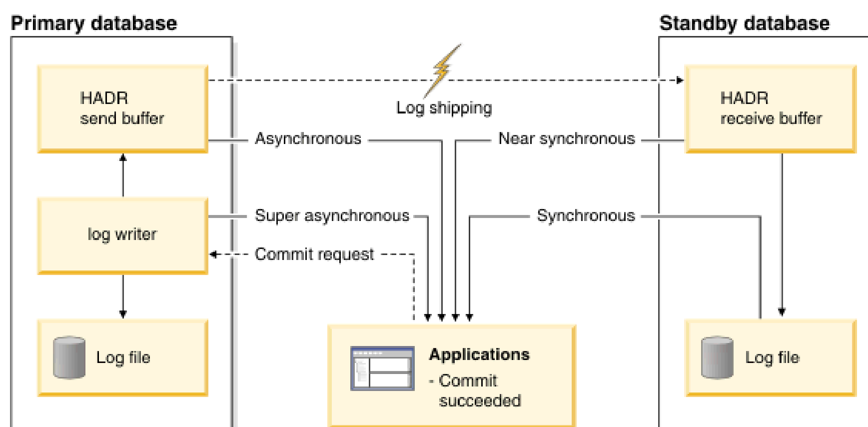


Figure 2.1: HADR Synchronization Mode [DB2hadr]

- **IBM DB2 Enterprise Server Horizontal Read Scalability**
IBM DB2 Enterprise Servers realize horizontal scaling by adding more standbys as they employ a shared-nothing architecture. It utilizes the HADR framework to add up to three standby DB2 servers in the cluster achieving such scalability. Each standby server is set up as a replication of the primary server and serves read-only requests. In the case of heavy read requests from clients, additional standby servers will share some traffic with the primary server offering a read performance benefit.

- **IBM DB2 pureScale shared-everything Database Horizontal Scalability Feature**
The IBM DB2 pureScale Server environment is intended for horizontal scaling solutions that are built on a shared-everything architecture. That means multiple database instances vest the same role as read-write members in a database cluster system sharing the same storage disk but owning different database partitions. Incoming database requests from DB2 clients are automatically directed to the member with the lowest workload to rebalance

the workload among the members. With the DB2 pureScale feature, we can transparently add more members to scale out and cope with OLTP workload growth without application changes and data redistribution.

2.1.2 PostgreSQL

- **High Availability and Disaster Recovery Design and Implementation**
  PostgreSQL maintains high availability by ensuring that a standby server will take over if the primary server crashes. As shown in Figure 2.2, there are two methods to ensure database synchronization: file-based log shipping and streaming replication. PostgreSQL implements file-based log shipping by transferring Write-Ahead Log (WAL) records one file at a time, which contains all changes made in the database. Compared with file-based log shipping, streaming replication allows more up-to-date data because it transfers WAL records between the primary server and the standby server without waiting for the WAL file to be filled. In the primary server, a process called *WAL Sender* is responsible for sending WAL records to the standby server via TCP connection. A process named *WAL Receiver* running on the standby server is used to receive WAL records. If the connection between the *WAL Sender* and the *WAL Receiver* is broken, the standby server will first restore all the WAL available in the archive directory via file-based shipping.

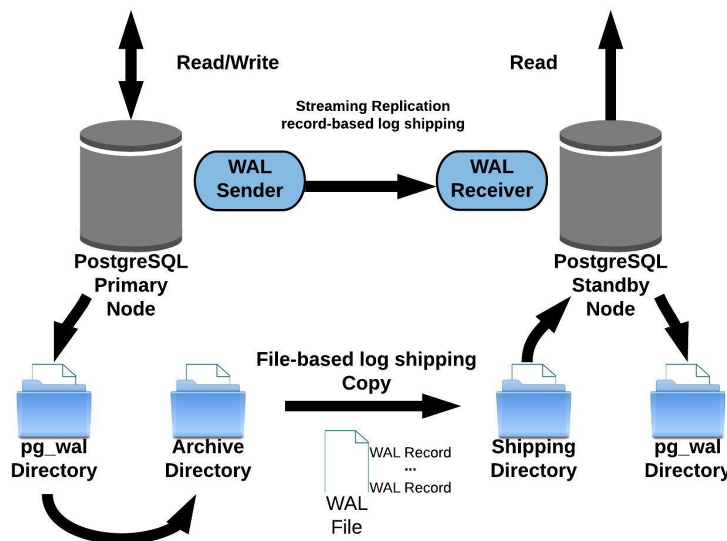


Figure 2.2: PostgreSQL File-based Log Shipping with Streaming Replication [posRep18]

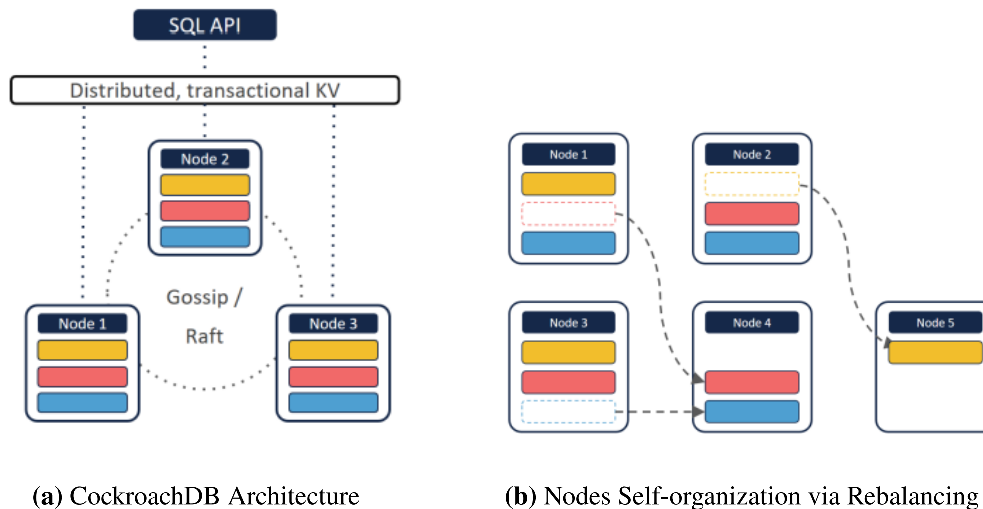
- **PostegreSQL Horizontal Read Scalability**
  PostgreSQL implements horizontal scaling in shared-nothing architecture and one primary multi-replicas system by adding more replica database servers. After adding more replicas to the cluster system, the PostgreSQL replication mechanism will synchronize data between the primary and replicas. Read scalability is achieved as well since replicas can only serve read requests.

2.2 Cloud Native RDBMS Characteristics

In this section, CockroachDB [3] and Google Spanner [4] are considered as examples of cloud native RDBMS to study HA, DR and horizontal scalability.

2.2.1 CockroachDB

- **High Availability and Disaster Recovery Design**
  High availability is accomplished by the consistent replication and automated repair features of CockroachDB. The data is divided into ranges algorithmically and distributed across nodes. Each range is replicated to nodes synchronously. CockroachDB ensures replication consistency by using *Raft* algorithm which requires a quorum of replicas to agree on any changes to a range before COMMIT [RM17]. It requires at a minimum three nodes in a cluster because three is the smallest number that can achieve a quorum. If the cluster contains three nodes, it can afford the failure of one node [RM17], which is shown in Figure 2.3a. The concept of a raft group is based on one range not for the whole database, and there is one master range but multiple replica ranges. If one node fails, the automated repair mechanism will restart the node and add it to the cluster again. When it rejoins the CockroachDB cluster, ranges are divided and rebalanced across the nodes automatically, which is reflected in Figure 2.3b.



(a) CockroachDB Architecture **(b)** Nodes Self-organization via Rebalancing

Figure 2.3: CockRoachDB HA and DR Features [RM17]

- **CockroachDB Horizontal Scalability**
  The solution for horizontal scaling in CockroachDB is simply incorporating new nodes into the cluster system. There is an example of scaling out from three nodes to five nodes,

[3] CockroachDB: https://www.cockroachlabs.com/
[4] Google Spanner: https://cloud.google.com/spanner

which is represented in Figure 2.4. When adding new nodes (Node 4 and Node 5) into the cluster, ranges are replicated first and rebalanced automatically across the nodes, as shown in Figure 2.3b. As presented in Figure 2.4, CockroachDB uses a peer-to-peer *gossip* protocol to communicate opportunities for rebalancing. This protocol provides an exchange of information between nodes such as storage capacity, network address or other information [RM17].

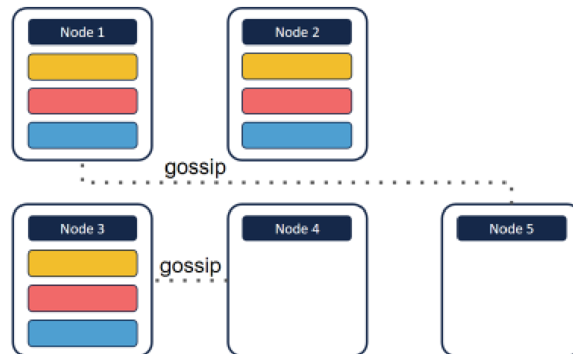


Figure 2.4: New Nodes Connection/Communication via Gossip [RM17]

2.2.2 Google Spanner

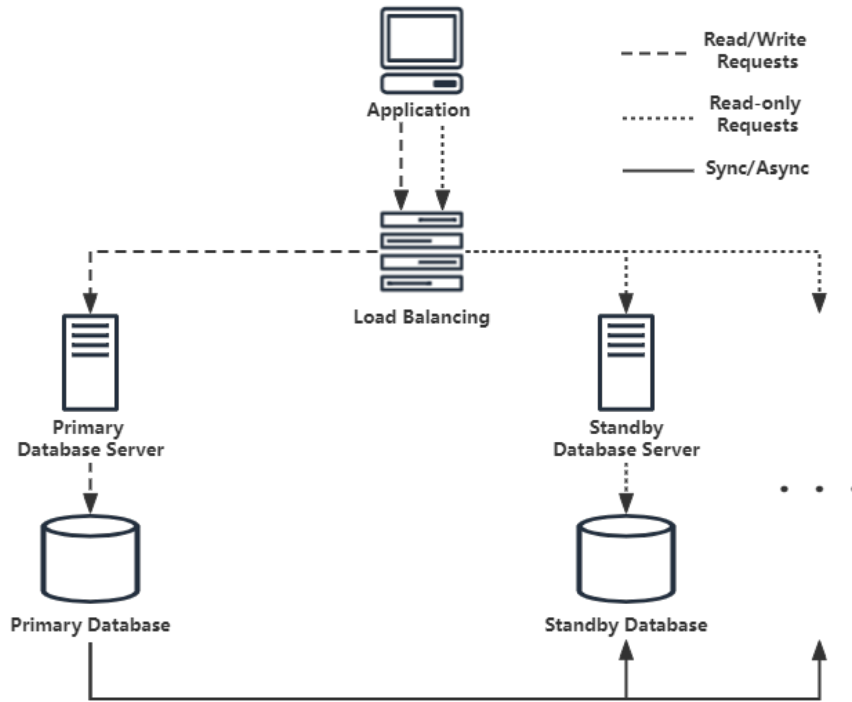
- **High Availability and Disaster Recovery Design**
  Google Spanner uses a similar approach as CockroachDB to achieve high availability. It provides high availability via synchronous replication between replicas in independent zones [spannerHA22]. Each table in the Spanner is broken up into several splits by using ranges of the primary key. These splits are rebalanced and distributed dynamically among different zones based on the amount of data and load. Spanner uses a *Paxos* based replication scheme in which writes are committed only when a majority quorum is reached [spannerHA22]. For each split, there is a *Paxos* group that contains one leader split and several follower splits. When a leader fails, the consensus is redetermined and a new leader is chosen using the *Paxos* algorithm.

- **Spanner Horizontal Scalability**
  Spanner achieves horizontal scalability thanks to its built-in scale-out architecture. It scales horizontally by incorporating new zones into the cluster. Splits are replicated and evenly distributed across all zones when a new zone comes along.

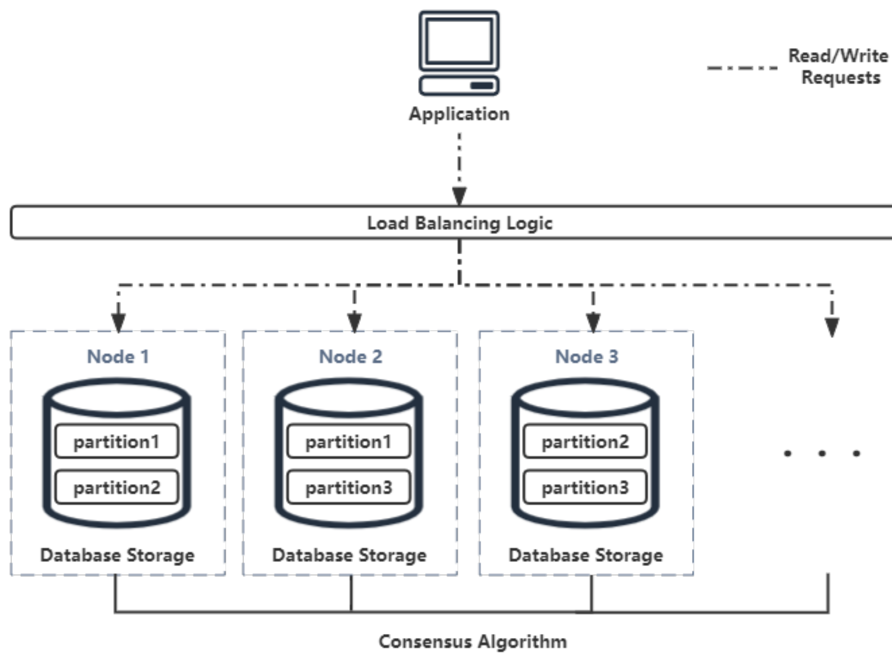
2.3 Comparison Between Traditional RDBMS and Cloud Native RDBMS

2.3.1 High Availability and Disaster Recovery Design

In traditional databases, HA and DR are achieved by an architecture with one primary and multiple standbys, which is shown in Figure 2.5a. When read and write requests come from client applications, they are load balanced and routed to any cluster member or the primary database instance respectively. Load balancing logic based on request type (Create, Retrieve, Update, Delete (CRUD)) is provided by an external custom component. The data can only be modified on the primary server. Due to its shared-nothing architecture, the write operations are replicated to the standby server in a synchronous or optionally asynchronous fashion such to keep data current and consistent. If committing the transaction with the synchronous method, one has to wait until the update of the data on the standby server is acknowledged back to the primary. This will lead to a longer response time depending on the network delay (i.e. geographical distance of the two or more servers). When using asynchronous replication, the individual transaction is not blocked and is committed without delay. The replication will happen later typically for a block of transactions after a certain period (which can be set by the administrator). During this time, dirty reads would be allowed if that was acceptable to the application. If the primary fails, the standby will become the new primary thanks to the failover mechanism.
However, cloud native databases are distributed by architecture and design, and employ a shared-nothing architecture, as presented in Figure 2.5b. The data is split into several partitions, evenly distributed across the nodes after being automatically replicated. Load balancing is provided by a built-in component of the database. Read/write traffic from client applications can be sent to any node. An incoming read request will be routed to the node containing the partition with the data requested. In terms of write operations, a consensus algorithm is used to keep data consistent across the cluster. Once consensus is reached, the written data is available for a read operation from any node in the cluster. If one node containing a partition that acts as master fails, this node will be restarted and rejoin the cluster again. At the same time, a new primary partition will be elected from other replicated partitions.



(a) Traditional Database HA and DR Architecture



(b) Cloud Native Database Architecture

Figure 2.5: Comparison Between Traditional and Cloud Native Database About HA and DR

2.3.2 Horizontal Scalability

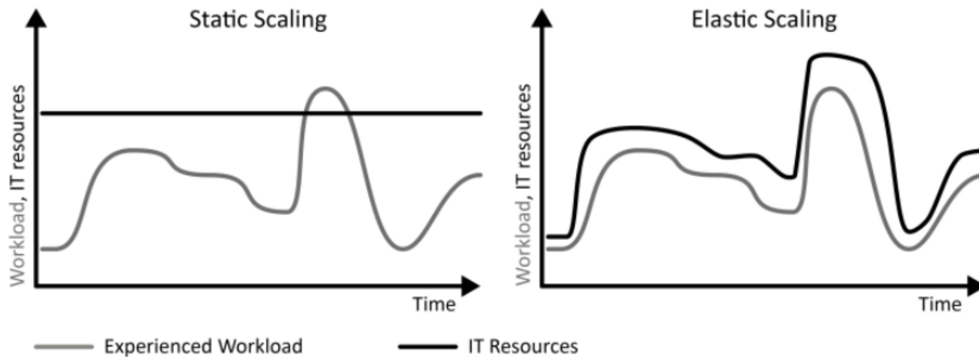


Figure 2.6: Static Scaling and Elastic Scaling for Unpredictable Workload Changes [FLR+14]

Traditional databases based on a shared-nothing architecture achieve their horizontal scaling by including more standby database servers in the cluster, which then synchronize their database data from the primary servers. However, traditional databases that have a shared-everything architecture scale out by simply adding more database servers, since their database data is automatically repartitioned across all cluster members. In the case of deploying traditional database servers on barematel infrastructure, adding more standby servers to achieve horizontal scaling will lead to an increase in the number of IT resources. As shown on the left side of Figure 2.6, the black line represents the number of IT resources that will remain at a certain value after horizontal scaling. The gray line represents the change in the number of workloads. If the workload is less than the number of IT resources (indicated by the gray line under the black line in the diagram), resulting in some IT resources being underutilized. While workloads exceed the number of IT resources (indicated by the gray line above the black line in the diagram), there is a lack of resources to support workloads.

Cloud native databases scale horizontally by inserting new nodes into the cluster. Thanks to the cloud environment, cloud native databases can be elastically scaled in response to changes in workloads and the provisioning and deprovisioning of IT resources via cloud platforms. As illustrated in the right part of Figure 2.6, the number of IT resources grows and shrinks as actual workloads change automatically.

3 Foundations

This chapter lays the necessary foundation for this work, including *Kubernetes* architecture, *Kubernetes* stateful architecture resources and the *Operator*.

3.1 Kubernetes

Containerization in *Docker* promotes the feasibility of running containerized applications on multiple hosts in the cloud environment. The cluster architecture in containers makes it necessary to operate multiple containers on different hosts. The cluster-based containerization in *Docker* creates a need to bridge the gap between the clusters and cluster management [MSK19]. A cluster orchestration platform should be able to monitor the scaling, load balancing and other services of containers residing across different hosts [MSK19]. *Kubernetes* is an open-source container orchestration engine for automating the deployment, scaling, and management of containerized applications [k8s22].

3.1.1 Kubernetes Architecture

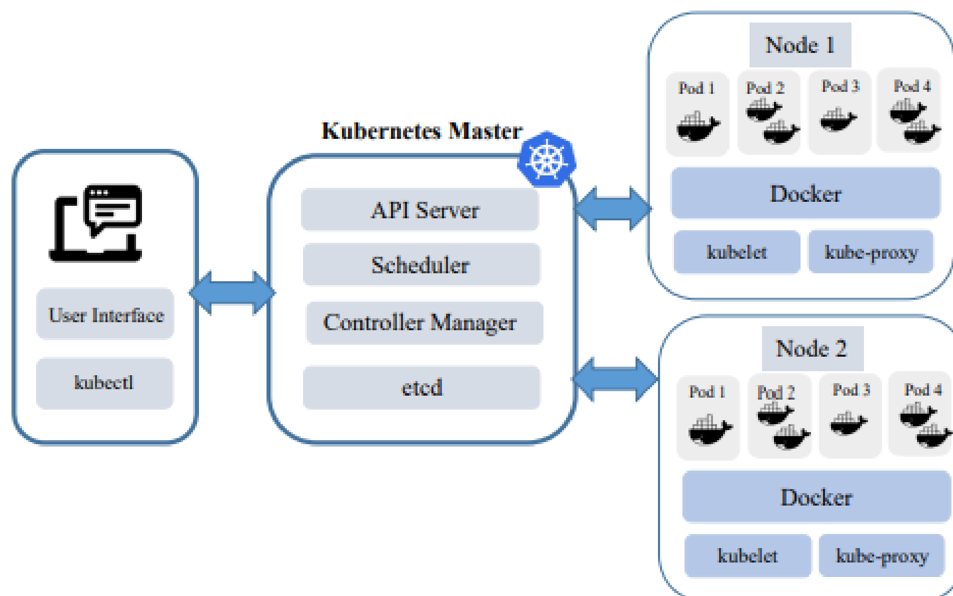


Figure 3.1: Kubernetes Architecture [MSK19]

The *Kubernetes* cluster shown in Figure 3.1 consists of one *Master Node*, and two *Worker Nodes*. When interacting with *Kubernetes*, the command line tool, *kubectl*, is used to communicate with *Master Node* to define and manage the whole cluster lifecycle.

Master Node

The *Master Node*, which is also known as *Control Plane*, is responsible for the management of *Kubernetes* cluster. It is mainly the entry point for all administrative tasks, and hosts the controlling processes that are available for the entire environment [SD19]. By default, there is a single *Master Node* responsible for controlling the cluster, but multiple *Master Nodes* can be utilized to provide high availability [NK20]. The *Master Node* is made up of different components including *API Server*, *Scheduler*, *Controller Manager*, and *ETCD* [NK20].

- **API Server**
  API Server provides an entry point for the *Kubernetes* control plane to control the entire *Kubernetes* cluster. It receives all requests from the client and all other components in the cluster, then authenticates them and updates the corresponding objects in the *Kubernetes's* database.
- **Scheduler**
  Scheduler looks out for unscheduled *Pods* and deploys these *Pods* to an appropriate node in the cluster. The scheduling decision is based on some factors such as resource requirements, hardware/software/policy constraints, and affinity and anti-affinity specifications.
- **Controller Manager**
  Controller Manager continuously watches the shared state of the cluster using *API Server* and tries to alter the current state to the desired state. For example, it is responsible for noticing and responding when nodes go down or ensuring the correct number of running replicas for the application in the cluster.
- **ETCD**
  ETCD is a distributed, consistent key-value store and is used to store all cluster metadata, including configuration data and the state of the cluster.

Worker Node

Applications are running on the *Pods* that are deployed in the *Worker Nodes*. Each *Worker Node* is managed by *Master Node* and contains the following components [NK20]:

- **Kubelet**
  Kubelet is responsible for managing the containers running on the machine. It communicates with the *Master Node* to report current states of the *Worker Node* and obtain decisions from the *Master Node*.
- **Container Runtime**
  Container runtime, such as *Docker*, is used to run containers in *Pods*. It is responsible for pulling the container image from a registry, unpacking the container, and running the container.

- **Kube-Proxy**
  Kube-Proxy runs on each *Worker Node* that implements the *Kubernetes Service*. It maintains the network rules that allow communication to *Pods* from inside or outside the cluster.

3.2 Kubernetes Stateful Architecture Resources

To deploy stateful service successfully in *Kubernetes*, the following components need to be utilized [k8s22; Luk17; MSK19; NK20; VSTK18]:

Namespaces

In *Kubernetes*, *Namespaces* provide a mechanism for isolating groups of resources within a single cluster. Names of resources need to be unique within a *Namespace*, but not across *Namespaces*. Using multiple *Namespaces* allows splitting complex systems with numerous components into smaller distinct groups. They can also be used for separating resources in a multi-tenant environment, splitting up resources into production, development and Quality Assurance (QA) environments. When a *Service* is created, a corresponding Domain Name System (DNS) entry is created as well. This entry is known as Fully Qualified Domain Name (FQDN), in the form of *<service-name>.<namespace-name>.svc.cluster.local*. If a container only uses *<service-name>*, it will resolve to the *Service* which is local to a *Namespace*. But in the case of connecting to *Service* across *Namespaces*, using the fully FQDN is required.

Pod

Kubernetes manages resources in the unit of *Pods*, which are the smallest logical unit. In the *Pod*, there is a group of one or more containers with shared storage and network resources, and a specification for how to run the containers. In terms of *Docker* concepts, a *Pod* is similar to a group of *Docker* containers with shared *Namespaces* and shared filesystem volumes. The *one-container-per-Pod* model is the most common *Kubernetes* use case. A *Pod* is considered as a wrapper around a single container. It can encapsulate an application composed of multiple co-located containers that are tightly coupled and need to share resources. *Kubernetes* provides *Deployment* and *StatefulSet* as implementation artefacts so called "workload resources" to define, create and manage multiple *Pods*. *Pods* are created from a *PodTemplate* written in the YAML file containing the definition and their desired state of components and associated services. The *Kubernetes* controller handles resource replication, deployment and automatic healing in the event of *Pods* failure.

Service

Since the IP address of *Pods* changes every time they are restarted, it is difficult to access a *Pod* directly using its IP address. A *Service* is an abstraction for a group of *Pods*. It is bound to a *ClusterIP*, which is a virtual IP address that never changes. When clients connect to the *ClusterIP* which is only reachable from within the cluster, their traffic is automatically transferred to an

appropriate backend *Pod*. To access the cluster from outside, there are two typical types of *Service* in *Kubernetes*: *NodePort* and *LoadBalancer*. Concerning the *NodePort* service, *Kubernetes* opens a static port on each node called *NodePort*. The service can be accessed from outside the cluster by using the IP address of the node and the NodePort, like *<NodeIP>:<NodePort>*. The *LoadBalancer* service works when using a cloud provider for the *Kubernetes* cluster. The cloud provider configures the load balancer in its network to proxy the *NodePort* on multiple nodes, and the load balancing algorithm depends on the cloud provider's implementation. In addition, the *NodePort* and *ClusterIP* services are also created automatically together with *LoadBalancer*, and are used to redirect the external and internal traffic respectively to an appropriate *Pod* in the cluster.

Endpoints

Endpoints in *Kubernetes* are objects that get or store one or more IP addresses of *Pods* that are assigned to them dynamically along with ports as well. If the service selector matches a pod label, *Kubernetes* will automatically create an *Endpoints* object with the same name as the *Service*, which stores the IP address and port of the *Pod*.

StorageClass

StorageClass is a *Kubernetes* storage mechanism that allows to provision *Persistent Volume* dynamically in a *Kubernetes* cluster. Different classes might map to different quality-of-service levels, or backup policies, or arbitrary policies determined by the cluster administrators.

Persistent Volume Claim

A *Persistent Volume Claim (PVC)* is a request for storage, on the specific size, *StorageClass* and access modes, etc. Access mode can be either *ReadWriteOnce*, or *ReadOnlyMany*, or *ReadWriteMany*. With *ReadWriteOnce* mode, only a single node can read or write the volume. In the case of *ReadOnlyMany* mode, many nodes can only read the volume. However, many nodes can read or write the volume in *ReadWriteMany* mode.

Persistent Volume

A *Persistent Volume (PV)* is a unit of storage in the cluster that has been provisioned by an administrator or dynamically provisioned using *Storage Classes*. *PVs* can be provisioned statically, and created by the administrator manually at configuration time. Dynamic provisioning is based on *StorageClass* which is defined at configuration time. At run time, *Kubernetes* allocates *PVs* if there is a *PVC* it matches to. *PVs* are independent of the lifecycle of the *Pod* that uses them, meaning that even if the *Pod* dies, the data in the volume is not erased.

StatefulSet

StatefulSet manages the deployment and scaling of a set of *Pods* where stateful applications are running and provides guarantees about the ordering and uniqueness of these *Pods*. A *StatefulSet* manages *Pods* that are based on an identical container specification, but maintains a sticky identity for each of their *Pods*. In other words, these *Pods* are not interchangeable since each *Pod* has a persistent identifier that it maintains across any rescheduling. *StatefulSet Pods* have a unique identity that is comprised of an ordinal, a stable network identity, and stable storage. Each *Pod* in a *StatefulSet* derives its hostname from the name of the *StatefulSet* and the ordinal of the *Pod*. The pattern for the constructed hostname is *$(statefulset name)-$(ordinal)*. Every *Pod* has its own stable *PV* either by default or as defined per *Storage Class*. The data in *PV* will survive even when all *Pods* die.

3.3 Operator

There are no primitives in *Kubernetes* to manage states by default. As a result, relying on *Kubernetes* primitives alone brings difficulty managing stateful application requirements such as replication, failover automation, backup/restore and upgrades. The *Operator Pattern* is introduced and can be used to solve the problem of managing state [OTH+21].

3.3.1 Operator Design Pattern

The operator design pattern defines how to manage application and infrastructure resources using domain-specific knowledge and a declarative state. With this pattern, the amount of manual imperative work is reduced, such as backup and upgrade, which is required to keep an application in the desired state [OTH+21].

As presented in Figure 3.2, a general operator has a software (called *Controller*) that reads the desired specification and creates and manages the resources that were described by a domain-specific language of the given custom resources. The *Operator Pattern* contains the following three components [OTH+21]:

- **Managed Resources**
  Managed resources are applications or infrastructure that we want to manage.
- **Domain Specific Language**
  A *Domain Specific Language* allows specifying the desired state of *Managed resources* in a declarative way.
- **Controller**
  A *Controller* runs continuously to read and be aware of the desired state, report the current state of *Managed resources* and apply changes to them in an automated way.

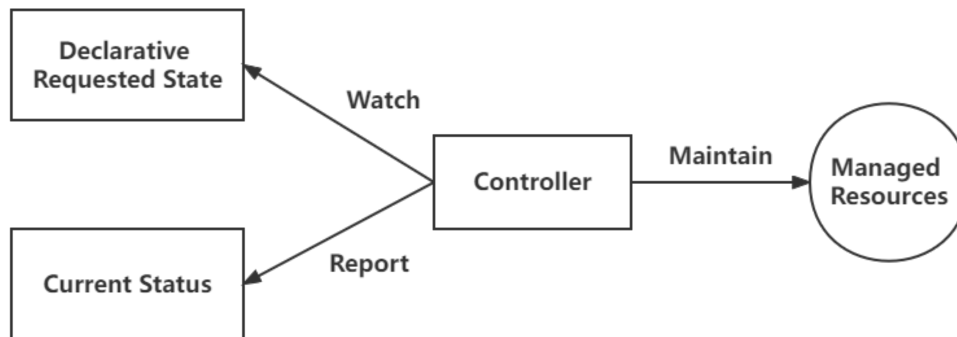


Figure 3.2: General Structure of an Operator [OTH+21]

3.3.2 Operator Components in Kubernetes

Operators are custom software extensions to *Kubernetes* that make use of *Custom Resource* to manage applications and their components in an application specific way [k8s22; OTH+21]. This is achieved by combining *Kubernetes Controllers* with watched objects that describe the desired state. There are three *Operator* components in *Kubernetes*:

- **Controller**
  The controller, which is the brain of the operator, can watch one or more objects. The objects can be either *Kubernetes* primitives such as *Deployments*, *Services* or things that reside outside of the *Kubernetes* cluster such as Virtual Machine (VM)s. To ensure the watched objects get transitioned to the desired state in a defined way, the controller will continuously compare the desired state with the current state using the reconciliation loop and keep them consistent.
- **Custom Resource**
  Custom Resource (CR) allows to extend *Kubernetes API* with additional types not available in the default *Kubernetes* distribution. Once a *CR* is installed, it can be queried via API and manipulated by *kubectl*, like native types, such as *Pods* or *Service*. The desired state is encapsulated in *CRs* and the controller applies changes to them to get their target state. *Custom Resource* consists of one or more *Kubernetes* objects, such as *Pod*, *StatefulSets* and *Service*. Therefore, updating the state of a *CR* is actually updating the state of corresponding *Kubernetes* objects.
- **Custom Resource Definition**
  Custom Resource Definition (CRD) API resource tells *Kubernetes* how to create a *CR* about what the new resource kind is, what its specification looks like and how to validate its fields.

The diagram in Figure 3.3 illustrates the architecture of the operator in *Kubernetes* and how it works. *CRD* defines a schema of settings available for configuring a *CR* that is an instance of *CRD*. When a user modifies the *CR* configuration file with the desired state, the controller will get notified about changes. There is a reconciliation loop in the controller that repeatedly compares the desired state of the *CR* to its current state. If these two states don't match, the controller takes action to adjust

the current state to match the desired state as expressed in the *CR* configuration file. In order to achieve the desired state of the *CR*, the controller applies changes based on the user's modification to *Kubernetes* objects that compose the *CR* to update their states.

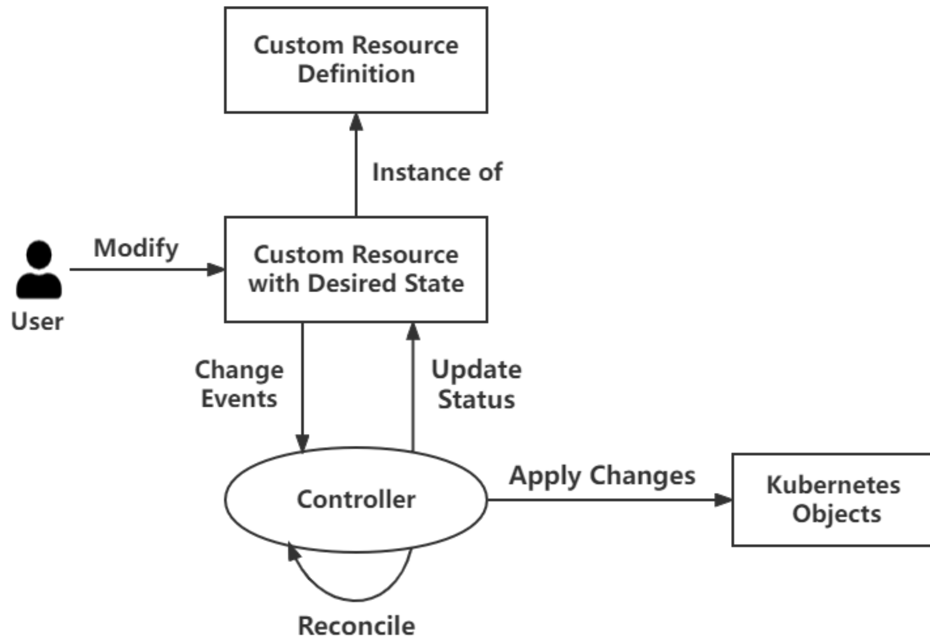


Figure 3.3: Kubernetes Operator Architecture and Mechanism

3.3.3 Operator SDK

Operators make it easy to manage complex stateful applications on top of *Kubernetes*. However, writing a *Kubernetes* operator today might be difficult because of challenges such as using low-level APIs, writing boilerplate, and a lack of modularity which leads to duplication [oprSDK20]. *Operator SDK*, is an open-source toolkit whose main purpose is to build *Kubernetes* operators. The set includes the operator-sdk utility, which provides a list of commands for generating an operator template for any type of CR. The SDK imposes a standard project layout, and in return creates skeletal Go source code for the basic *Kubernetes* API controller implementation and placeholders for application-specific handlers. In addition, the SDK provides convenience commands for building a *Kubernetes* operator and wrapping it in a Linux container, generating the YAML-format *Kubernetes* manifests required for deployment [DW20].

4 Related Work

This chapter describes related works conducted by previous master students of the University of Stuttgart, and which were used to provide the foundation for this thesis.

4.1 Design Changes for Decomposing Monolithic ECM Systems

According to the degree of coupling, Shao [Sha20] divided monolithic ECM applications into independent components. This allowed confining loosely coupled components into distinct containers, whereas tightly coupled ones are put together in the same container. These components run within containers and are packaged with all necessary dependencies and libraries. This division enables the opportunity for continuous delivery, continuous integration, and cost-effective scaling. Figure 4.1 shows the topology of the prototype developed by Shao. Four independent applications running on Docker containers form the ECM platform. *rmdbsrv* includes the *Object Catalog*, and *lsdbsrv* includes the *Data Catalog*. Both containers are built using IBM's publicly available Docker image: *ibmcom/db2:latest*. The *Resource Manager Application* and an HTTP server are both included in *wasrm*. *wasicn* consists of an HTTP server, a web client, and its configuration database. They had to be constructed manually based on the image: Centos 7. Users can communicate with the system through the *Web Client*, which sends requests to or receives data from the *Data Catalog*, *Resource Manager* and *Object Catalog*.

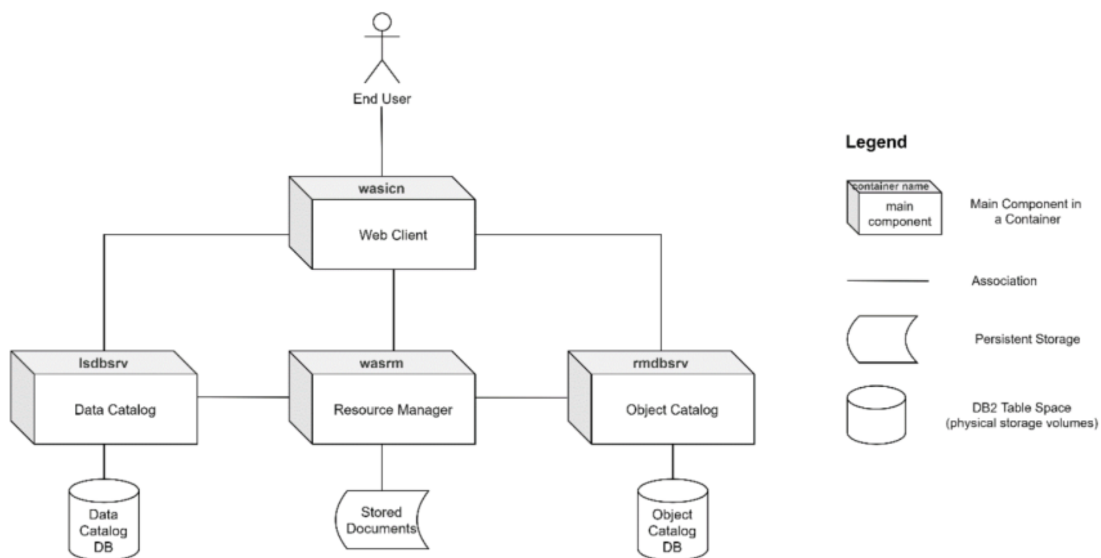


Figure 4.1: Topology of ECM System Developed by Shao [Sha20]

4.2 Deploying ECM Workloads in Cloud Environments Based on Kubernetes and Docker

Based on the prototype developed by Shao [Sha20], Trybek [Try21] migrated containerized ECM solution into a *Kubernetes* cluster. The solution can be divided into two categories, web applications (i.e. *Web Client* and *Resource Manager*) represent stateless components, while the database components (i.e. *Resource Manager* and *Object Catalog*) are stateful services. Managing stateful services like databases in a *Kubernetes* cluster is not simple, as *Kubernetes* was initially intended for stateless workloads. Therefore, Trybek constructed a prototype that left stateful components outside the *Kubernetes* cluster, as presented in Figure 4.2. *Web Client* and *Resource Manager* provide the external *Service* with *NodePort* type to users and administrators respectively. The *Web Client* requires internal connections to *Object Catalog* as well as *Resource Manager*. In addition, *Resource Manager* connects to *Object Catalog* and *Data Catalog* via an internal *Service*. *Object Catalog* and *Data Catalog* are operated as *Docker* containers and connect to database services which are integrated into the *Kubernetes* cluster through *Endpoints*.

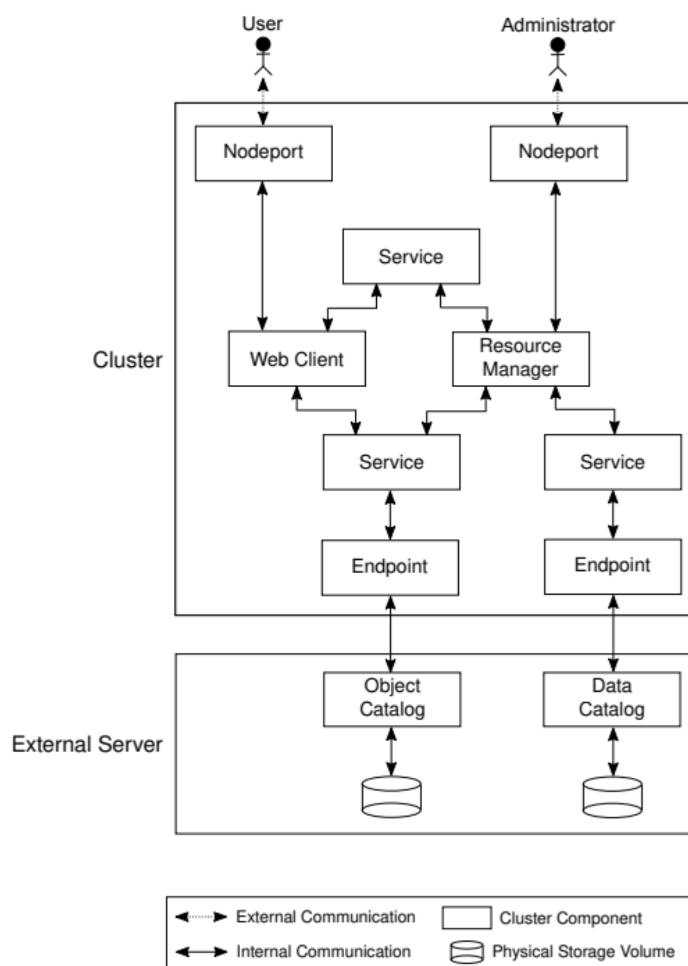


Figure 4.2: Topology of ECM System Inside a Kubernetes Cluster Developed by Trybek [Try21]

5 The MAPE Workload Management Concept

This chapter describes at a high level the concepts behind this thesis for developing a *Kubernetes* operator, responsible for managing a cluster for DB2 to provide stable stateful database services. To start with, we introduce the MAPE concept proposed by Ritter et al. [RMM12] and explain how we used it to implement a *Kubernetes* operator for DB2 based on MAPE with the *Kubernetes Operator SDK*.

5.1 MAPE Concept

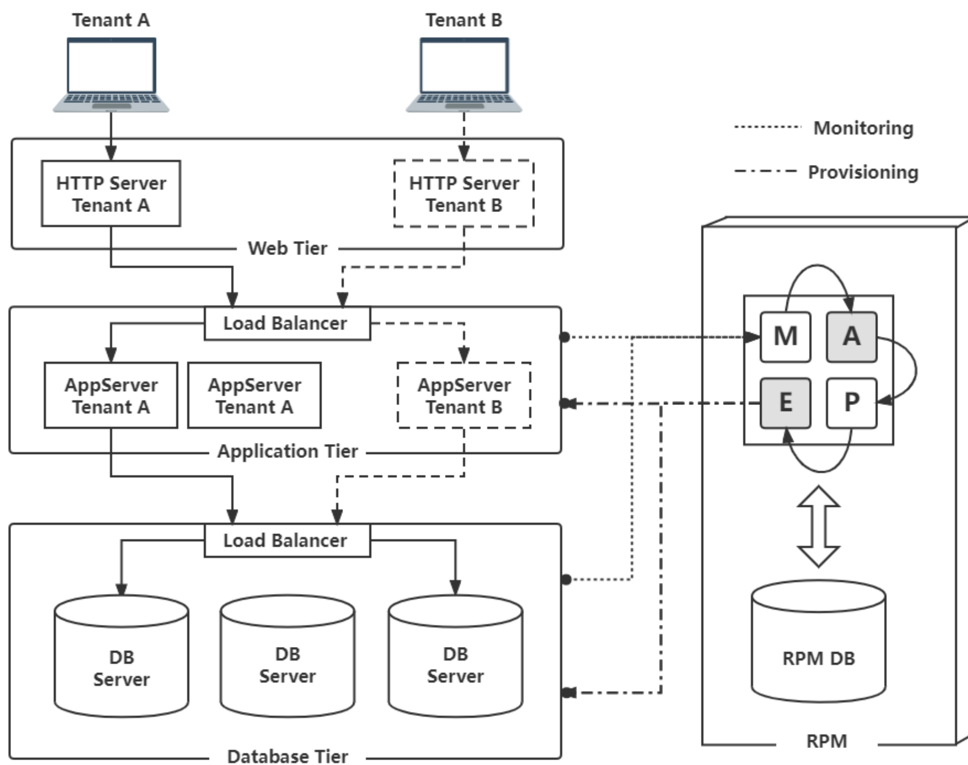


Figure 5.1: System Topology Applying the MAPE Loop Concept [RMM12]

Ritter et al. [RMM12] presented MAPE, a loop-based dynamic and cost-effective provisioning framework for multi-tenant capable system topologies in 2012. Figure 5.1 shows a typical architecture of a common three-tier system: *Web Tier*, *Application Tier* and *Database Tier*. A

framework called Resource Provisioning Manager (RPM) built upon the MAPE loop concept, monitors and manages (dynamically provisions) critical resources (application servers, database servers). MAPE is a generic control-loop concept that uses four phases to control a target system described as follows [RMM12]:

- **Monitor**
  During the monitor phase, actual resource workloads for application and database servers on a per tenant basis are gathered, aggregated and persisted to the RPM database.
- **Analyze**
  In the analyze phase, the data resulting from the monitor phase is used to predict tenant-specific workload peaks for critical system resources and future workloads.
- **Plan**
  The plan phase compiles a plan of execution actions that adjust the system topology to achieve the target values based on the results of the analyze phase.
- **Execute**
  During the execute phase, the plan from the plan phase is finally executed by an orchestrator that drives provisioning and de-provisioning workflows.

5.2 MAPE Implemented by Kubernetes Control Loop

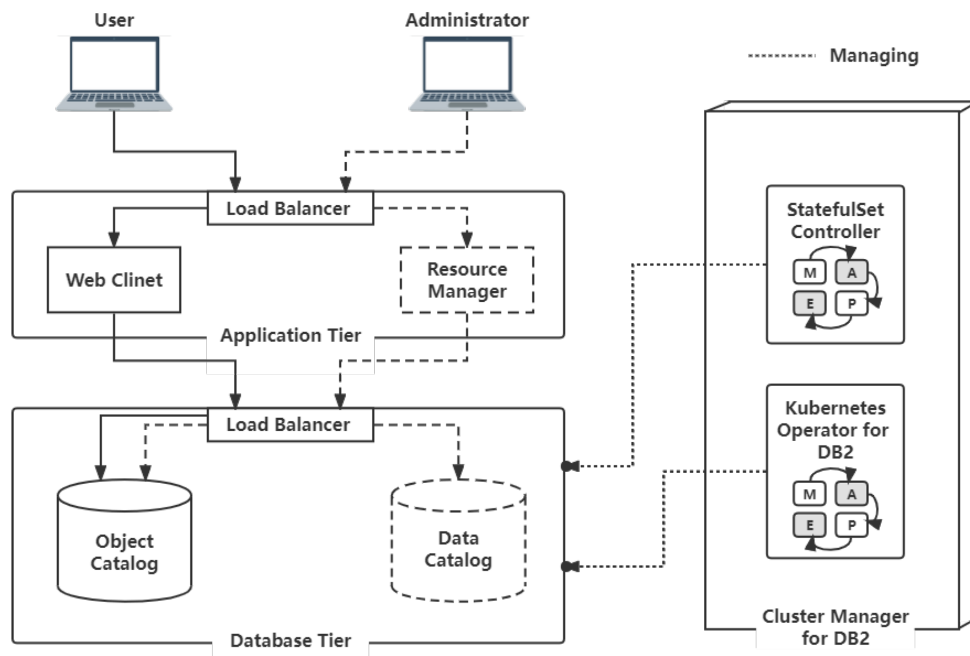


Figure 5.2: ECM System Topology Refactored by Shao [Sha20] and Trybek [Try21] Applying a Cluster Manager for DB2 Based on the MAPE Loop Concept in *Kubernetes*

We compared the MAPE with the *Kubernetes* design concept and found that the *Kubernetes* control-loop together with the implementation of the Operator pattern do fit very well together as Figure 5.2 outlines. This work proposes an approach to orchestrate DB2 database applications as stateful services using a custom operator, namely, a *Kubernetes* operator for DB2. It is the missing piece for the ECM system refactored by Shao [Sha20] and Trybek [Try21]. We develop and utilize a cluster manager for DB2 to manage the cluster containing DB2 databases based on MAPE theory proposed by Ritter et al. [RMM12]. Figure 5.2 shows the ECM system topology refactored by Shao [Sha20] and Trybek [Try21] now using the cluster manager for DB2. The cluster manager for DB2 is a logical component that consists of a *StatefulSet Controller* and a *Kubernetes* operator for DB2. We utilize a *StatefulSet* to deploy DB2 database applications, as *StatefulSet* manages stateful applications in *Kubernetes*. The *StatefulSet Controller* is a built-in *Kubernetes* controller managing *Pods* in the *StatefulSet* in terms of creation, termination, recreation, rolling updates, etc. Nevertheless, the *Kubernetes* operator for DB2 is a custom operator that is responsible for managing the *Pods* hosting the DB2 database instances which together provide the HA characteristic and the DR feature. Both controllers employ control loops to manage the state of their resources, as described in the MAPE control-loop concept. The MAPE control loop phases implemented by *Kubernetes* are summarized below:

- **Monitor**
  During the monitor phase, the controller monitors the target resource and notifies its reconciler of a change event once the resource has changed.
- **Analyze**
  In the analyze phase, the reconciler analyzes whether the current state of the resource is the same as the desired state.
- **Plan**
  The plan phase compiles a plan of executive actions that describes how to communicate with *Kubernetes API Server* to adjust the resource to achieve the desired state.
- **Execute**
  During the execute phase, *Kubernetes API Server* executes the actions finally which are in the plan from the plan phase.

6 Prototype

This chapter describes how to design, develop, test and evaluate our prototype for orchestrating stateful database services in *Kubernetes*. Section 6.1 describes the design details regarding the cluster for DB2 and the *Kubernetes* operator for DB2. The implementation of them is introduced in Section 6.2, and test scenarios and evaluations of our stateful database services are presented in Section 6.3.

6.1 Design Approach

In the following sections, the design of the stateful database service prototype is discussed. Section 6.1.1 describes the overall design of the prototype. Next, the design of the two most important components of the prototype: i) a cluster for DB2 and ii) a *Kubernetes* operator for DB2, will be elaborated in Section 6.1.2 and Section 6.1.3 respectively.

6.1.1 Design Overview

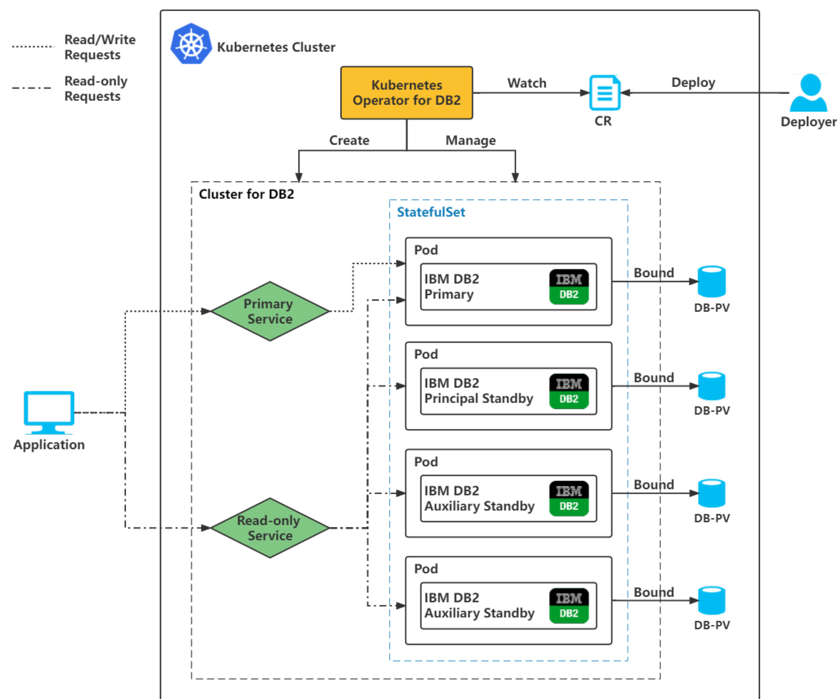


Figure 6.1: Topology of Stateful DB2 Database Service Prototype Inside a Kubernetes Cluster

Figure 6.1 shows the topology of the DB2 stateful database service prototype which is based on the concept introduced in Chapter 5. The prototype contains two components, a *Kubernetes* operator for DB2 that creates and manages a cluster for DB2 based on user-defined CR-yaml file, and the cluster for DB2 that is able to provide DB2 stateful database services for external applications.
After writing a CR-yaml file that is an instance of the CRD named *DB2Cluster*, the deployer deploys it in the *Kubernetes* cluster. Once a CR is created by *Kubernetes*, the *Kubernetes* operator for DB2 will create a CR namely a cluster for DB2, based on the specification of the CR-yaml file. In addition, it will continuously monitor the current state of the cluster for DB2 and compare it with the desired state. If the current state of the cluster for DB2 is not the same as the desired state, the *Kubernetes* operator for DB2 will take action to update the state of components contained in the cluster for DB2 to achieve the desired state.
In the cluster for DB2, *StatefulSet* which is actually a *Kubernetes controller* manages and maintains four *Pods* containing DB2 database applications. A database application is running on a container in the *Pod*. Under the management of *StatefulSet*, each *Pod* has an ordered, stable identity and unique network identifier and is bound to a PV. If a pod dies, *StatefulSet* will recreate it with the same identity and rebound it to the previous PV containing all database data. These four *Pods* play different roles in the cluster for DB2: one is the primary, one is the principal standby and up to two others are auxiliary standbys. The primary accepts both read and write requests, while all of them support read-only requests. Although the data can only be changed in the primary database, the primary does synchronize the changes with all standbys via log shipping. A built-in feature of IBM DB2 was implemented to support HA and DR scenarios.

6.1.2 Design of the Cluster for DB2

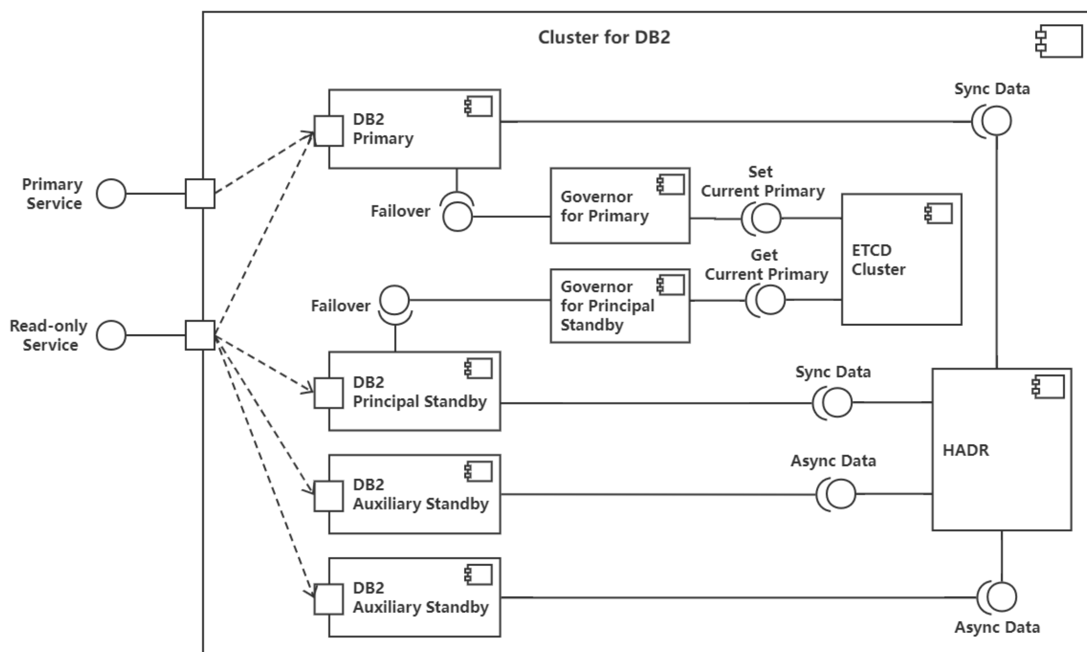


Figure 6.2: Component Diagram of a Cluster for DB2

The cluster for DB2 is expected to provide stable and reliable stateful database services supporting HA, DR and read-scalability. Figure 6.2 shows, the cluster for DB2 contains four DB2 *Pods*, two *Governor* components for the primary and the principal standby respectively, an *ETCD Cluster* and a HADR framework. The following subsections will discuss in detail how HA, DR and read scalability are achieved.

HA

The HA strategy adopted in this work is based on the DB2 built-in HADR feature presented in Figure 6.2. This feature is fully integrated and requires no special hardware or software. Since this DB2 Enterprise Edition supports the shared-nothing architecture shown in Figure 6.1, i.e. each database uses its own storage. Thus, the HA is achieved by replicating data changes from the primary database to all standby databases to keep them in sync, and in case of a primary failure, a failover event from primary to standby is triggered.

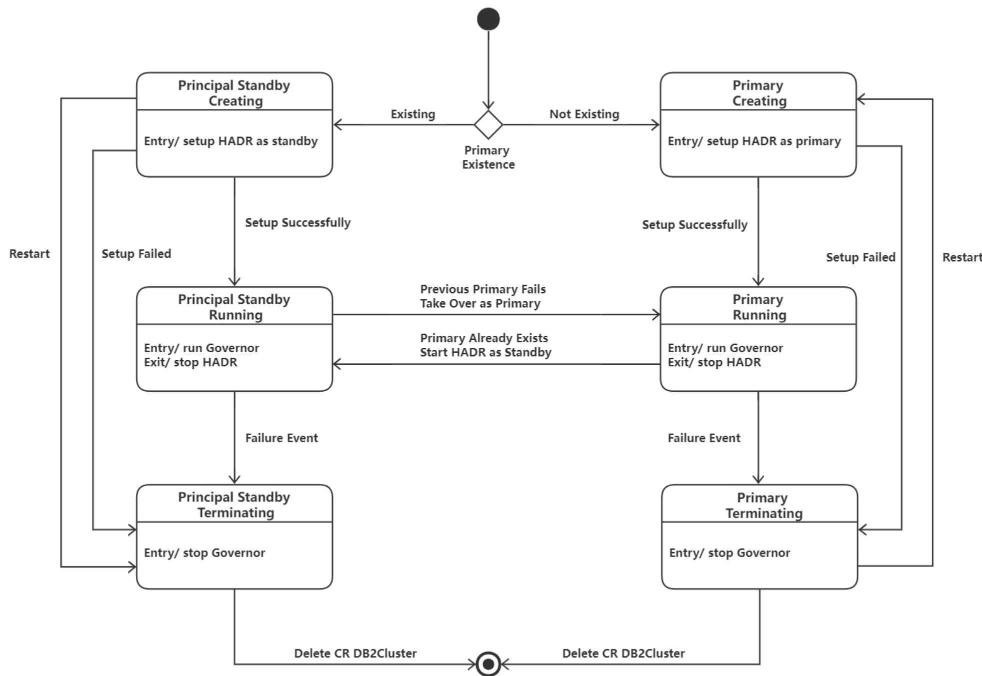


Figure 6.3: State Diagram of *Pods* Related to Failover

Failover is the process of transferring ownership of database services from a failed server to a healthy one. Failover can be manual or automated. Relying on manual failover might result in higher downtime compared to automated failover. Therefore, in our prototype, we implement a component called *Governor* that is responsible for performing failover automatically. Due to the limitation of the HADR implementation, auxiliary standbys can only asynchronize data with the primary database. The failover between the auxiliary standby and the primary will result in the loss of certain data that was not synchronized in time. Thereby, we merely consider the automated faiover between the principal standby and the primary. For both the primary and the principal

standby, there is a *Governor* running on it separately, shown in Figure 6.2. The *Governor* monitors the state of the primary and conducts the failover once the primary fails. The *Kubernets* built-in *ETCD Key-Value Database Cluster* stores the data including the database instance designated as the current primary, in the form of <*CurrentPrimary*, *hostname*> (i.e.<key, value>).

The overall process of failover for DB2 *Pods* with state transfer aspect is presented in Figure 6.3. During the initial setup phase, if there is no primary in the cluster for DB2, the *Pod* will be set up as the primary. Once the primary *Pod* is running, *Governor* running on it will set the value of *CurrentPrimary* as its hostname and continuously check its health. If a primary already exists, the next activated *Pod* will be assigned the principal standby role and linked with the primary as its peer by the HADR framework. The *Governor* on the principal standby will take no action since it confirms that it is not the current primary and the current primary is healthy now. Once the primary fails due to a failure event, such as failed network/servers and the deletion of CR (DB2Cluster), the automatic failover is triggered. In the failover phase, the *Governor* for the principal standby finds the unhealthy state of the primary, then it will take over as the new primary and set its hostname as the new value of *CurrentPrimary* in the *ETCD Cluster*. When the previous primary comes back, *Governor* on it will solve the split-brain problem. The previous primary's *Governor* obtains the value of *CurrentPrimary* and detects that it is no longer the current primary. It also finds that the new primary is healthy now. Based on these two conditions, the *Governor* for the previous primary will reconfigure the previous primary as the new principal standby to rejoin the cluster for DB2.

DR

Benefiting from HADR, the cluster for DB2 implements DR feature by replicating data from the primary to two auxiliary standbys in asynchronization mode, which is presented in Figure 6.2. Therefore, auxiliary standbys exist as database backups of the primary. In addition, the primary and auxiliary standbys are deployed in two separate locations. In the event that both the primary and the principal standby stop working due to a disaster, DB2 database services can be provided again after performing a disaster recovery plan manually. The disaster recovery plan restores database services by means of recreating the primary and the principal standby from two auxiliary standbys.

Read scalability

In this work, the strategy for horizontal scaling of the cluster for DB2 is increasing the number of standbys that exclusively serve read queries, as presented in Figure 6.2. This approach for horizontal scalability also improves the total read performance for the read-only service. Compared with only one standby, scaling to three standbys helps reduce the number of read requests each standby needs to handle.

6.1.3 Design of the *Kubernetes* Operator for DB2

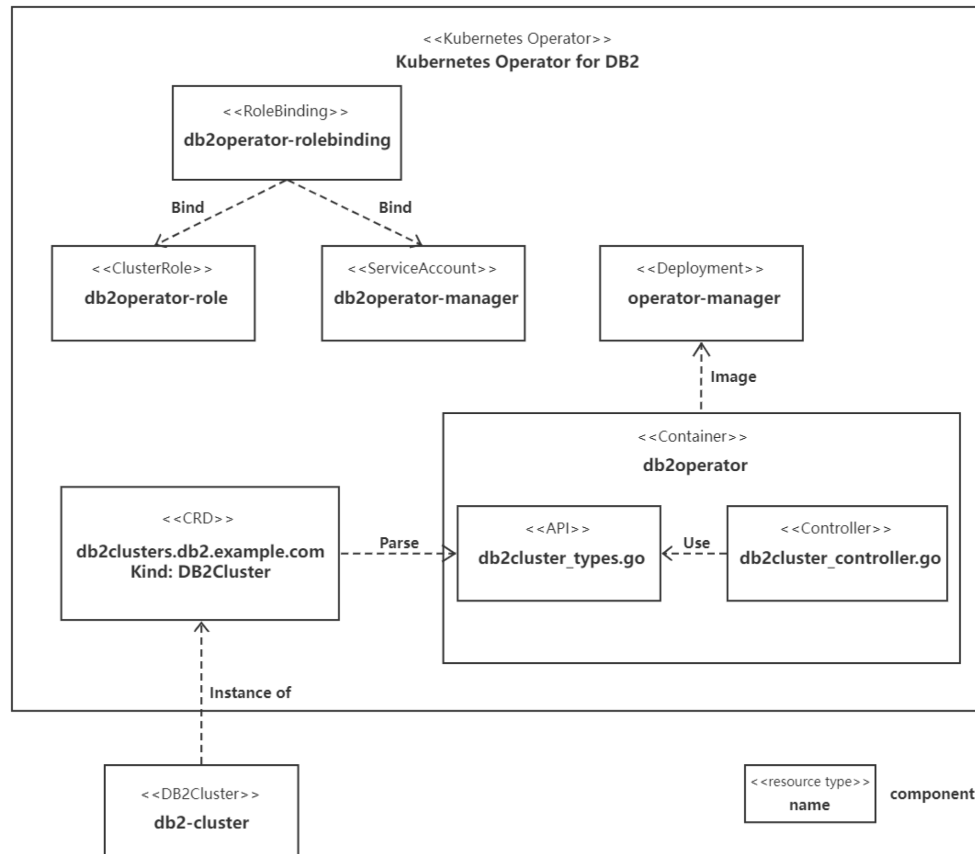


Figure 6.4: Structure of a *Kubernetes* Operator for DB2

The goal of the *Kubernetes* operator for DB2 is deploying a cluster for DB2 based on the *CR* specification automatically, continuously monitoring and updating its state. As illustrated in Figure 6.4, the following components form the three main parts of the *Kubernetes* operator for DB2:

- **API**
  API describes the configuration of a cluster for DB2, including programmatic API (i.e. *db2cluster_types.go*), CR and CRD. The CRD defines a schema of settings available for configuring a cluster for DB2. The component *db2cluster_types.go* defines the same data schema as the CRD but is implemented using GO programming language. CR specifies values for the settings defined by the CRD. The relationship between them is clearly shown in Figure 6.4, the CRD named *db2clusters.db2.example.com* can be generated by parsing definitions in the *db2cluster_types*, and the CR is an instance of CRD.

- **Controller**
  The controller is the most important component of the *Kubernetes* operator for DB2. The controller creates managed resources by referring to the definition in the *db2cluster_types.go*

and using the corresponding specific value in the CR. In the controller, the reconcile loop is implemented which is responsible for enforcing the desired state on the actual state of the cluster for DB2.

- **Role and Service Accounts**
  Role-based access control (RBAC) is a method of controlling access to *Kubernetes* resources based on the roles of individual users. As shown in Figure 6.4, a *ServiceAccount* called *db2operator-manager* provides the *Kubernetes* operator for DB2 running in the *Pod* with the identity that *Kubernetes* uses for authentication. The *ClusterRole* named *db2operator-role* specifies the permissions that allow the *Kubernetes* operator for DB2 to interact with the resources it manages, such as create, get and delete. Finally, *db2operator-rolebinding* which is a *RoleBinding* object binds the *ServiceAccount* with *ClusterRole*, i.e. the *Kubernetes* operator for DB2 has the specified permission to access *Kubernetes* resources.

6.2 Implementation

Section 6.2.2 The following section describes the implementation of the prototype which is designed in Section 6.1. Section 6.2.1 and Section 6.2.2 introduces the implementation details of the cluster for DB2 and the *Kubernetes* operator for DB2 respectively.

6.2.1 Implementation of the Cluster for DB2

Pods that comprise containers running a DB2 database application form a cluster for DB2. In addition, containers are dependent on images to use them to construct a run-time environment and run an application. In order for the cluster to have HA and DR characteristics, we need to containerize the DB2 database application as an image, which is set up with the DB2 HADR feature and has included our *Governor* component developed to manage automatic failover.

Initializing HADR

In this work, a shell script named *setup_db2_instance.sh* is adopted to initialize each DB2 instance in the cluster for DB2. DB2 instances will be set up HADR with different HADR roles, such as primary, principal standby and auxiliary standby. The HADR roles of them rely on the number of instances in the cluster for DB2. To achieve the HA feature realized by HADR, the cluster for DB2 must consist of at least two DB2 instances, one is the primary and the other is the principal standby. If there are three DB2 instances existing in the cluster for DB2, in addition to the primary and the principal standby, the other one is set up as the auxiliary standby. Consequently, DR capability will be enabled. In the case of four DB2 instances, aside from the required primary and the principal standby, the other two are auxiliary standbys. The pseudocode for setting up HADR on a DB2 instance is presented in Algorithm 6.1. First, the HADR configuration file: */hadr/hadr.cfg* is mounted on a shared persistent volume, as each instance needs to write HADR configuration to it. There is a potential for write conflicts owing to each instance writing to the */hadr/hadr.cfg*. Therefore, before each DB2 instance writes to */hadr/hadr.cfg*, it examines if the file is locked (i.e. if another instance is writing). If the file is unlocked, the DB2 instance writes

directly, otherwise, it waits until it is unlocked. The example in Listing 6.1 shows the format of */hadr/hadr.cfg*. When the first instance finds that */hadr/hadr.cfg* is empty, then it will write its hostname and IP address into */hadr/hadr.cfg* as primary. According to the number of DB2 instances in the cluster specified in the CR-yaml file, the primary calculates the number of standbys and learns about their HADR role types. Moreover, the primary checks the keywords (i.e. standby_hostname, standby1_hostname, standby2_hostname) in */hadr/hadr.cfg* every 30 seconds to determine if all standbys are signed up. Otherwise, the primary will wait until all standbys complete the registration of configuration information. Once all standbys have written the configuration information in */hadr/hadr.cfg*, the primary will set the HADR configuration parameters, and then start HADR on the database as the primary. Subsequent instances find that there is already existing information about the primary instance in */hadr/hadr.cfg*, then their roles will be initialized to standby. The role of the standby instance is principal standby or auxiliary standby depending on the number of instances and the existing configuration in the */hadr/hadr.cfg*. The setting priority of the principal standby is higher than that of the auxiliary standby. In other words, the DB2 instance activated earlier will be configured as the principal standby first. Similar to the primary, the standbys also constantly check the keywords of */hadr/hadr.cfg* to examine whether the remaining DB2 instances are registered in it. After waiting for all instances to write configuration, the standby will also set the HADR configuration parameters. However, before starting HADR on the database as a principal standby or auxiliary standby, enabling read-only mode for the standbys is required. *setup_hadr.sh* allows the standbys to be readable via using the commands: "DB2SET DB2_HADR_ROS=ON" and "DB2SET DB2_STANDBY_ISO=UR".

Listing 6.1 Example of */hadr/hadr.cfg*

```
primary_hostname=db2-0
primary_ipaddr=192.168.54.121
standby_hostname=db2-1
standby_ipaddr=192.168.182.222
standby1_hostname=db2-2
standby1_ipaddr=192.168.54.78
standby2_hostname=db2-3
standby2_ipaddr=192.168.182.219
```

Algorithm 6.1: Setting up HADR on DB2 Instances with Different HADR Roles

input : The number of DB2 instances num, The name of DB2 database where set up HADR on $dbName$

output : A DB2 instance that is set up with HADR

```
1  F_hadr ← /hadr/hadr.cfg ;                    /* HADR configuration file */
2  if F_hadr is empty then
3      write hostname and IP to F_hadr as primary;
4      wait for all standbys to write in F_hadr;
5      set HADR configuration parameters for primary;
6      start HADR on database dbName as primary;
7  else
8      if num = 2 then
9          write hostname and IP to F_hadr as principal standby;
10         standbyRole ←principal standby;
11     if num = 3 then
12         if info of principal standby does not exist in F_hadr then
13             write hostname and IP to F_hadr as principal standby;
14             standbyRole ←principal standby;
15         else
16             write hostname and IP to F_hadr as auxiliary standby1;
17             standbyRole ←auxiliary standby1;
18     if num = 4 then
19         if info of principal standby does not exist in F_hadr then
20             write hostname and IP to F_hadr as principal standby;
21             standbyRole ←principal standby;
22         else if info of auxiliary standby1 does not exist in F_hadr then
23             write hostname and IP to F_hadr as auxiliary standby1;
24             standbyRole ←auxiliary standby1;
25         else
26             write hostname and IP to F_hadr as auxiliary standby2;
27             standbyRole ←auxiliary standby2;
28     wait for all instances to write in F_hadr;
29     set HADR configuration parameters for standbyRole;
30     enable read-only mode for the standby;
31     start HADR on database dbName as standbyRole;
```

Setting HADR configuration parameters is the most critical step during setting up HADR. The following discussion is about how to set HADR configuration parameters in the case of four DB2 instances existing in the cluster for DB2. Table 6.1 describes the implication of each HADR configuration parameter. Table 6.2, Table 6.3, Table 6.4 and Table 6.5 respectively introduce the detailed HADR configuration parameter settings of the primary, the principal standby, the auxiliary standby1 and the auxiliary standby2. In terms of the primary, "HADR_LOCAL_HOST" is set as its hostname, while "HADR_REMOTE_HOST" is set as the hostname of the principal standby.

The settings of "HADR_LOCAL_SVC" and "HADR_REMOTE_SVC" refer to the corresponding service name in */etc/services*, which is shown in Listing 6.2. "HADR_REMOTE_INST" is set as the instance of the principal standby. The value of "HADR_SYNCMODE" is NEARSYNC, since it is nearly as good as SYNC, with significantly less communication overhead. "HADR_TIMEOUT" is set to 120, which means that if a database does not receive a heartbeat message from its partner database within 120s, the database considers the connection down and closes the TCP connection. "HADR_PEER_WINDOW" is set with 120 as well, it enables failover operations with no data loss if the primary failed within 120s. The combination of the hostnames and service ports of all other instances is set as the value of "HADR_TARGET_LIST". However, in the case of only one principal standby in the cluster, the setting of "HADR_TARGET_LIST" is not required. On the principal standby, the values of "HADR_REMOTE_HOST", "HADR_REMOTE_SVC" and "HADR_REMOTE_INST" correspond to the hostname, service port, and instance name of the primary. "HADR_SYNCMODE" is set to NEARSYNC as well because if the principal standby switches roles with the primary during failover, the synchronization mode for the new primary and principal standby pair will also be NEARSYNC. Compared with the settings on the principal standby, the biggest difference between the auxiliary standby settings is that "HADR_SYNCMODE" is set to SUPERASYNC. The reason is that HADR restricts auxiliary standbys to be in SUPERASYNC mode only, which has the shortest transaction response time of all synchronization modes.

Listing 6.2 HADR Service Ports in */etc/services*

```
db2_hadrp       60006/tcp    /*service port for the primary*/
db2_hadrs       60007/tcp    /*service port for the principal standby*/
db2_hadra       60008/tcp    /*service port for the auxiliary standby1*/
db2_hadrb       60009/tcp    /*service port for the auxiliary standby2*/
```

HADR Configuration Parameter	Implication
HADR_LOCAL_HOST	HADR local hostname
HADR_LOCAL_SVC	HADR local service name
HADR_REMOTE_HOST	HADR remote hostname
HADR_REMOTE_SVC	HADR remote service name
HADR_REMOTE_INST	HADR instance name of remote server
HADR_TIMEOUT	HADR timeout value
HADR_SYNCMODE	HADR log write synchronization mode
HADR_PEER_WINDOW	HADR peer window duration (seconds)
HADR_TARGET_LIST	HADR target list

Table 6.1: Implications of HADR Configuration Parameters

HADR Configuration Parameter	**Value**
HADR_LOCAL_HOST	*primary_hostname*
HADR_LOCAL_SVC	db2_hadrp
HADR_REMOTE_HOST	*principal_standby_hostname*
HADR_REMOTE_SVC	db2_hadrs
HADR_REMOTE_INST	*principal_standby_instance_name*
HADR_TIMEOUT	120
HADR_SYNCMODE	NEARSYNC
HADR_PEER_WINDOW	120
HADR_TARGET_LIST	*principal_standby_hostname*:db2_hadrs |*auxiliary_standby1_hostname*:db2_hadra |*auxiliary_standby2_hostname*:db2_hadrb

Table 6.2: HADR Configuration Parameters for the Primary

HADR Configuration Parameter	**Value**
HADR_LOCAL_HOST	*principal_standby_hostname*
HADR_LOCAL_SVC	db2_hadrs
HADR_REMOTE_HOST	*primary_hostname*
HADR_REMOTE_SVC	db2_hadrp
HADR_REMOTE_INST	*primary_instance_name*
HADR_TIMEOUT	120
HADR_SYNCMODE	NEARSYNC
HADR_PEER_WINDOW	120
HADR_TARGET_LIST	*primary_hostname*:db2_hadrp |*auxiliary_standby1_hostname*:db2_hadra |*auxiliary_standby2_hostname*:db2_hadrb

Table 6.3: HADR Configuration Parameters for the Principal Standby

HADR Configuration Parameter	**Value**
HADR_LOCAL_HOST	*auxiliary_standby1_hostname*
HADR_LOCAL_SVC	db2_hadra
HADR_REMOTE_HOST	*primary_hostname*
HADR_REMOTE_SVC	db2_hadrp
HADR_REMOTE_INST	*primary_instance_name*
HADR_TIMEOUT	120
HADR_SYNCMODE	SUPERASYNC
HADR_PEER_WINDOW	120
HADR_TARGET_LIST	*primary_hostname*:db2_hadrp |*principal_standby_hostname*:db2_hadrs |*auxiliary_standby2_hostname*:db2_hadrb

Table 6.4: HADR Configuration Parameters for the Auxiliary Standby1

HADR Configuration Parameter	**Value**
HADR_LOCAL_HOST	*auxiliary_standby2_hostname*
HADR_LOCAL_SVC	db2_hadrb
HADR_REMOTE_HOST	*primary_hostname*
HADR_REMOTE_SVC	db2_hadrp
HADR_REMOTE_INST	*primary_instance_name*
HADR_TIMEOUT	120
HADR_SYNCMODE	SUPERASYNC
HADR_PEER_WINDOW	120
HADR_TARGET_LIST	*primary_hostname*:db2_hadrp |*principal_standby_hostname*:db2_hadrs |*auxiliary_standby1_hostname*:db2_hadra

Table 6.5: HADR Configuration Parameters for the Auxiliary Standby2

Implementing the Automatic Failover with Governor

Governor component is responsible for automatic failover leading to less downtime of stateful database services. It continuously checks whether the primary is healthy, and performs failover immediately once the primary is dead. *Governor* is a template for creating a custom fit high availability solution using *Kubernetes ECTD Key-Value Cluster* for PostgreSQL [1], and the IBM team modified it to fit DB2. On this basis, we refactored the code of *Governor* to allow it to realize automatic failover for the designed cluster for DB2.

The shell script *setup_db2_instance.sh* also contributes to the generation of a *Governor* configuration file called *db2.yml*. *db2.yml* not only collects configurations about the DB2 database but also contains the settings of *Kubernetes ECTD Key-Value Cluster*.

[1] Governor: https://github.com/compose/governor

Listing 6.3 Example of *db2.yml*

```
timestamp_file: /database/config/db2inst1/timestamp_file
force_takeover_window: 300
loop_wait: 10
env: test
truth_manager: etcd3
etcd3:
 scope: etcd
 ttl: 30
 endpoint: ['10.109.75.82:2379']
 timeout: 20
db2:
 ip: db2-0
 ip_other: db2-1
 db: HADRDB
 authentication:
   username: db2inst1
   password: db2inst1
op_timeout:
 connect: 120
 start: 180
 start_as_standby: 180
 start_as_primary: 180
```

Listing 6.3 presents an example of *db2.yml*, and its settings are explained below:

- ***timestamp_file***: the absolute path of the timestamp file recording the time when the DB2 database is set as primary.
- ***force_takeover_window***: the number of seconds in which a forced takeover will not occur. The *Governor* on the standby compares whether the time since the last connection to its peer is within this duration, and if not, a forced takeover will be performed.
- ***loop_wait***: the number of seconds each loop will sleep, including the loop for running a HA manager, the loop for checking the health of the current primary, the loop for examining the running status of the DB2 database, etc.
- ***env***: the purpose of the *Governor* development. In this work, it is fixed at *test*.
- ***truth_manager***: the version of ETCD interface. In this work, it is fixed at *etcd3*.
- ***etcd3***: configurations of *Kubernetes ECTD Key-Value Cluster*.
  - ***scope***: the relative path used on ETCD's HTTP API. In this work, it is fixed at *etcd*.
  - ***ttl***: the Time to live (TTL) to update the data of the current primary stored in the *ETCD Cluster*. It is considered as the duration before the automatic failover process is initiated.
  - ***endpoint***: the endpoint of the *ETCD Cluster*. Its scheme is *host:port*.
  - ***timeout***: the number of seconds the *Governor* waits to establish a connection to the *ETCD Cluster*.

- ***db2***: configurations of the DB2 database.
  - ***ip***: the IP address of the local host.
  - ***ip_other***: the IP address of the remote host (its peer).
  - ***db***: the name of the DB2 database.
  - ***authentication***: the configuration of the DB2 database authentication.
    - ***username***: the username for accessing the DB2 database specified in the *.db2.db* field.
    - ***password***: the password for accessing the DB2 database specified in the *.db2.db* field.
- ***op_timeout***: configurations of the timeout.
  - ***connect***: the number of seconds to wait for a connection to the DB2 database specified in the *.db2.db* field.
  - ***start***: the number of seconds to wait to start the DB2 database manager.
  - ***start_as_standby***: the number of seconds to wait to start HADR as primary on the DB2 database specified in the *.db2.db* field.
  - ***start_as_primary***: the number of seconds to wait to start HADR as standby on the DB2 database specified in the *.db2.db* field.

The failover logic of the *Governor* is illustrated in Algorithm 6.2. This algorithm is running in a loop to monitor the cluster for DB2 continuously. The algorithm implements HA for the cluster for DB2 employing a HA manager that contains a *state_handler* (maps to a db2 object) to check for HADR roles, and a *truth_manager* (maps to an *ETCD* object) to put or get the hostname of the current primary. From the viewpoint of the *Governor*, the host whose HADR role is primary is considered as the leader. The automated failover of the *Governor* is divided into two phases: i) the principal standby takeover phase, and ii) the phase where the previous primary comes back as the new principal standby. Once the current primary fails, the *Governor* on the principal standby will enter the takeover phase (corresponding to line 2 to line 8 in Algorithm 6.2). As mentioned before, *db2.yml* specifies the TTL of updating the data of the current leader saved in the *ETCD Cluster*. This TTL is a lease on the key-value pair <*leader, value*> as well, indicating that the key-value pair will expire beyond this period. The failure of the primary causes its *Governor* to stop updating this key-value pair. As a result, the *Governor* on the principal standby gets nothing about the current leader from the *ETCD* cluster, implying that there is no leader currently. Thereby, the *Governor* of the principal standby will attempt to update the value of the current leader as the local hostname using the method $truth_manager.acquire_leader()$. Since its HADR role is not primary, the method $state_handler.promote()$ is called to perform the takeover by running the command: "db2 take over hadr on db *db_name* by force". Consequently, the previous principal standby becomes the new primary. When the previous primary recovers from a failure, the *Governor* on it will begin the phase of setting the previous primary as the new principal standby (corresponding to line 24 to line 27 in Algorithm 6.2). The *Governor* of the previous primary finds that a leader already exists, but its current HADR role is primary. A split-brain issue will arise, implying that the cluster for DB2 will most likely have two primaries. To solve the split-brain problem, the previous primary

will demote itself with the method $state_handler.demote()$ executing the command: "db2stop". The *Governor* detects that the previous primary is no longer active after demotion as it is always monitoring the health of the DB2 database. Due to the fact that the timestamp of the current primary storing in the *ETCD Cluster* is more recent than the timestamp of the previous primary when it became the primary, the *Governor* of the previous primary will restart HADR on it as the new principal standby employing command: "db2 start hadr on db *db_name* as standby".

Algorithm 6.2: Failover Algorithm of Governor

```
input  : A HA manager ha = (state_handler, truth_manager), where state_handler is
         a db2 object, and truth_manager is an ETCD object
output : The status of ha, status
1  if db2 state is ok then
2      if no leader exists then
3          if truth_manager.acquire_leader() then
4              if not state_handler.is_primary() then
5                  state_handler.promote();
6                  update leader timestamp in truth_manager to current timestamp;
7                  update primary timestamp in state_handler to current timestamp;
8                  status ← " promoted self to leader ";
9              else
10                 status ← " already be a leader ";
11         else
12             if state_handler.is_primary() then
13                 state_handler.demote();
14                 status ← " demoted self due to potential split brain ";
15     else
16         if I am the leader then
17             if not state_handler.is_primary() then
18                 state_handler.promote();
19                 update leader timestamp in truth_manager to current timestamp;
20                 update primary timestamp in state_handler to current timestamp;
21                 status ← " promoted self to leader ";
22             else
23                 status ← " already be a leader ";
24         else
25             if state_handler.is_primary() then
26                 state_handler.demote();
27                 status ← " demoted self due to potential split brain ";
28 else
29     restart HADR;
30 return status;
```

Building Custom DB2 Image

This section will introduce how to build a DB2 image that contains HADR feature and the *Governor* component with *Dockerfile*. Listing 6.4 shows the detailed *Dockerfile* responsible to build a custom DB2 image. The steps included in the *Dockerfile* are as follows:

1) Choose *ibmcom/db2* as a base image.
2) Copy the directory *db2_setup* which contains shell scripts to set up HADR and *Governor*, and the *Governor* component coded in Python.
3) Copy and run the shell script *install_python2.sh* installing python2 and related packages required in the *Governor* component
4) Copy shell scripts *a_setup_governor.sh* and *b_create_table.sh* to the directory */var/custom*, and add execution permission to them. They will be automatically executed after the DB2 database setup has been completed. *a_setup_governor.sh* runs the *Governor* component in the background with no hangups. *b_create_table.sh* creates a table in the DB2 database. It is utilized to verify the HA feature by inserting data into the primary database and reading the same data from all standby databases.

Listing 6.4 Dockerfile for Building a Custom DB2 Image

```
FROM ibmcom/db2

COPY db2_setup /var/db2_setup
COPY install_python2.sh install_python2.sh
RUN bash install_python2.sh
RUN mkdir /var/custom
COPY a_setup_governor.sh /var/custom
RUN chmod a+x /var/custom/a_setup_governor.sh
COPY b_create_table.sh /var/custom
RUN chmod a+x /var/custom/b_create_table.sh
```

Before building our DB2 image, we deploy a registry server running within the *Docker* container utilizing the below commands:

```
# create the registry directory
sudo mkdir -p /opt/data/registry

# start the registry container
sudo docker run -d --restart=always -p 5000:5000 -v /opt/data/registry:/tmp/registry --name
db2-operator-registry registry
```

The following commands are used to build the custom DB2 image and push it to our private image registry server. They must be executed in the same directory as the above *Dockerfile*.

```
# build image
docker build -t 129.69.209.196:5000/my-db2:latest .

# push image to our registry
docker push 129.69.209.196:5000/my-db2:latest
```

6.2.2 Implementation of the Kubernetes Operator for DB2 with Kubernetes Operator SDK

This section describes how to develop a Go-based *Kubernetes* operator for DB2 with *Kubernetes Operator SDK* in detail. *Kubernetes Operator SDK* makes writing operators easier by providing high-level APIs and abstractions to write the operational logic in a more intuitive way. Moreover, its scaffolding and code generation are helpful to bootstrap a new project quickly. *Kubernetes Operator SDK* provides the following workflow for a new Go-based operator [oprSDK20]:

1) Create a new operator project using the SDK Command Line Interface (CLI)
2) Define new resource APIs by adding CRD
3) Implement the Controller with managed resources, reconciling logic and RBAC
4) Use the SDK CLI to build and generate the operator deployment manifests

The following sections will describe the detailed implementation of each step in the workflow separately.

Creating a Project of the Kubernetes Operator for DB2

First of all, we create a directory for the project of the *Kubernetes* operator for DB2 and initialize it with the following commands:

```
# create a project directory
mkdir db2-operator

# init the project
cd db2-operator
operator-sdk init --domain example.com --repo github.com/example/db2-operator
```

Then we create an API for CR using *DB2Cluster* type and a controller utilizing the command as follows:

```
# create API and controller
operator-sdk create api --group db2 --version v1 --kind DB2Cluster --resource --controller
```

Defining the API for CRD DB2Cluster

After creating an API, *DB2Cluster* resource API is scaffolded at *api/v1/db2cluster_types.go* where we can define this API. The definition of *DB2Cluster* resource is described in Listing 6.5. *Size* specifies the number of *Pods* in a cluster for DB2 with valid values from two to four. *StorageClassforDB* defines the *StorageClass* type to create PVs for mounting databases. The following specifications: *DBName*, *DBInstance*, *DBInstancePassword*, and *StorageClassforHADR* are related to HADR feature. *EtcdEndpoint* is the entry for *ETCD Cluster* used by the *Governor* component. *DBName*, *DBInstance*, and *DBInstancePassword* are related to the DB2 database configuration. *StorageClassforHADR* defines the *StorageClass* type to create PV for mounting the HADR configuration file (*/hadr/hadr.cfg*). According to the above definition of DB2Cluster

resource API, the CR configuration file has to be formatted as shown in Listing 6.6. Once the API is defined with spec/status fields and CRD validation markers, the CRD manifests can be generated and updated by performing the command: "make manifests".

Listing 6.5 Defining DB2Cluster Resource API in *api/v1/db2cluster_types.go*

```
// DB2ClusterSpec defines the desired state of DB2Cluster
type DB2ClusterSpec struct {
        // INSERT ADDITIONAL SPEC FIELDS - desired state of cluster
        // Important: Run "make" to regenerate code after modifying this file

        // Size: the number of Pods in the Cluster for DB2
        // +kubebuilder:validation:Minimum=2
        // +kubebuilder:validation:Maximum=4
        Size int32 `json:"Size"`
        // EtcdEndpoint: the endpoint of etcd cluster
        EtcdEndpoint string `json:"EtcdEndpoint"`
        // DBName: name of db2 database
        // +kubebuilder:default:="HADRDB"
        DBName string `json:"DBName,omitempty"`
        // DBInstance: name of db2 instance
        // +kubebuilder:default:="db2inst1"
        DBInstance string `json:"DBInstance,omitempty"`
        // DBInstancePassword: password of db2 instance
        // +kubebuilder:default:="db2inst1"
        DBInstancePassword string `json:"DBInstancePassword,omitempty"`
        // StorageClassforHADR: StorageClass to mount hadr.cfg file which can be shared
        StorageClassforHADR string `json:"StorageClassforHADR"`
        // StorageClassforDB: StorageClass to mount database
        // +kubebuilder:default:="default"
        StorageClassforDB string `json:"StorageClassforDB,omitempty"`
}
```

Listing 6.6 CR Configuration File Format

```
apiVersion: db2.example.com/v1
kind: DB2Cluster
metadata:
  name:
spec:
  Size:
  EtcdEndpoint:
  DBName:
  DBInstance:
  DBInstancePassword:
  StorageClassforHADR:
  StorageClassforDB:
```

Implement the Controller

The controller of a *Kubernetes* operator for DB2 is developed in the file *controllers/db2cluster_controller.go* by specifying managed resources, implementing reconciling logic and defining RBAC permissions.

1) Specifying Resources Watched by the Controller

Listing 6.7 Specifying Resources Watched by the Controller in *controllers/db2cluster_controller.go*

```
// SetupWithManager sets up the controller with the Manager.
func (r *DB2ClusterReconciler) SetupWithManager(mgr ctrl.Manager) error {

        return ctrl.NewControllerManagedBy(mgr).
                For(&db2v1.DB2Cluster{}).
                Owns(&appsv1.StatefulSet{}).
                Owns(&corev1.Service{}).
                Owns(&corev1.PersistentVolumeClaim{}).
                Complete(r)

}
```

As mentioned before in Section 6.1.1, a cluster for DB2 needs two *Service*s and a *StatefulSet* to manage *Pods* running DB2 database applications. Section 6.2.1 describes that the HADR configuration file (*/hadr/hadr.cfg*) must be mounted in a shared PV for each DB2 instance to access. Therefore, we need to declare that *Service*, *StatefulSet* and PVC are owned and managed by the controller, as shown in Listing 6.7.

2) Implementing Reconciling Logic

The most critical part of implementing the controller is to realize reconcile loop of a reconciler to create and manage a cluster for DB2.
Figure 6.5 presents the process of reconciliation of the *Kubernetes* operator for DB2. If the *db2cluser* object which is an entity of CRD *DB2Cluster* does not exist, the reconciler will remove PVCs used by *Pods* in the *StatefulSet*, since the associated PVCs are not destroyed automatically in order to persistently preserve the data when the *StatefulSet* is deleted. Next, the controller examines whether these *Kubernetes* resources exist in the specified *Namespace*, *Service*s for the primary service and the read-only service, PVC for HADR and *StatefulSet* for managing *Pods* that constituted a cluster for DB2. According to the results, it will create *Kubernetes* resources that do not currently exist. A primary service and a read-only service generated by the reconciler using the *Kubernetes API* k8s.io/api/core/v1 are identical to those created using YAML files in Listing 6.8 and Listing 6.9. The *Service* type is set to *NodePort* to allow both services to be accessible for applications outside of the *Kubernetes* cluster. External applications are able to connect to the primary service and the read-only service via *nodePort* 30001 and 30002 respectively. The configurations of *port* and *targetPort* are the same for both services. The specified port in the *spec* field relates to port 50000

exposed within the *Kubernetes* cluster while the *targetPort* corresponds with port 50000 that is identical to the *containerPort* of each *Pod*. With the selector *role:primary*, the *Pod* owning the label *role:primary* will be targeted by the primary service. Similarly, the read-only service identifies its member *Pods* have the label *app:db2* which is the same as its selector.

Listing 6.8 Primary Service YAML File

```
# Primary Service
apiVersion: v1
kind: Service
metadata:
  name: {db2Cluster.Name}-primary       #db2Cluster.Name is specified in the CR YAML file
  labels:
    app: db2
spec:
  type: NodePort
  ports:
    - name: writeport
      targetPort: 50000
      port: 50000
      nodePort: 30001
  selector:
    role: primary
```

Listing 6.9 Read-only Service YAML File

```
# Read-only Service
apiVersion: v1
kind: Service
metadata:
  name: {db2Cluster.Name}-read-only     #db2Cluster.Name is specified in the CR YAML file
  labels:
    app: db2
spec:
  type: NodePort
  ports:
    - name: readport
      targetPort: 50000
      port: 50000
      nodePort: 30002
  selector:
    app: db2
```

The PVC for creating a PV to mount the HADR configuration file (*/hadr/hadr.cfg*) is generated by the reconciler employing *Kubernetes API* k8s.io/api/core/v1. The equivalent YAML file is presented in Listing 6.10. Since the PV referring to this PVC has to be accessible by many *Pods*, the *StorageClass* with a Network File System (NFS) volume plugin is chosen to provision the PV in this work. Moreover, its *accessModes* is configured as *ReadWriteMany* allowing multiple *Pods* to read/write on the PV bound to it. This PVC requests 1 Gi storage for its corresponding PV.

Listing 6.10 YAML File of PVC for Generating PV to Mount HADR Configuration File

```
# PVC for HADR
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
   name: {db2Cluster.Name}-shared-hadr      #db2Cluster.Name is specified in the CR YAML file
spec:
   accessModes:
     - ReadWriteMany
   #db2Cluster.Spec.StorageClassforHADR is specified in the CR YAML file
   storageClassName: {db2Cluster.Spec.StorageClassforHADR}
   resources:
     requests:
       storage: 1Gi
```

The reconciler utilizes *Kubernetes API* k8s.io/api/apps/v1 to create the *StatefulSet* for the cluster for DB2. The configurations of this *StatefulSet* are illustrated in Listing 6.11. The selector of the *StatefulSet* is set to *app:db2* to ensure that all *Pods* have this label within the specific *NameSpace* belong to it. The *.spec.selector* field of the *StatefulSet* must match the labels of its section *.spec.template.metadata.labels*. Failing to specify a matching selector will result in a validation error during the *StatefulSet* creation. As every *Pod* is labeled with *app:db2* in the section *.spec.template.metadata.labels*, all *Pods* in the *StatefulSet* are related to the read-only service. The *.spec.replicas* field defines the number of *Pods* in the cluster for DB2 depending on the specification from the CR YAML file. The *StatefulSet* will use the settings under the *.spec.template* domain as a template to create each *Pod* it manages. Each *Pod* of this *StatefulSet* launches one container to run the *my-db2* image at the latest version hosted by our private registry. This image is pulled only if it does not exist locally owing to *IfNotPresent* setting of the *.spec.template.spec.containers.imagePullPolicy* section. In the *.spec.template.spec.containers.ports* section, port 50000 is selected as the DB2 database manager is listening on it for connections. The following section *.spec.template.spec.containers.env* describes necessary environment variables concerned with the settings of HADR and the *Governor* component. The explanation of these environment variables is described in Table 6.6. The PV called *db2database* is mounted to the path */database* within the container storing the data of the database persistently. The shared PV with the name *hadr-data* mounted at */hadr* to provide the storage for the HADR configuration file (*/hadr/hadr.cfg*). In order to read the log of the *Governor* locally, a *hostPath* volume mounts the directory */home/xiaomin/log* from the host node's filesystem into the container at */var/log/governor*. The field *.spec.template.spec.containers.securityContext* defines constraints applied to the container. Its setting *privileged: true* indicates the container is run as privileged.

Listing 6.11 YAML File of StatefulSet Deployed in the Cluster for DB2

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: {db2Cluster.Name}       #db2Cluster.Name is specified in the CR YAML file
  labels:
    app: db2
spec:
  selector:
    matchLabels:
      app: db2
  replicas: {db2Cluster.Spec.Size}      #db2Cluster.Spec.Size is specified in the CR YAML file
  template:
    metadata:
      name: db2
      labels:
        app: db2
    spec:
      containers:
        - name: db2
          image: 129.69.209.196:5000/my-db2:latest
          imagePullPolicy: "IfNotPresent"
          ports:
          - containerPort: 50000
          env:
          - name: BLU
            value: "false"
          - name: ENABLE_ORACLE_COMPATIBILITY
            value: "false"
          - name: UPDATEAVAIL
            value: "NO"
          - name: REPODB
            value: "false"
          - name: IS_OSXFS
            value: "false"
          - name: PERSISTENT_HOME
            value: "true"
          - name: DBNAME
            #db2Cluster.Spec.DBName is specified in the CR YAML file
            value: {db2Cluster.Spec.DBName}
          - name: DB2INSTANCE
            #db2Cluster.Spec.DBInstance is specified in the CR YAML file
            value: {db2Cluster.Spec.DBInstance}
          - name: DB2INST1_PASSWORD
            #db2Cluster.Spec.DBInstancePassword is specified in the CR YAML file
            value: {db2Cluster.Spec.DBInstancePassword}
          - name: LICENSE
            value: "accept"
          - name: TO_CREATE_SAMPLEDB
            value: "false"
```

```
        - name: HADR_ENABLED
          value: "true"
        - name: ETCD_ENDPOINT
          #db2Cluster.Spec.EtcdEndpoint is specified in the CR YAML file
          value: {db2Cluster.Spec.EtcdEndpoint}
        - name: REPLICAS
          #db2Cluster.Spec.Size is specified in the CR YAML file
          value: {db2Cluster.Spec.Size}
        volumeMounts:
        - name: db2database
          mountPath: /database
        - name: hadr-data
          mountPath: /hadr
        - name: log-path
          mountPath: /var/log/governor
        securityContext:
          privileged: true
    volumes:
      - name: hadr-data
        persistentVolumeClaim:
          #db2Cluster.Name is specified in the CR YAML file
          claimName: {db2Cluster.Name}-shared-hadr
      - name: log-path
        hostPath:
          path: /home/xiaomin/log
volumeClaimTemplates:
  - metadata:
      name: db2database
    spec:
      #db2Cluster.Spec.StorageClassforDB is specified in the CR YAML file
      storageClassName: {db2Cluster.Spec.StorageClassforDB}
      accessModes: ["ReadWriteOnce"]
      resources:
        requests:
          storage: 1Gi
```

Environment Variables	Value	Implication
BLU	false	sets BLU Acceleration to disabled
ENABLE_ORACLE_COMPATIBILITY	false	sets Oracle Compatibility to disabled
UPDATEAVAIL	NO	there is no existing instance running a new container with a higher Db2 level
REPODB	false	does not create a Data Server Manager repository database
IS_OSXFS	false	the operating system is not macOS
PERSISTENT_HOME	true	the default setting is "true" and should only be specified as "false" when running it on Windows
DBNAME	db2Cluster.Spec.DBName	sets the name of the DB2 database
DB2INSTANCE	db2Cluster.Spec.DBInstance	sets the name of the DB2 insatnce
DB2INST1_PASSWORD	db2Cluster.Spec.DBInstancePassword	sets the password of the DB2 insatnce
LICENSE	accept	accepts the terms and conditions of the Db2 software contained
TO_CREATE_SAMPLEDB	false	does not create a sample database
HADR_ENABLED	true	enables HADR feature
ETCD_ENDPOINT	db2Cluster.Spec.EtcdEndpoint	specifies the endpoint of the *ETCD Key-Value Cluster*
REPLICAS	db2Cluster.Spec.Size	defines the number of *Pods* in the cluster for DB2

Table 6.6: Environment Variables of the Container Launched in *StatefulSet's Pods*

Once the specification of the cluster for DB2 is changed, the reconciler will update the corresponding *Kubernetes* resources to achieve the desired state. Finally, the reconciler will update the labels of *Pods* to *role:primary* or *role:standby* based on their HADR roles. As a result, the *Pod* with the label *role:primary* is always mapped to the primary service. This reconciling process is running continuously to enforce the desired CR state on the current state of the cluster for DB2.

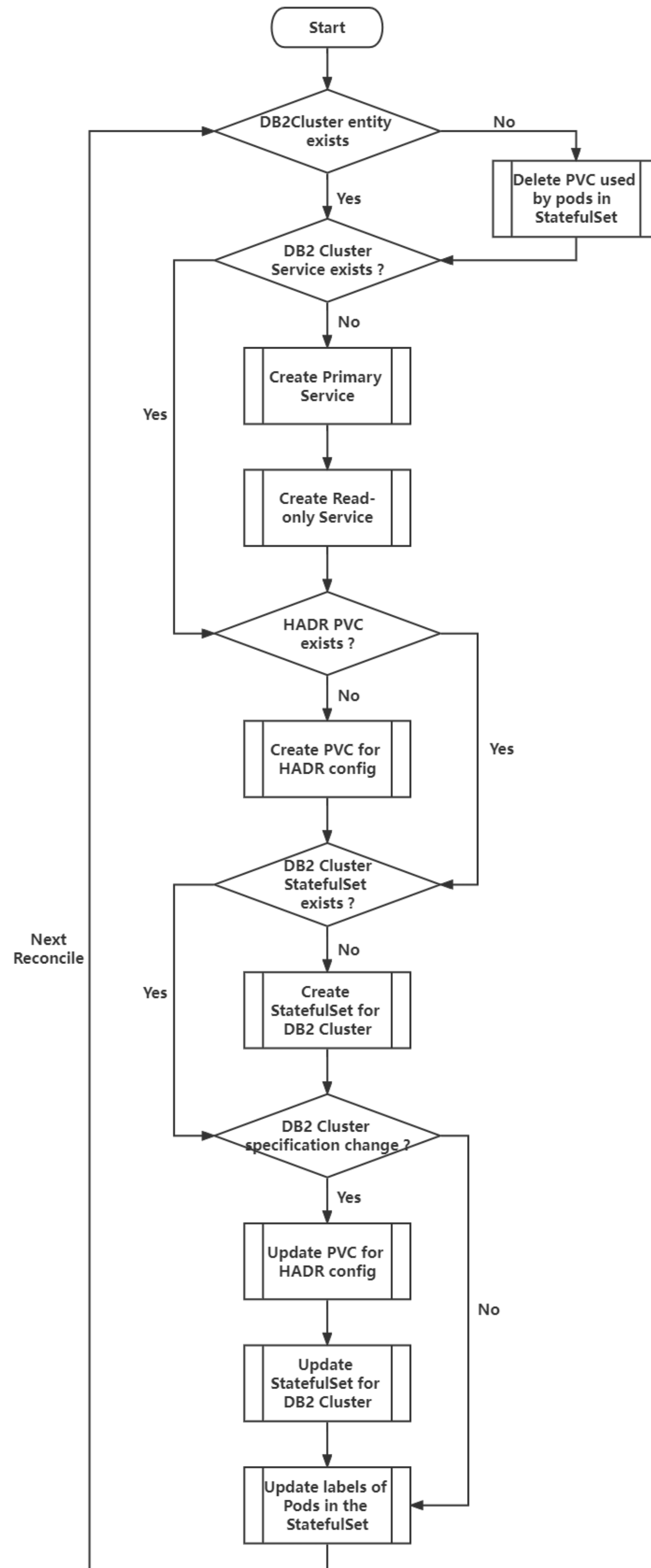


Figure 6.5: Reconciling Logic of the *Kubernetes* Operator for DB2

Algorithm 6.3: Failover Management of Reconciler in the *Kubernetes* Operator for DB2

input : A list of *Pods* that are in the *StatefulSet podList*
output : Labels of *Pods* are updated with HADR roles, and auxiliary standbys connect to the new primary

```
1  for pod in podList do
2      hadrRole ← getHadrRolefromPod();
3      hadrState ← getHadrStatefromPod();
4      hadrConnectStatus ← getHadrConnectStatusfromPod();
5      if hadrConnectState = CONNECTED then
6          if hadrRole = PRIMARY then
7              set pod's label as "role:primary";
8              currentPrimary ← pod.Name;
9          else if hadrRole = STANDBY and hadrState = PEER then
10             set pod's label as "role:standby";
11             currentStandby ← pod.Name;
12         else if hadrRole = STANDBY and hadrState = REMOTE_CATCHUP then
13             set pod's label as "role:standby";
14     if (hadrState = DISCONNECTED_PEER or hadrState = DISCONNECTED) and
        hadrConnectState = DISCONNECTED then
15         if hadrRole = PRIMARY then
16             if CurrentPrimary ! = CurrentStandby and
                pod.Name! = CurrentPrimary then
17                 set pod's label as "role:primary";
18                 currentPrimary ← pod.Name;
19                 auxiliary standbys connect to currentPrimary;
20             else if hadrRole = STANDBY then
21                 if CurrentPrimary = CurrentStandby then
22                     set pod's label as "role:standby";
23                     currentStandby ← pod.Name;
```

Standby	HADR Parameter	Primary	
		Connect	Disconnect
Principal Standby	HADR_STATE	PEER	DISCONNECTED DISCONNECTED_PEER
	HADR_CONNECT_STATUS	CONNECTED	DISCONNECTED
Auxiliary Standby	HADR_STATE	REMOTE_CATCHUP	REMOTE_CATCHUP_PENDING
	HADR_CONNECT_STATUS	CONNECTED	DISCONNECTED

Table 6.7: HADR State and HADR Connect Status of DB2 Databases

After failover, labels of *Pods* are required to change as the HADR role changes. In addition, the *Kubernetes* operator for DB2 needs to update HADR configuration parameters "HADR_REMOTE_HOST" and "HADR_REMOTE_SVC" of auxiliary standbys with the hostname and service port of the new primary allowing them to connect to the new primary, as the *Governor* component just manages failover between the primary and the principal standby. The reconciler addresses both of these requirements utilizing an algorithm, the pseudocode for which is given in Algorithm 6.3. For each *Pod* of *StatefulSet*, the reconciler gets HADR configuration by means of executing the command: "db2pd -db *database_name* -hadr " in the *Pod* and extracts the HADR role, the HADR state and the HADR connect status from it. The reconciler determines HADR connectivity between the primary and standbys based on the values of the HADR state and the HADR connect status. Table 6.7 Figure 1 illustrates the HADR state and the HADR connect status of the primary and each standby with and without connection. In the case of connected, the reconciler will set a role label for *Pod* according to its HADR role and record the *Pod's* name of the current primary and the principal standby respectively. When the failover has been completed, but the HADR connection is lost because the previous primary has not been recreated as the new principal standby. When the previous principal standby becomes the new primary, there will be an inconsistency between its label and HADR role. At this point, the reconciler will update its label allowing auxiliary standbys to connect to it. Once the previous primary becomes the new principal standby, its label will be updated accordingly as well.

3) Specifying RBAC Permissions

As mentioned in Section 6.1.3, the *Kubernetes* operator for DB2 needs a *ServiceAccount* bounding to a *ClusterRole* that specifies the permissions to access *Kubernetes* resources. Thanks to *Kubernetes Operator SDK*, permissions can be specified via RBAC markers above the reconcile function like the following:

```
//+kubebuilder:rbac:groups=core,resources=persistentvolumeclaims,verbs=get;list;watch;create;
update;patch;delete
//+kubebuilder:rbac:groups=core,resources=services,verbs=get;list;watch;create;update;patch;
delete
//+kubebuilder:rbac:groups=apps,resources=statefulsets,verbs=get;list;watch;create;update;
patch;delete
//+kubebuilder:rbac:groups=core,resources=pods,verbs=get;list;watch;create;update;patch;delete
//+kubebuilder:rbac:groups=core,resources=pods/exec,verbs=get;list;watch;create;update;patch;
delete
```

In the above specification, the *Kubernetes* operator for DB2 is granted all permissions to access *Kubernetes* resources: *Service*, *StatefulSet*, *PVC*, *Pod* and *Pod/exec*. The *ClusterRole* manifest is generated from these markers via controller-gen after executing the command: "make manifests".

Deploying the Kubernetes Operator for DB2

Benefiting from *Kubernetes Operator SDK*, building an image of the *Kubernetes* operator for DB2 and deploying it to the *Kubernetes* cluster is reasonably simple as only the following commands are required:

```
# build an image of the Kubernetes operator for DB2 and push it to the private registry
make docker-build docker-push IMG="129.69.209.196:5000/db2-operator:latest"

# deploy the Kubernetes operator for DB2 to the Kubernetes cluster
make deploy IMG="129.69.209.196:5000/db2-operator:latest"
```

Kubernetes Operator SDK uses GoogleContainerTools' *distroless* as the base image to build the image of the *Kubernetes* operator, as it only contains the custom application and its runtime dependencies without shells or any other programs in a standard Linux distribution. The image of the *Kubernetes* operator for DB2 is uploaded to our private registry server. After executing the above commands, a *Kubernetes* operator for DB2 is up and running in the namespace *db2-operator-system*, as well as related *ServiceAccount*, *ClusterRole*, *RoleBinding*, and CRD are deployed.

6.3 Test and Evaluation

In this section, we first introduce the infrastructure and settings of the test environment. Moreover, we design various test scenarios to evaluate the prototype mentioned in Section 6.1.1 regarding HA, DR and read scalability.

6.3.1 Infrastructure

The prototype was tested on the infrastructure of the Institute of Parallel and Distributed Systems at the University of Stuttgart. It consists of an Open Stack instance that manages the virtual machines running Ubuntu 20.04.5 that are used in this work. We constructed a *Kubernetes* cluster composed of three VMs, one of which serves as the master node, while the other two serve as worker nodes. *Kubernetes* 1.24.4 runs on all VMs and the container engine is *Docker* 20.10.21.

6.3.2 Test Setup

Before conducting different test scenarios, we first deployed a cluster for DB2 within the above *Kubernetes* cluster. The deployment process is summarized in the following steps:

1) Deploy an *ETCD* cluster providing an *ETCD* service.
2) Deploy a *Kubernetes* operator for DB2 using the commands mentioned in Section 6.2.2.
3) Create a cluster for DB2 based on the configuration file shown in Listing 6.13.
4) Set up an HAProxy running on the *Docker* container to connect to DB2 database services according to the configuration file presented in Listing 6.12.
5) Develop a client which connects to the HAProxy and performs CRUD operations on the DB2 database for testing.

Listing 6.12 HAProxy Configuration File

```
global
    log         stdout format raw local0 info
defaults
    mode            tcp
    log             global
    option          tcplog
    option          dontlognull
    option          http-server-close
    option          redispatch
    retries         3
    timeout         http-request 10s
    timeout         queue   1m
    timeout         connect 10s
    timeout         client  1m
    timeout         server  1m
    timeout         http-keep-alive 10s
    timeout         check   10s
    maxconn         3000

frontend        db2-primary
    bind        0.0.0.0:23307
    mode        tcp
    log         global
    default_backend db2_cluster_primary

backend     db2_cluster_primary
    log         global
    balance roundrobin
    server  node01 129.69.209.194:30001 check inter 5s rise 2 fall 3
    server  node02 129.69.209.195:30001 check inter 5s rise 2 fall 3

frontend        db2-read-only
    bind        0.0.0.0:23308
    mode        tcp
    log         global
    default_backend db2_cluster_read_only

backend     db2_cluster_read_only
    log         global
    balance roundrobin
    server  node01 129.69.209.194:30002 check inter 5s rise 2 fall 3
    server  node02 129.69.209.195:30002 check inter 5s rise 2 fall 3
listen stats
    mode    http
    bind    0.0.0.0:1080
    stats   enable
    stats   hide-version
    stats uri /haproxyamdin?stats
    stats realm Haproxy\ Statistics
    stats auth admin:admin
    stats admin if TRUE
```

As described in Listing 6.12, the HAProxy configures a frontend named *db2-primary* which handles all incoming read/write requests on port 23307. This frontend will forward all received requests to the backend named *db2_cluster_primary* as specified in the section *default_backend*. Considering the *Service* type of the primary service is *Nodeport*, the backend *db2_cluster_primary* corresponds to it via IP addresses of VM nodes and port 30001. Similarly, a frontend with port 23308 called *db2-read-only* is set up to forward read requests to the backend *db2_cluster_read_only* which is related to the read-only service of the cluster for DB2 through port 30002. Both backends employ a round-robin load balancing strategy, indicating that the traffic is forwarded to the corresponding server in turn. The last section configures HAProxy stats which provides detailed statics on HAProxy deployment including data transmission, the number of connections as well as server status. The real-time information of HAProxy implementation is accessible via *<hostip:1080>* from a web browser using the name and the password given in the section *stats auth*.

Listing 6.13 CR Configuration File

```
apiVersion: db2.example.com/v1
kind: DB2Cluster
metadata:
  name: db2
spec:
  Size: 4
  EtcdEndpoint: "10.109.75.82:2379"
  DBName: "HADRDB"
  DBInstance: "db2inst1"
  DBInstancePassword: "db2inst1"
  StorageClassforHADR: "managed-nfs-storage"
  StorageClassforDB: "openebs-hostpath"
```

A cluster for DB2 is created and the external client can access the database service after performing the above steps. The cluster for DB2 contains four *Pods* with names ranging from *db2-0* to *db2-3*. DB2 database applications are deployed inside containers of *Pods* that are governed by *StatefulSet*. Each DB2 database application has a db2 instance named *db2inst1* and a database named *HADRDB*.

An example of how a client can connect to the primary database via the primary service is shown in Figure 6.6. Once a *Service* object with the name *primary-service* and a selector *role:primary* is created, the corresponding *Endpoints* will also be created containing the IP addresses and ports of all *Pods* with the label *role:primary*. The IP addresses and ports of *Pods* in *Endpoints* are updated dynamically based on the specification changes of the *primary-service* selector. The type of *primary-service* is *NodePort*, thus it will open port 30001 on each VM node within the *Kubernetes* cluster. When the client sends requests to HAProxy, HAProxy redirects requests to the VM node which has an available primary database service, and the VM node will forward them to *primary-service* accordingly. *primary-service* will select an IP address of *Pod* with port 50000 from *Endpoints* in a random mode and forward requests from the client to it. The *Service* in *Kubernetes* is an abstraction that specifies a logical collection of *Pods*. The forwarding function of *Service* is actually implemented by *kube-proxy* running on each node of a *Kubernetes* cluster. Under

the *iptables* mode, *kube-proxy* watches *Service* and *Endpoints* objects, and accordingly updates *iptables* on its host nodes to allow forwarding traffic to the actual IP address of the *Pod*. Finally, a connection between the client and the primary db2 database is established.

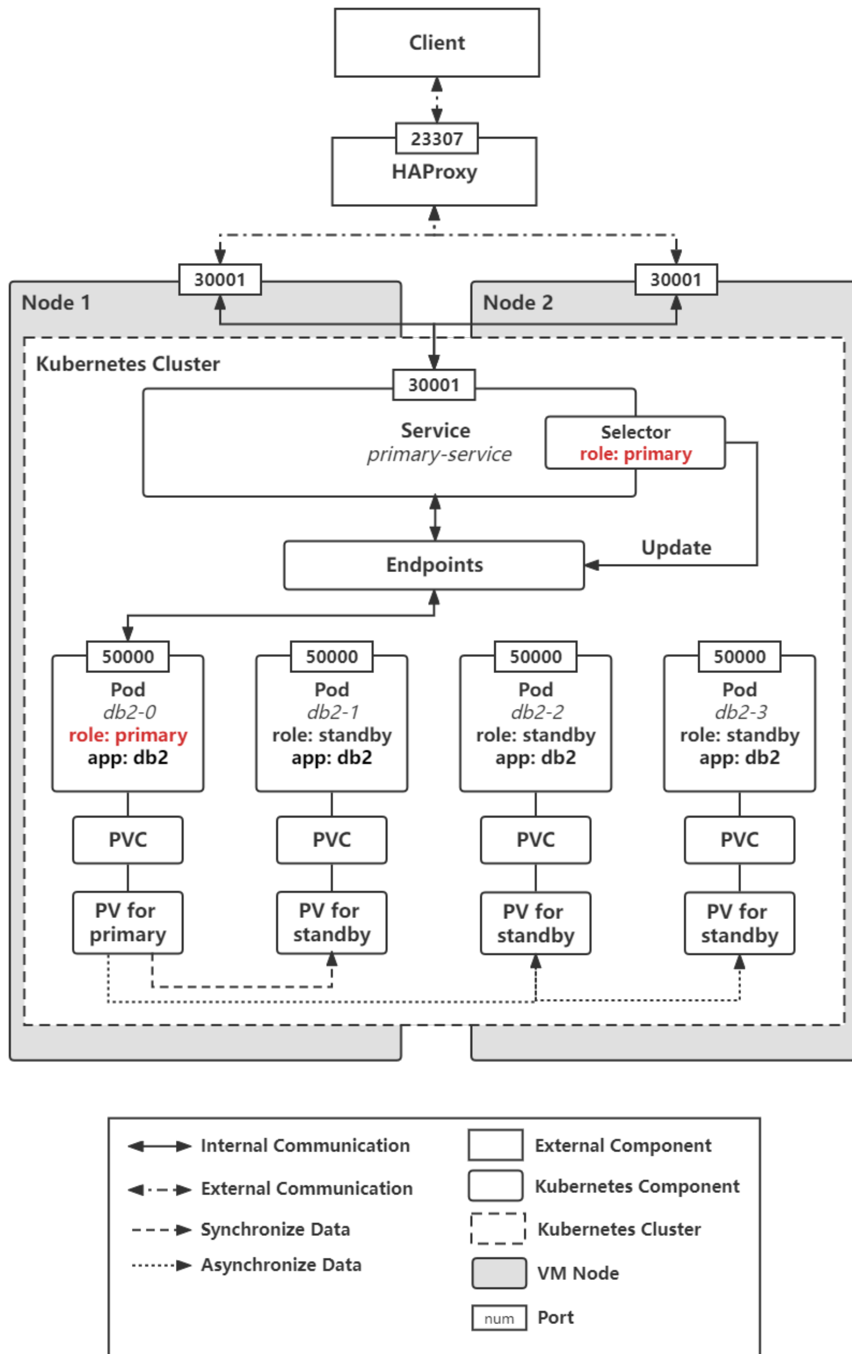


Figure 6.6: Example of a Client Accessing the Primary DB2 Database Service

The following Figure 6.7 illustrates an example of a client connecting to the read-only DB2 database service. The process for a client to access the read-only service is the same as that for the primary service. However, the read-only service called *read-only-service* employs a selector *app:db2* to target all *Pods* owning the label *app:db2*. Because the primary database and all standby databases can handle read queries, each *Pod* in the *StatefulSet* is given the general label *app:db2*.

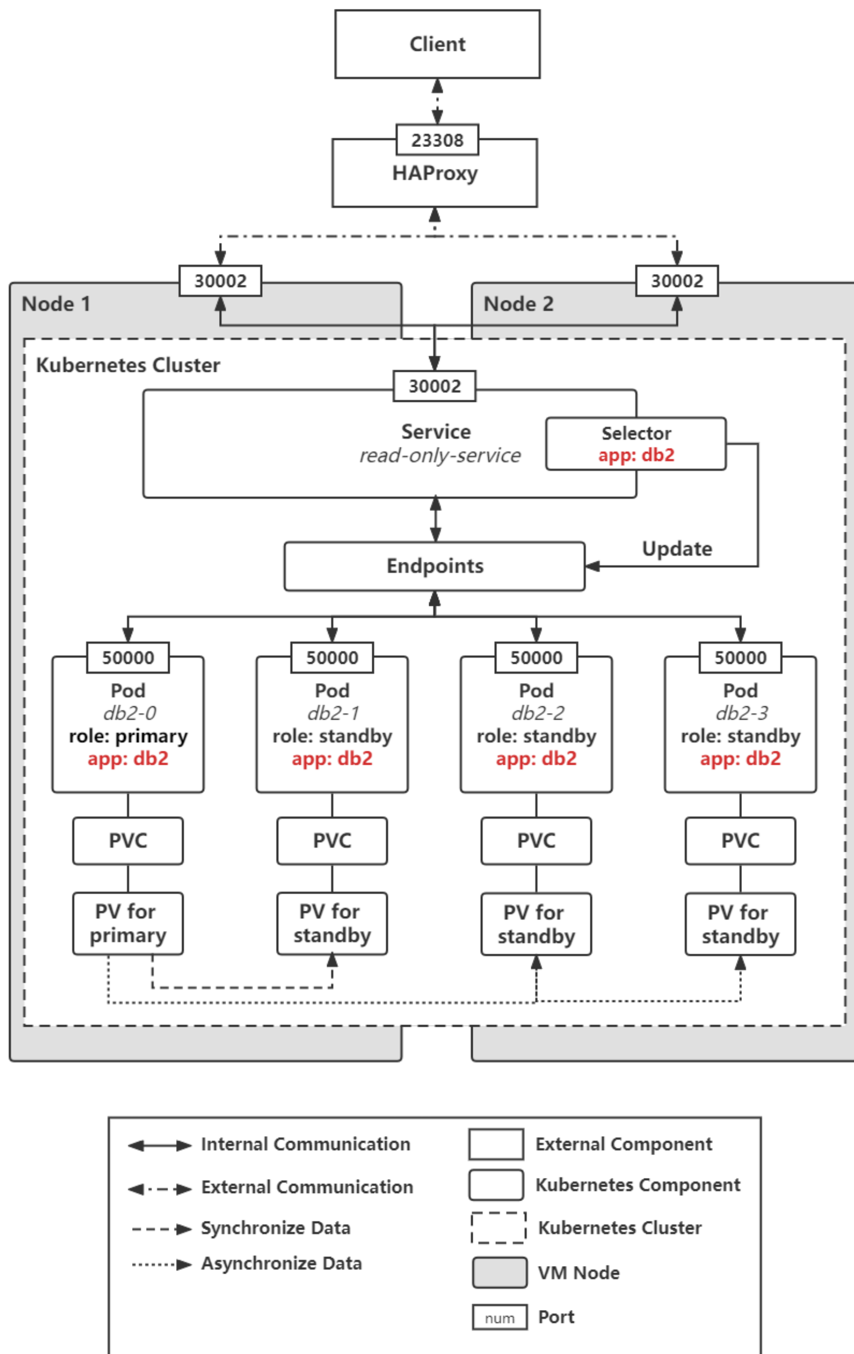


Figure 6.7: Example of a Client Accessing the Read-only DB2 Database Service

6.3.3 Test Scenario of High Availability

In this section, we describe a test scenario to evaluate the prototype in terms of the high availability of the primary service. In our scenario, the primary service outage is due to the failure of the *Pod* in which the primary database application is running. We simulated a failure event by deleting *Pod* running the primary DB2 database, and observed whether the following reactions occurred as expected:

1) *StatefulSet* recreates the deleted *Pod* with the same identity
2) The principal standby takes over as the new primary
3) The previous primary becomes the new principal standby
4) Auxiliary standbys connect to the new primary

HADR_ROLE	STANDBY_ID	HADR_STATE	PRIMARY_MEMBER_HOST	STANDBY_MEMBER_HOST
PRIMARY	1	PEER	db2-0	db2-1
PRIMARY	2	REMOTE_CATCHUP	db2-0	db2-2
PRIMARY	3	REMOTE_CATCHUP	db2-0	db2-3

(a) HADR Details of the Primary (*db2-0*)

HADR_ROLE	STANDBY_ID	HADR_STATE	PRIMARY_MEMBER_HOST	STANDBY_MEMBER_HOST
STANDBY	0	PEER	db2-0	db2-1

(b) HADR Details of the Principal Standby (*db2-1*)

HADR_ROLE	STANDBY_ID	HADR_STATE	PRIMARY_MEMBER_HOST	STANDBY_MEMBER_HOST
STANDBY	0	REMOTE_CATCHUP	db2-0	db2-2

(c) HADR Details of the Auxiliary Standby1 (*db2-2*)

HADR_ROLE	STANDBY_ID	HADR_STATE	PRIMARY_MEMBER_HOST	STANDBY_MEMBER_HOST
STANDBY	0	REMOTE_CATCHUP	db2-0	db2-3

(d) HADR Details of the Auxiliary Standby2 (*db2-3*)

Figure 6.8: HADR Details of Each *Pod* Before Failover

```
NAME    READY   STATUS              RESTARTS   AGE     ROLE
db2-0   1/1     Running             0          6m6s    primary
db2-1   1/1     Running             0          5m58s   standby
db2-2   1/1     Running             0          5m52s   standby
db2-3   1/1     Running             0          5m46s   standby
db2-0   1/1     Terminating         0          6m12s   primary
db2-0   1/1     Terminating         0          6m19s   primary
db2-0   0/1     Terminating         0          6m20s   primary
db2-0   0/1     Terminating         0          6m20s   primary
db2-0   0/1     Terminating         0          6m20s   primary
db2-0   0/1     Pending             0          0s
db2-0   0/1     Pending             0          0s
db2-0   0/1     ContainerCreating   0          0s
db2-0   0/1     ContainerCreating   0          1s
db2-0   1/1     Running             0          2s
db2-1   1/1     Running             0          6m24s   primary
db2-0   1/1     Running             0          4m4s    standby
```

Figure 6.9: Process of Label Changes for *Pods* in Failover

```
HADR_ROLE     STANDBY_ID HADR_STATE              PRIMARY_MEMBER_HOST  STANDBY_MEMBER_HOST
------------- ---------- ----------------------- -------------------- --------------------
STANDBY                0 PEER                    db2-1                db2-0
```

(a) HADR Details of the Primary (*db2-0*)

```
HADR_ROLE     STANDBY_ID HADR_STATE              PRIMARY_MEMBER_HOST  STANDBY_MEMBER_HOST
------------- ---------- ----------------------- -------------------- --------------------
PRIMARY                1 PEER                    db2-1                db2-0
PRIMARY                2 REMOTE_CATCHUP          db2-1                db2-2
PRIMARY                3 REMOTE_CATCHUP          db2-1                db2-3
```

(b) HADR Details of the Principal Standby (*db2-1*)

```
HADR_ROLE     STANDBY_ID HADR_STATE              PRIMARY_MEMBER_HOST  STANDBY_MEMBER_HOST
------------- ---------- ----------------------- -------------------- --------------------
STANDBY                0 REMOTE_CATCHUP          db2-1                db2-2
```

(c) HADR Details of the Auxiliary Standby1 (*db2-2*)

```
HADR_ROLE     STANDBY_ID HADR_STATE              PRIMARY_MEMBER_HOST  STANDBY_MEMBER_HOST
------------- ---------- ----------------------- -------------------- --------------------
STANDBY                0 REMOTE_CATCHUP          db2-1                db2-3
```

(d) HADR Details of the Auxiliary Standby2 (*db2-3*)

Figure 6.10: HADR Details of Each *Pod* After Failover

Before deleting the *Pod* with the primary database application, we checked the HADR role, state and connection status of the DB2 database in each *Pod*. According to Figure 6.8 showing HADR details, we concluded that *db2-0* was the primary, *db2-1* was the principal standby, as well as *db2-2* and *db2-3* were auxiliary standbys before failover. Next, we deleted the *Pod* and monitored the process of failover using the commands: "kubectl delete pod db2-0" and "kubectl get pods -L role -w" separately. Figure 6.9 presents the process of the primary being destroyed, recreated, and becoming a new principal standby, as well as the principal standby becoming the new primary once the previous primary fails. In Figure 6.9, ① indicates the primary and the principal standby before failover, while ② indicates the situation after failover. Finally, we examined the HADR role, state and connection status again to confirm that the failover was successful, which is illustrated

in Figure 6.10. After failover, *db2-1* (the previous principal standby) became the new primary, and *db2-0* (the previous primary) became the new principal standby. *db2-2* and *db2-3* were still auxiliary standbys but connected to the new primary *db2-1*.

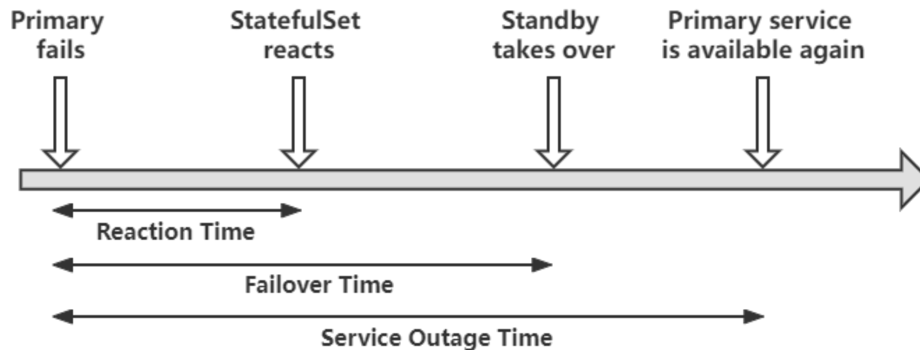


Figure 6.11: Availability Metrics

The availability metrics used to evaluate our proposed prototype from the HA perspective are defined as follows, as well as in Figure 6.11:

- **Reaction Time**
  The time between the failure event of the primary and the reaction of *StatefulSet* that recreates the failed *Pod*.
- **Failover Time**
  The time between the failure event of the primary and the previous principal standby taking over as the new primary.
- **Service Outage Time**
  The duration in which the primary database service was not available.

We repeated this scenario 10 times and all measurements are reported in Table 6.8 in unit seconds. As observed in Table 6.8, the reaction times of all experiments are relatively close and measured between 1.720 to 2.859 seconds. However, the failover times are unstable and vary widely, the minimum is 3.621 seconds while the maximum is 36.309 seconds. This is due to the *Governor* component checking whether the situation satisfies the failover condition (i.e. the primary fails and no data of the current primary exists in the *ETCD* cluster) every 20 seconds, and performing failover based on this condition. The duration of the failover period is determined by the location of the primary failure event within the 20-second interval. If the primary failure happens immediately following a condition check, it will not be identified until the next condition check occurs. Conversely, the condition check occurs directly after the primary failure event, then the primary failure event will be detected right away. Equally, the *Kubernetes* operator for DB2 examines for HADR role changes on a regular basis. Thus, the location of the failover event during the interval determines the duration of a service outage. On the other hand, the service outage time also depends on the failover time, as shown in Figure 6.11.

In summary, the cluster for DB2 provides a stateful primary database service featuring HA achieved by automated failover. This primary service is able to recover within 19.642 seconds on average in the event of a primary failure.

(Unit: seconds)	**Reaction Time**	**Failover Time**	**Service Outage Time**
1	2.859	3.621	6.425
2	1.995	4.322	6.839
3	2.403	9.655	18.180
4	2.066	5.856	13.793
5	2.022	36.309	41.639
6	2.051	9.632	14.555
7	1.720	4.839	9.570
8	2.058	29.886	32.728
9	2.624	9.679	18.111
10	2.059	33.286	34.581
Average	**2.186**	**14.709**	**19.642**

Table 6.8: Results of 10 Experiments Measuring Availability

6.3.4 Test Scenario of Disaster Recovery

This section introduces the test scenario examining our disaster recovery plan with aspects of RPO and RTO. The definitions and the differences between these two metrics are described below and illustrated in Figure 6.12 as well:

- **RPO**
  RPO refers to the interval between the last data backup and a disaster. It focuses on how far back in time the disaster occurred. In other words, RPO measures the amount of data lost in a disaster.
- **RTO**
  RTO is the period within which the service is restored after a disaster. It emphasizes how long the service can be recovered following a disaster i.e. the downtime of the service.

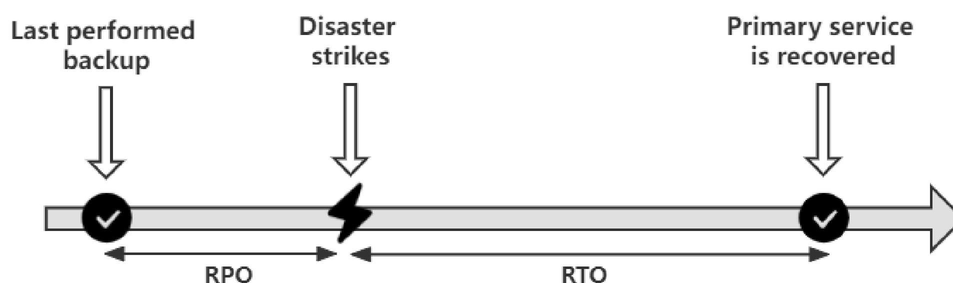


Figure 6.12: Definition and Differences Between RPO and RTO

First, we inspected the current state of the cluster for DB2 before a disaster, which is shown in Figure 6.13. Next, we simulated a disaster by shutting down HADR and the *Governor* component on *db2-0* (the primary) and *db2-1* (the principal standby), as they cannot be recovered by both the *StatefulSet Controller* and the *Kubernetes* operator for DB2. Next, we removed the role labels from them using the command: "kubectl label –overwrite pods *pod_name* role=". At this point, the primary service was no longer available. Following this, we conducted the disaster recovery plan in the following steps:

1) Set up HADR on *db2-2* as the primary using the commands list in Listing 6.14
2) Set up HADR on *db2-3* as the principal standby using the commands list in Listing 6.15
3) Start *Governor* on both *db2-2* and *db2-3*
4) Check the primary service is available again

After performing the above disaster recovery plan, we examined the current state of the cluster for DB2 again, which is presented in Figure 6.14. Moreover, we extracted the time of the last performed backup from the value of the HADR parameter *STANDBY_REPLAY_LOG_TIME*. We computed the RPO and RTO based on it, as well as the recorded disaster occurrence time and service recovery time. The RPO and RTO of our cluster for DB2 are 17 minutes and 6 minutes respectively.

In conclusion, the designed cluster for DB2 can provide the DR feature with better RPO and RTO. Nonetheless, the service is limited to being restored by operating a disaster recovery plan manually.

```
HADR_ROLE     STANDBY_ID HADR_STATE              PRIMARY_MEMBER_HOST  STANDBY_MEMBER_HOST
------------- ---------- ----------------------- -------------------- --------------------
PRIMARY                1 PEER                    db2-0                db2-1
PRIMARY                2 REMOTE_CATCHUP          db2-0                db2-2
PRIMARY                3 REMOTE_CATCHUP          db2-0                db2-3
```

(a) HADR Details of the Primary (*db2-0*)

```
HADR_ROLE     STANDBY_ID HADR_STATE              PRIMARY_MEMBER_HOST  STANDBY_MEMBER_HOST
------------- ---------- ----------------------- -------------------- --------------------
STANDBY                0 PEER                    db2-0                db2-1
```

(b) HADR Details of the Principal Standby (*db2-1*)

```
HADR_ROLE     STANDBY_ID HADR_STATE              PRIMARY_MEMBER_HOST  STANDBY_MEMBER_HOST
------------- ---------- ----------------------- -------------------- --------------------
STANDBY                0 REMOTE_CATCHUP          db2-0                db2-2
```

(c) HADR Details of the Auxiliary Standby1 (*db2-2*)

```
HADR_ROLE     STANDBY_ID HADR_STATE              PRIMARY_MEMBER_HOST  STANDBY_MEMBER_HOST
------------- ---------- ----------------------- -------------------- --------------------
STANDBY                0 REMOTE_CATCHUP          db2-0                db2-3
```

(d) HADR Details of the Auxiliary Standby2 (*db2-3*)

Figure 6.13: HADR Details of Each *Pod* Before a Disaster

HADR_ROLE	STANDBY_ID	HADR_STATE	PRIMARY_MEMBER_HOST	STANDBY_MEMBER_HOST
PRIMARY	1	PEER	db2-2	db2-3

(a) HADR Details of the New Primary (*db2-2*)

HADR_ROLE	STANDBY_ID	HADR_STATE	PRIMARY_MEMBER_HOST	STANDBY_MEMBER_HOST
STANDBY	0	PEER	db2-2	db2-3

(b) HADR Details of the New Principal Standby (*db2-3*)

NAME	READY	STATUS	RESTARTS	AGE	ROLE
db2-0	1/1	Running	0	56m	
db2-1	1/1	Running	0	56m	
db2-2	1/1	Running	0	56m	primary
db2-3	1/1	Running	0	56m	standby

(c) HADR Role of Each *Pod* in Current Cluster for DB2

Figure 6.14: HADR Details of Each *Pod* After a Disaster

Listing 6.14 Commands for Setting up HADR on *db2-2* as Primary

```
# take over as the new primary
db2 takeover hadr on db HADRDB by force
# HADR configurations
db2 "update db cfg for HADRDB using HADR_TARGET_LIST NULL"
db2 "update db cfg for HADRDB using HADR_SYNCMODE NEARSYNC"
db2 "update db cfg for HADRDB using HADR_REMOTE_HOST db2-3"
db2 "update db cfg for HADRDB using HADR_REMOTE_SVC db2_hadrb"
# restart HADR as the primary
db2 deactivate db HADRDB
db2 stop hadr on db HADRDB
db2 start hadr on db HADRDB as primary by force
db2 activate db HADRDB
```

Listing 6.15 Commands for Setting up HADR on *db2-3* as Principal Standby

```
# HADR configurations
db2 "update db cfg for HADRDB using HADR_TARGET_LIST NULL"
db2 "update db cfg for HADRDB using HADR_SYNCMODE NEARSYNC"
db2 "update db cfg for HADRDB using HADR_REMOTE_HOST db2-2"
db2 "update db cfg for HADRDB using HADR_REMOTE_SVC db2_hadra"
# restart HADR as the standby
db2 deactivate db HADRDB
db2 stop hadr on db HADRDB
db2 start hadr on db HADRDB as standby
db2 activate db HADRDB
```

6.3.5 Test Scenario of Read Scalability

In this section, two scenarios are described to examine the read performance of the cluster for DB2 and the read request distribution. The first scenario focuses on the read performance based on the average response time for read requests per client. Furthermore, the second scenario is designed to evaluate the distribution of read requests across *Pods* in the cluster for DB2. In both scenarios, we deployed a cluster for DB2 named *db2* with two *Pods* (one primary and one principal standby), three *Pods*(one primary, one principal standby and one auxiliary standby) and four *Pods* (one primary, one principal standby and two auxiliary standbys) separately. The names of these *Pods* consist of the cluster name *db2* and its order. In terms of *HADR* roles, *db2-0* is the primary, *db2-1* is the principal standby, *db2-2* and *db2-3* are auxiliary standbys. In order to ensure HA of the cluster for DB2, there must be at least two *Pods* in it. But we need to test the case where only one *Pod* handles read requests as a control for other experiments as well. Nonetheless, we carried it out by sending read-only requests to the primary service, implying that these requests are only sent to one primary *Pod*.

In the first scenario, a client application was realized to simulate various numbers of clients simultaneously sending read requests to the read-only service, with each client sending 1000 read queries. We repeated experiments on different clusters for DB2 with two to four *Pods* in the cases of 10 clients, 20 clients and 30 clients respectively, and calculated the average response time for 1000 read requests per client in all cases. In the Figure 6.15 which shows the results of experiments on read performance, the horizontal axis represents the number of *Pods* handling read requests while the vertical axis indicates the average response time for 1000 read requests per client measured in seconds. Independent of the number of clients connected to the read-only service, there is a gradual decrease in the average response time for 1000 read requests per client as the number of *Pods* handling read requests increases. Specifically, compared with the case of only one *Pod*, the read-only service corresponds to more *Pods* that can provide better read performance.

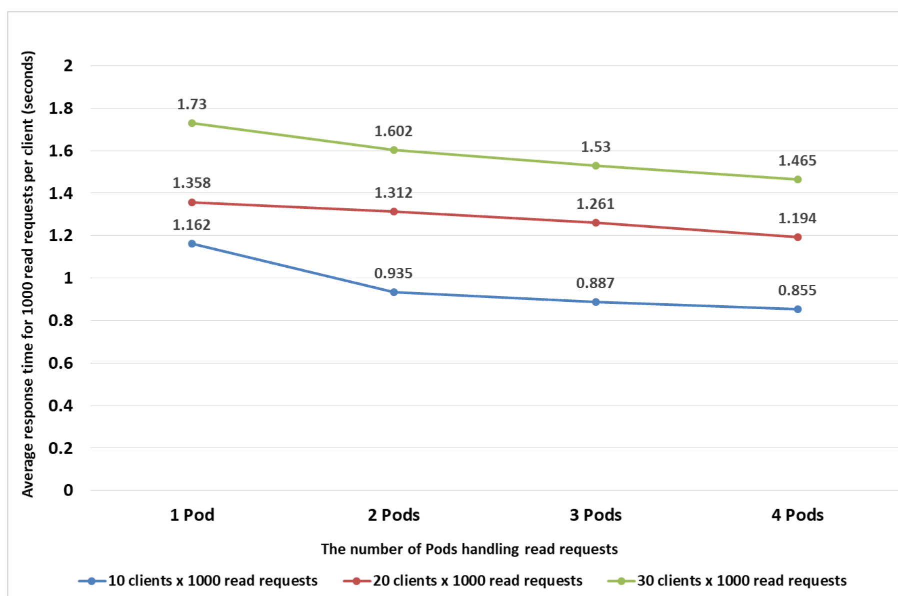


Figure 6.15: Average Response Time for 1000 Read Requests Per Client in Different Clusters for DB2

In the second scenario, we used the client application to simulate that various numbers of clients send read requests to the read-only service at the same time, with each client performing 1000 read queries. These experiments are also repeated on different clusters for DB2 with two to four *Pods* in the cases of 10 clients, 20 clients and 30 clients respectively, and their results are presented in Figure 6.16. Its horizontal axis represents the number of *Pods* handling read requests, and the vertical axis of it indicates how many transactions are handled per *Pod*. Since each client performed a read query in one transaction, the number of transactions collected on each *Pod* can infer how many read requests were handled by it. Regardless of how many clients are connected, as the number of standbys grows, the number of transactions handled by the primary (the blue part in Figure 6.16) falls. This means that the read-only service offers load balancing, and the standbys share some of the traffic with the primary when there are a large number of read requests to be processed.

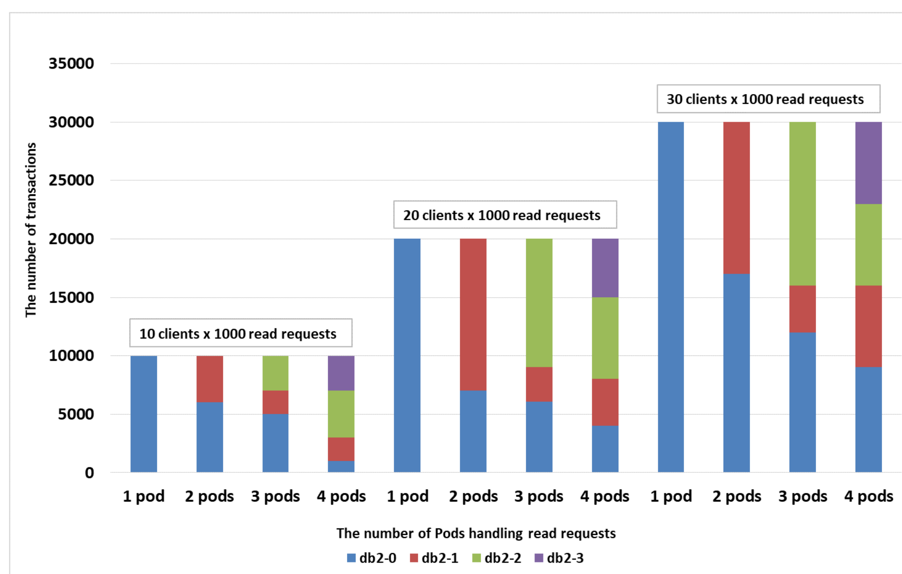


Figure 6.16: Transaction Distribution in Different Clusters for DB2 with Different Numbers of Clients

To sum up, the cluster for DB2 can provide a read-only service with better read performance and distribute read requests to each *Pod* corresponding to it in a load-balanced manner in the event of more standbys.

7 Conclusion and Outlook

This thesis investigated on how to orchestrate data governance workloads as stateful services in cloud environments. More specifically, it focuses on deploying stateful database services in the *Kubernetes* managed cluster for the ECM system. We compared traditional databases (IBM DB2 and PostgreSQL) and modern cloud native databases (Cockroach DB and Google Spanner) with regards to architecture, HA, DR features and horizontal scalability. We saw that each node in cloud native databases is equivalent and can serve read/write operations. In addition, cloud native databases are designed to exploit the cloud's elasticity to scale up or down in response to workload changes. Due to the limitation of architecture design, traditional databases can only guarantee HA through a primary-standby architecture with a failover mechanism, with the primary server serving read/write requests and standby servers handling read-only requests. Based on this analysis, we designed a cluster for DB2 that offers stateful database services, with the primary service focusing on read and write requests while the read-only requests being served exclusively by standby servers. We adopted the IBM DB2 HADR framework to support the HA of the cluster for DB2 and implemented a custom *Governor* component to perform failover automatically between the primary and the principal standby. Incorporating additional auxiliary standbys into the cluster ensures the DR characteristics of the cluster for DB2. Moreover, we designed and implemented a *Kubernetes* operator for DB2 to deploy a cluster for DB2 mentioned above in an automated manner. To prove our approach, we conducted various test scenarios against our prototype in terms of HA, DR and read scalability. The results of the HA tests show that failover can be performed automatically and the primary service can recover in an average of 19.642 seconds in the event the primary server fails. The results of the DR tests illustrate that the primary service can be restored after a disaster by applying a disaster recovery plan manually. The results of read scalability tests demonstrate that adding standbys can improve the read performance of the cluster for DB2 and share read traffic for the primary.

The future works might include implementing automatic failover not only between the primary and all standbys but also between the principal standby and auxiliary standbys, as well as automatically adding or removing standbys based on the volume of traffic requested.

Bibliography

[BDK+12] S. Bartkowski, C. De Buitlear, A. Kalicki, M. Loster, M. Marczewski, A. Mosaad, J. Nelken, M. Soliman, K. Subtil, M. Vrhovnik, et al. *High availability and disaster recovery options for DB2 for Linux, UNIX, and Windows*. IBM Redbooks, 2012 (cit. on p. 20).

[DB2hadr] Db2 HADR development team. *HADR Sychronization Mode*. URL: https://ibm.github.io/db2-hadr-wiki/hadrSyncMode.html (cit. on p. 20).

[DW20] J. Dobies, J. Wood. *Kubernetes operators: Automating the container orchestration platform*. O'Reilly Media, 2020 (cit. on p. 33).

[FLR+14] C. Fehling, F. Leymann, R. Retter, W. Schupeck, P. Arbitter. *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications*. Springer, 2014. DOI: 10.1007/978-3-7091-1568-8 (cit. on p. 26).

[k8s22] Kubernetes. *Kubernetes Documentation*. 2022. URL: https://kubernetes.io/docs/home/ (cit. on pp. 27, 29, 32).

[Luk17] M. Luksa. *Kubernetes in action*. Simon and Schuster, 2017 (cit. on p. 29).

[MSK19] S. Muralidharan, G. Song, H. Ko. "Monitoring and managing iot applications in smart cities using kubernetes". In: *CLOUD COMPUTING* 11 (2019) (cit. on pp. 27, 29).

[NK20] N. Nguyen, T. Kim. "Toward highly scalable load balancing in kubernetes clusters". In: *IEEE Communications Magazine* 58.7 (2020), pp. 78–83 (cit. on pp. 28, 29).

[oprSDK20] Noor Muhammad Malik,Haseeb Tariq,Ish Shah,Fanmin Shi,Eric Stroczynski,Pavlos Ratis,Austin Macdonald. *Operator SDK Documentation*. 2020. URL: https://sdk.operatorframework.io/docs/ (cit. on pp. 33, 56).

[OTH+21] J. S. Omer Kahani, A. J. Thomas Schuetz, G. G. Hongchao Deng, J. K. Noah Kantrowitz, D. M. Philippe Martin, C. S. Roland Pellegrini. *CNCF Operator WHITE PAPER*. Tech. rep. July 2021 (cit. on pp. 31, 32).

[posRep18] Sebastian Insausti. *PostgreSQL Streaming Replication – a Deep Dive*. 2018. URL: https://severalnines.com/blog/postgresql-streaming-replication-deep-dive/ (cit. on p. 21).

[RM17] K. Rabbani, I. Masli. "Cockroachdb: Newsql distributed, cloud native database". In: *Universite Libre De Bruxelles* (2017) (cit. on pp. 22, 23).

[RMM12] T. Ritter, B. Mitschang, C. Mega. "Dynamic provisioning of system topologies in the cloud". In: *Enterprise Interoperability V*. Springer, 2012, pp. 391–401 (cit. on pp. 17, 37–39).

[SD19] J. Shah, D. Dubaria. "Building modern clouds: using docker, kubernetes & Google cloud platform". In: *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)*. IEEE. 2019, pp. 0184–0189 (cit. on p. 28).

[Sha20] G. Shao. "About the design changes required for enabling ECM systems to exploit cloud technology". MA thesis. 2020 (cit. on pp. 17, 35, 36, 38, 39).

[spannerHA22] Albert Harper. *Google Cloud Platform – Introduction to Cloud Spanner*. 2022. URL: https://copyprogramming.com/howto/google-cloud-platform-introduction-to-cloud-spanner (cit. on p. 23).

[Try21] TrybekChristoph. "Investigating the Orchestration of Containerized Enterprise Content Management Workloads in Cloud Environments Using Open Source Cloud Technology Based on Kubernetes and Docker". MA thesis. 2021 (cit. on pp. 17, 36, 38, 39).

[VSTK18] L. A. Vayghan, M. A. Saied, M. Toeroe, F. Khendek. "Deploying microservice based applications with kubernetes: Experiments and lessons learned". In: *2018 IEEE 11th international conference on cloud computing (CLOUD)*. IEEE. 2018, pp. 970–973 (cit. on p. 29).

All links were last followed on December 08, 2022.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature