

École polytechnique de Louvain

High Availability Solution for Slurm in an HPC Cluster using Kubernetes

Author: **Badr-Eddine AJI**

Supervisors: **Etienne RIVIERE, Damien FRANÇOIS**

Reader: **Nicolas DETIENNE**

Academic year 2023–2024

Master [60] in Computer Science

Abstract

This master's thesis aims to study and analyze High-Availability solutions for the workload manager Slurm in an HPC cluster using the container orchestration framework Kubernetes. The primary objective is to explore the possibility of ensuring high availability for Slurm through Kubernetes. This involves both integrating Slurm into a Kubernetes cluster and assessing Kubernetes' capability to guarantee high availability for Slurm. The study focuses on integrating Slurm's controller daemon and database daemon into the Kubernetes environment. It delves into Kubernetes and its potential to provide high availability to Slurm components running in the Kubernetes cluster. Additionally, the thesis covers the selection of Kubernetes-Slurm cluster implementation, the high availability strategy, and presents the results of high availability tests, including simulations of various failure scenarios, to evaluate Kubernetes' capability to ensure High availability and data persistence for Slurm.

Acknowledgement

I would like to express my sincere gratitude to Professor Etienne Riviere for his support and for proposing the topic for my thesis. His guidance, time, and advice were crucial in the completion of this work.

I am also deeply grateful to Mr. Damien François for his assistance throughout this work. His dedication, expertise, continuous feedback, and constructive remarks helped me refine my ideas and overcome challenges.

I would also like to express my gratitude to Mr. Nicolas Detienne for his contribution as a reader to this Master's thesis.

Use of AI

During the completion of this work, generative AI was used to rephrase my texts and translate paragraphs written in French. The Grammarly tool was also used for grammar and spelling correction.

Contents

1	Introduction	6
2	Background	8
2.1	HPC (High-Performance Computing) cluster	8
2.1.1	Network	9
2.1.2	Storage	9
2.2	Slurm Bare-Metal Cluster	9
2.2.1	Slurm	9
2.2.2	Slurm's role in HPC cluster	10
2.2.3	Storage	10
2.2.4	Slurm Architecture	10
2.3	Existing High availability solutions in HPC cluster	11
2.3.1	HA-OSCAR	11
2.3.2	Linux HA	13
3	Overview of High Availability in Kubernetes and Slurm Integration	14
3.1	Kubernetes	14
3.2	Kubernetes architecture	15
3.3	Kubernetes keys components	15
3.3.1	Control plane (Master Node)	16
3.3.2	Nodes (Worker Nodes)	16
3.3.3	Pod	17
3.3.4	Services	17
3.3.5	Storage	18
3.4	Kubernetes High Availability	21
3.5	Integration approaches for Kubernetes and Slurm	22
3.5.1	Existing implementation of Slurm integration on Kubernetes cluster	26

4	System Architecture	28
4.1	Chosen Approach and Methodology	28
4.2	Kubernetes-Slurm cluster implementation	29
4.2.1	Cluster architecture	29
4.2.2	Slurm Images with Docker	35
5	High availability strategy	37
5.1	Tools and technologies required for the study	37
5.2	High availability Architecture for Slurm	38
5.2.1	Multi-node cluster	38
5.2.2	Deployments	39
5.2.3	Services	39
5.2.4	Persistent Storage	40
6	Result and Analysis	41
6.1	Test Scenarios	41
6.1.1	Metrics	41
6.1.2	Slurmctld container failure	42
6.1.3	Slurmctld Pod failure	43
6.1.4	Node failure	44
6.1.5	Extra Measurments	45
7	Conclusion	48
A	Slurm Docker image	52
A.1	Prerequisites	52
A.2	Build Slurm image	52
B	Ansible playbooks	53
B.1	Prerequisites	53
B.2	Create Kind Cluster	53
B.3	Deploy Cluster	53
B.4	Delete Cluster	54

Chapter 1

Introduction

In the field of High-performance Computing (HPC), where speed, reliability, and resource availability are essential, an HPC cluster is a group of connected computer nodes, that work together to perform very difficult and large calculations. These clusters are used for tasks like running jobs, analyzing large amounts of data, and more. In this context of HPC clusters, the existence of a **high-availability** solution within an HPC environment is of paramount importance. High Availability (HA) refers to the ability of a system to remain operational and accessible despite hardware or software failures. High availability is essential to ensure continuous availability of critical services and minimize downtime. HA mechanisms include redundancy, failover mechanisms, and automatic recovery processes designed to maintain uninterrupted operation.

Slurm, is a powerful cluster management tool that efficiently allocates resources and schedules jobs on Linux clusters, and plays a crucial role in optimizing the use of available resources in HPC clusters. To keep Slurm available, it needs a strong high availability (HA) solution to prevent interruptions. There are several HA solutions for HPC clusters and Slurm, such as **LinuxHA**, an open-source implementation for Linux systems, which ensures continuous operation even during hardware or software failures. Another similar solution is **HAproxy**, high-availability load-balancing software, which ensures continuous service availability by evenly distributing network traffic among servers.

At the same time, Kubernetes, a container orchestration framework offers several high-availability features, for containerized applications. Kubernetes can be also a solution to ensure high availability for the Slurm workload manager, but integrating Slurm into a Kubernetes cluster may be more complex. This is because traditional HPC clusters and cloud computing, where Kubernetes is commonly used, have different approaches. Traditional HPC uses powerful bare-metal machines, on the

other hand, cloud computing norm involves virtualization and containers. This difference makes integrating the two environments a complex challenge.

The **CISM [1](Centre de Calcul Intensif et de Stockage de Masse)** is exploring the possibility of integrating Slurm into a Kubernetes cluster. They are specifically looking into integrating the Slurm controller daemon (Slurmctld) and the database daemon (Slurmdbd) while running the Slurmd daemon on bare-metal servers. Additionally, CISM wants also to ensure that Kubernetes can provide an efficient high availability and guarantee persistent storage for Slurm.

This master thesis focuses on the study of the implementation of high-availability solutions in HPC clusters, especially for the Slurm workload manager using Kubernetes. We will explore the possibilities of integrating Slurm into a Kubernetes environment, focusing on ensuring the high availability of Slurm components. This integration aims to create a hybrid HPC environment, where Slurm's advantages in terms of task and resource management are combined with Kubernetes' orchestration and high-availability capabilities. In addition, we will also examine how existing bare-metal servers can be reused and integrated into a Kubernetes cluster.

In this study, we will examine how Kubernetes can effectively ensure high availability for Slurm's critical components and offer a data persistence solution. Our study also aims to investigate the challenges and complexity associated with integrating Slurm into a Kubernetes environment, focusing on high availability and resource efficiency.

Chapter 2

Background

In this chapter, we will explore the different concepts of HPC clusters and their components. We will also provide an overview of the workload manager Slurm and discuss the existing techniques for high availability in a cluster.

2.1 HPC (High-Performance Computing) cluster

An HPC cluster is a set of interconnected computer nodes that work together to perform intensive, large-scale calculations. These clusters are used for applications such as numerical modeling, simulation, massive data analysis, and more. They deliver high computing power using thousands of parallel processors. There are two types of nodes in an HPC cluster:

Management nodes : A management node, known also as the head node, acts as the central managing server, it takes charge of job scheduling, resources monitoring, software deployment, etc. The management node does not participate in computational tasks but indeed, provides and manages resources. In the HPC cluster, it plays an important role by providing a central interface for users to submit and manage jobs, monitor their computations, and allocate resources. Moreover, it also manages the cluster's file system by ensuring that data is stored correctly and accessed easily by computing nodes.

Computing nodes : A computing node in High-Performance Computing (HPC) is an individual server or computer in a cluster that is designed to work on tasks and processes. These nodes come with strong processors, large amounts of memory, and high-speed communication channels for communication among nodes. Every computing node in an HPC cluster works together to carry out difficult calculations

and data-intensive activities, so the fast execution of these processes is allowed by it.



Figure 2.1: CISM HPC cluster [1]

2.1.1 Network

The network in an HPC cluster is crucial as it links all the nodes together, enabling high-speed communication and data sharing. Network architectures in HPC clusters are designed to offer high bandwidth and minimal latency, which is essential for the smooth running of parallel applications.

2.1.2 Storage

Storage in an HPC cluster plays a crucial role, it refers to the infrastructure storage used to store the data required for computing. There are several storage options for high-performance computing jobs. One of the solutions is NFS (Network File System). NFS is a protocol that facilitates remote access to data and files on a network.

2.2 Slurm Bare-Metal Cluster

In a typical HPC environment with a Slurm bare-metal cluster, Slurm plays a central role in the management of computing resources. Here is a detailed analysis of the various aspects of its operation in this context:

2.2.1 Slurm

Slurm is an open-source, fault-tolerant, and highly scalable cluster management and job scheduling system for large and small Linux clusters. Slurm requires no kernel modifications for its operation and is relatively self-contained [2]. The workload Slurm has three principal functions. It allocates access to computing node resources

for users, provides a framework for starting, executing, and monitoring users' work, and resolves conflicts for resources by managing the pending jobs queue. This is possible by compiling commands shown in Figure 2.2, for example, **sbatch** to execute a job, **squeue** pulls up information about jobs in the queue, **sacct** displays accounting data for all jobs.

2.2.2 Slurm's role in HPC cluster

Slurm cluster is coordinated and monitored by the management node. The main roles of Slurm on this node are, first is **job scheduling**, where Slurm receives job requests from users, schedules them according to available resources, and distributes them to the appropriate compute nodes. Second is resource allocation, where Slurm manages all cluster resources (memory, processors, storage devices) and allocates them to running tasks. Third is monitoring and logging, Slurm monitors continuously the status of the cluster and resource utilization while producing logs to enable debugging and performance optimization. The compute node is responsible for executing tasks assigned by Slurm. Slurm interacts with compute nodes through task execution, where it sends instructions to start, stop, and monitor the execution of assigned tasks. It also manages local resources such as memory, CPU, and storage on each compute node to ensure efficient task execution.

2.2.3 Storage

As mentioned before, storage is crucial for data persistence and sharing between different cluster nodes in an HPC environment. Slurm itself is not directly responsible for storage, but it often interacts with distributed file systems such as GPFS (high-performance clustered file system software developed by IBM), BeeGFS (a parallel file system for HPC), or NFS (Network File System) to provide consistent and fast access to the data needed for running tasks.

2.2.4 Slurm Architecture

We defined Slurm as a workload manager for managing and monitoring computing jobs on HPC clusters. The mandatory components of Slurm are, **Slurmctld**, the control daemon which is a centralized manager, to monitor resources and tasks. **Slurmd** (Slurm daemon), is present on each compute server (node), which is waiting for work and responsible for executing compute jobs. The slurmd daemons provide fault-tolerant hierarchical communications. **Slurmdbd** (Slurm DataBase Daemon) is an additional component which is used to record accounting information for multiple Slurm clusters in one database. In Figure 2.2, the Slurmdbd daemon is connected to a database and stores information related to executed tasks. This

enables users to access information about job history. The database can be SQLite, MySQL, or PostgreSQL, depending on the system configuration.

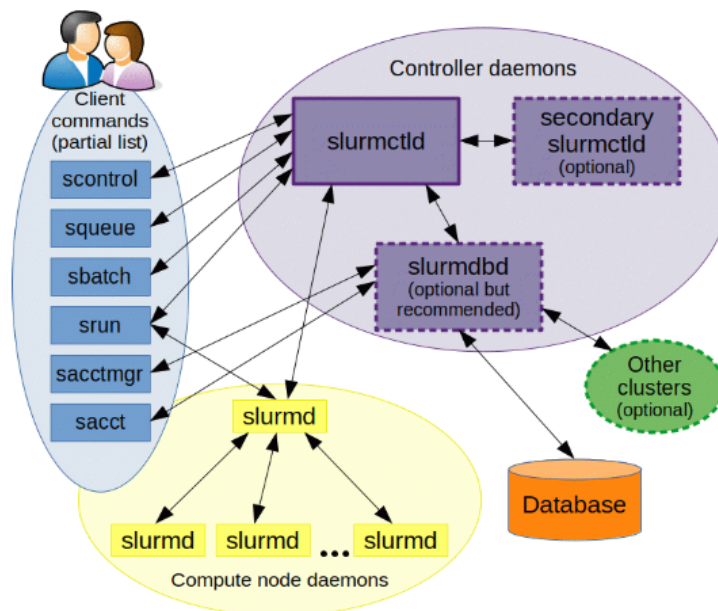


Figure 2.2: Slurm components [2]

2.3 Existing High availability solutions in HPC cluster

In this section, we will explore some existing high-availability solutions in HPC clusters, such as LinuxHA and HA-OSCAR.

2.3.1 HA-OSCAR

HA-OSCAR [3] is an open-source project designed to ensure continuous operation of services in High-Performance Computing (HPC) environments by combining high availability and performance computing solutions. The main goal of the HA-OSCAR project is to enhance the Beowulf cluster system for critical applications and infrastructures that are sensitive to downtime. A Beowulf cluster is a homogeneous computing grid whose components are low-cost PCs.

HA-OSCAR offers several high-availability features, including scalability, security, and availability. It provides head node redundancy and self-recovery for hardware

and services. As shown in Figure 2.3, HA-OSCAR consists of a primary server, which is active and responsible for receiving and distributing requests from clients to the computing nodes. There is also a secondary server, called the standby server, which monitors the primary server to ensure it is functioning properly. If a failure occurs on the active server, the standby server takes over until the primary server recovers. Then the primary server re-takes the lead.

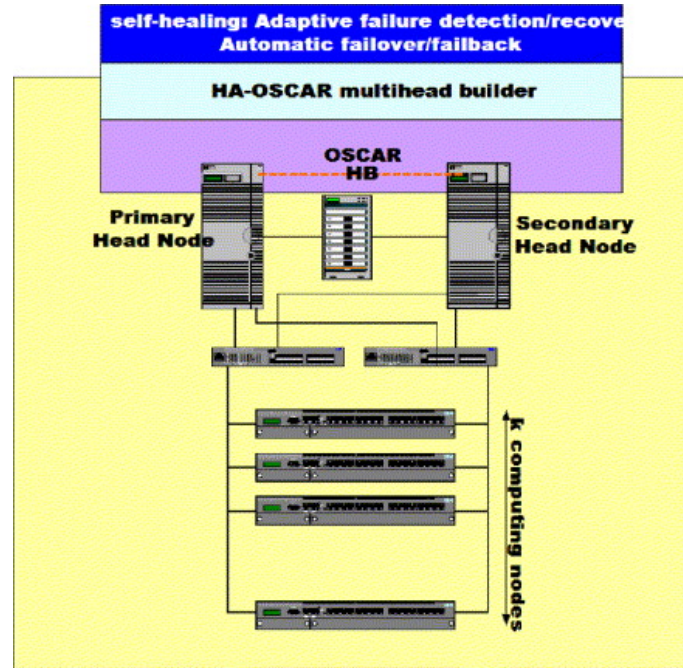


Figure 2.3: HA-OSCAR architecture [3]

This process is managed by a self-healing daemon installed on each head node, which works with SystemImager, an open-source tool used to clone the primary server and its configuration to create the standby server. The Figure 2.4 illustrates the self-healing architecture.

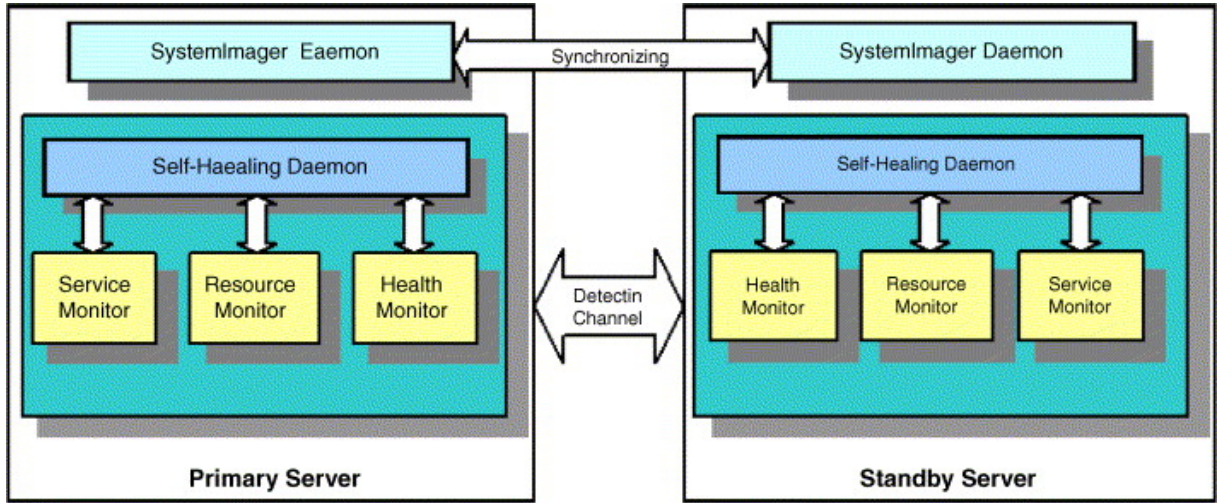


Figure 2.4: HA-OSCAR self-healing architecture [3]

Each head node (primary and standby) has three types of monitors to detect failures: first, the health monitor, which detects hardware, network, and OS issues and performs recovery with a failover server, second, the system service monitor, which checks several critical services like NFS, and finally the resource monitor daemons. In the event of a failure, the self-healing system decides what actions to take based on the history of previous outages, which is stored in a log.

2.3.2 Linux HA

LinuxHA is a widely adopted high-availability solution that enables you to implement high-availability tools within an HPC cluster, guaranteeing the continuous availability of a Linux computing system. Its core component, **Heartbeat**, facilitates communication between a group of machines, enabling them to monitor each other's status and manage resources. In the event of a node failure, these resources can be automatically restarted, transferred to another node, and deleted from the failed node. **Heartbeat** is usually combined with cluster management software such as **Pacemaker**, which supports the management of dependencies between services and the automatic execution of failovers.

Chapter 3

Overview of High Availability in Kubernetes and Slurm Integration

In this chapter, we will take a closer look at how Kubernetes works and ensures high availability. Additionally, we will give an overview of the current state of the art regarding Slurm’s implementation within the Kubernetes cluster and describe the approaches currently employed to achieve this integration.

3.1 Kubernetes

Kubernetes is an open-source container orchestration engine that enables the automated deployment, scaling, and management of containerized applications. It allows the creation and distribution of containerized applications and microservices among the different nodes of an established cluster. Kubernetes manages these applications in different environments, such as physical machines, virtual machines, or cloud environments. Additionally, Kubernetes handles essential functions such as storage, network configuration, and workload placement, and makes deployments easier for developers. Kubernetes enhances deployment speed by enabling rapid deployment of container-based applications, allowing code to move quickly from development to production and facilitating rapid iteration and adaptation to changes. It ensures quick disaster recovery by maintaining the desired state of the system and automatically redeploying containers if failures occur. Additionally, Kubernetes simplifies infrastructure management by handling storage, network configuration, and workload placement, which makes developers free from these complexities compared to traditional deployment methods.

3.2 Kubernetes architecture

To understand Kubernetes architecture as presented in Figure 3.1, it is important to know its key components and their roles within the cluster. First, a Kubernetes cluster consists of multiple nodes, which can be either virtual or physical machines. The control plane, also known as the master node, handles major control functions, including cluster coordination, monitoring, scheduling, and serves as the primary access point for cluster administration. The other nodes are the worker nodes, where containerized applications run. They are responsible for managing applications, such as starting, restarting, and stopping them. They also implement networking to ensure network reachability to containers and services running on these nodes.

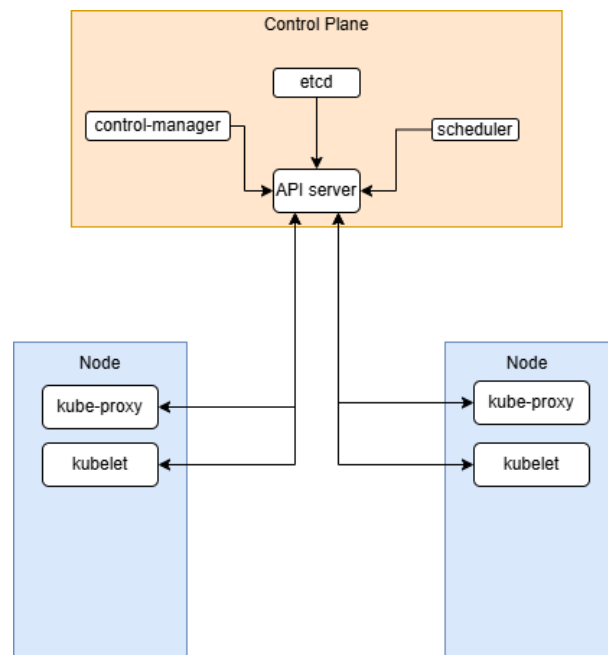


Figure 3.1: Kubernetes Cluster Architecture

3.3 Kubernetes keys components

In this section, we will delve into a detailed explanation of the various components that make up the Kubernetes cluster as shown in Figure 3.1.

3.3.1 Control plane (Master Node)

The brain of the cluster presented in Figure 3.2, manages the desired state, pod scheduling, and health monitoring. The control plane is making sure our cluster is running as expected. Their components are:

1. **API:** The central management entity and the only component that interacts directly with the cluster.
2. **ETCD:** A consistent key-value store used as Kubernetes's backing store. It contains all Kubernetes cluster resources.
3. **Scheduler:** As the name suggests, the scheduler decides which node a pod will run on
4. **Controller Manager:** The last component from the control plane is the controller manager. The controller manager contains several types of controllers that are responsible for aligning the actual state of the cluster with the desired state

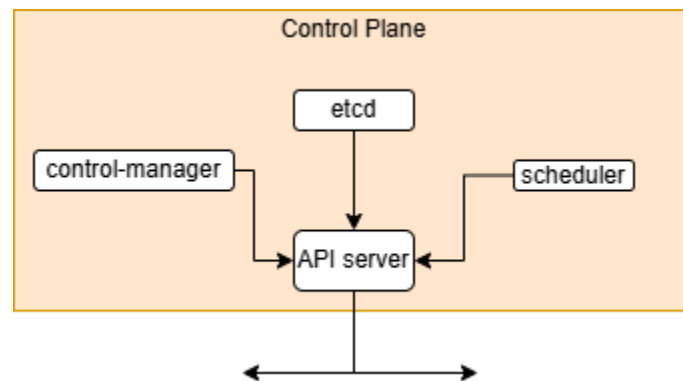


Figure 3.2: Kubernetes Master Node

3.3.2 Nodes (Worker Nodes)

The Node worker is responsible for running containerized applications. It hosts pods, which are the smallest deployable units in Kubernetes and manages the container runtime, networking, and storage for these pods. Their components are :

1. **Kubelet:** As shown in Figure 3.1 every worker node hosts a “kubelet“, a small application that interacts with the Kubernetes control plane(Master node). Kubelet ensures that the containers defined in the pod configuration

are operational on its node and manage their lifecycle. It carries out the instructions given by the control plane.

2. **kube-proxy**: Each worker node runs “kube-proxy” as shown in Figure 3.1, a network proxy that supports Kubernetes networking services. Kube-proxy manages all network traffic within and outside the cluster, either forwarding traffic or utilizing the packet filtering capabilities of the operating system.

3.3.3 Pod

A “pod” is the fundamental unit for packaging and running applications. It can host one or more Docker or similar containers that share resources such as storage and network. These containers are always scheduled and deployed together on the same node in the Kubernetes cluster. Each pod has its own IP, and pods in the same node can communicate using this IP. The pod’s IP is ephemeral and changes when the pod is restarted.

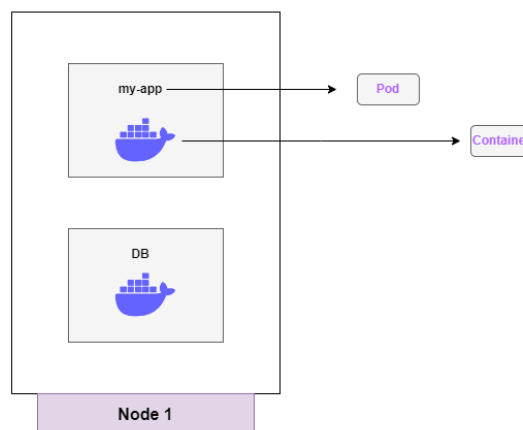


Figure 3.3: Kubernetes Pod

3.3.4 Services

In Kubernetes, each pod has its own temporary IP address. When a pod is deleted or restarted, a new IP address is assigned, which can disrupt communication. To solve this issue, Kubernetes uses services to provide stable and static IP addresses that remain constant even if the pod restarts or dies. Each pod is represented by a service, which ensures persistent and stable IP address access. Kubernetes services

facilitate seamless communication within and outside the cluster, eliminating any loose coupling. There are several types of services in Kubernetes:

1. **ClusterIP service:** The default service type in Kubernetes, it is used to provide connectivity between applications internally. When the ClusterIP service is created, Kubernetes assigns a virtual IP address to this service, making it accessible by pods running in the cluster. This virtual IP is persistent even if the pods are restarted or shut down. The ClusterIP service is a good choice for internal communication within the cluster.
2. **NodePort service:** NodePort service extends the functionality of ClusterIP by allowing external access to applications. When a NodePort service is created, Kubernetes opens up a port number from a predefined range of 30000-32767 to forward traffic to the corresponding ClusterIP service. NodePort service is ideal for applications that need to be reachable from outside by using the node's IP and the assigned port number.
3. **LoadBalancer service :** provides external access to applications. When the LoadBalancer service is created, Kubernetes provisions a load balancer in its cloud environment and forwards the traffic to the nodes running the services. This powerful service is ideal for applications that need to handle high traffic volumes. It makes access to applications easier by using a single IP address assigned to the load balancer during its creation.

3.3.5 Storage

Storage in Kubernetes is managed through volumes, abstracted storage units that allow Pods to store and share data. Especially the persistent volumes, which can be divided into two types: static, which are created by cluster administrators and are available for cluster users to consume, and dynamic, which are created automatically when users need it[4].

1. **Persistent Volume Claim (PVC) :** The Persistent Volume Claim is a Kubernetes resource used to request persistent storage space in a cluster. It is essentially a request for a certain type of storage specified by the developers, enabling applications to guarantee durable storage for their data, even when pods are deleted or recreated.

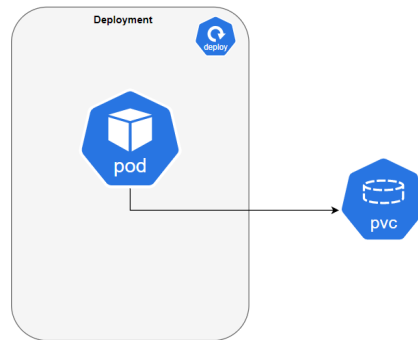


Figure 3.4: Persistent Volume Claim

2. **Persistent Volume (PV)** : Persistent Volume provides an abstraction for physical storage in Kubernetes. It enables cluster administrators to provision storage resources and make them available to users via PVCs. PVCs guarantee data persistence, regardless of the lifetime of the associated pods, by providing storage that persists even if pods are deleted, restarted, or recreated. They also offer the flexibility to connect to various storage sources, whether local storage on cluster nodes the case in Figure 3.5 or remote storage, such as cloud storage services, network file systems, or distributed storage solutions.

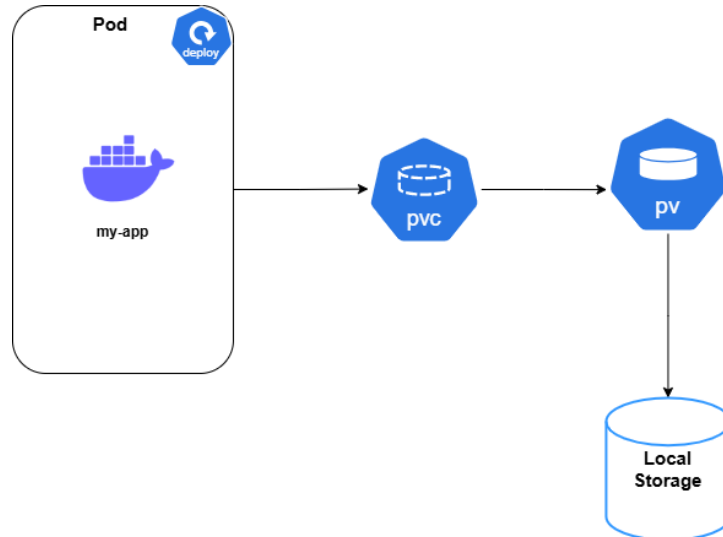


Figure 3.5: Persistent Volume

3. **Storage Class volume** : In a Kubernetes cluster, deploying multiple pods and manually managing the Persistent Volumes (PVs) required for each pod

can become complex. Each pod may have different storage requirements, so statically managing these PVs can be almost impossible. To address this issue, Kubernetes uses Storage Classes. As shown in Figure 3.6 Storage Class defines storage types with specific characteristics and automatically provisions the persistent volumes required for pods based on the needs specified by developers via a PV controller.

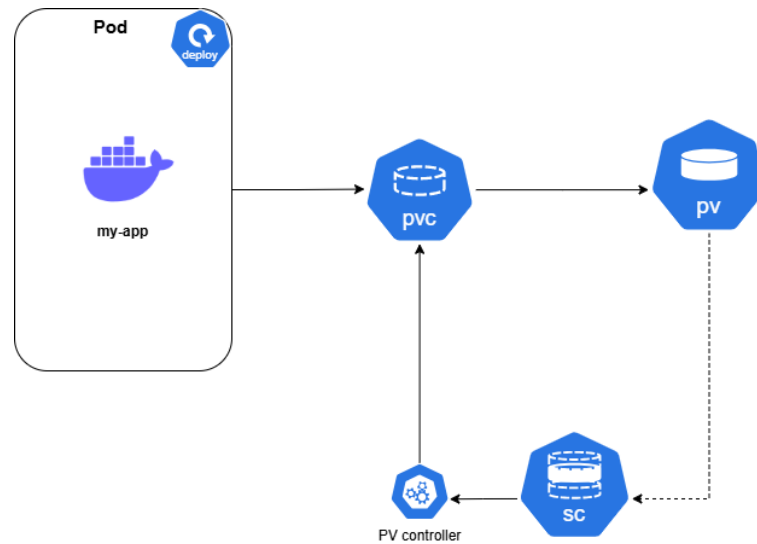


Figure 3.6: Storage Class Volume

4. **NFS volume** Additionally, Kubernetes offers another type of volume, **NFS (Network File System)**. An NFS volume enables the mounting of an existing NFS share into a pod or more Pods. The contents of an NFS volume are preserved, and the volume is unmounted when not in use. This means an NFS volume can be filled with data beforehand, and this data can be shared between Pods. Furthermore, NFS supports simultaneous mounting by multiple writers[4].

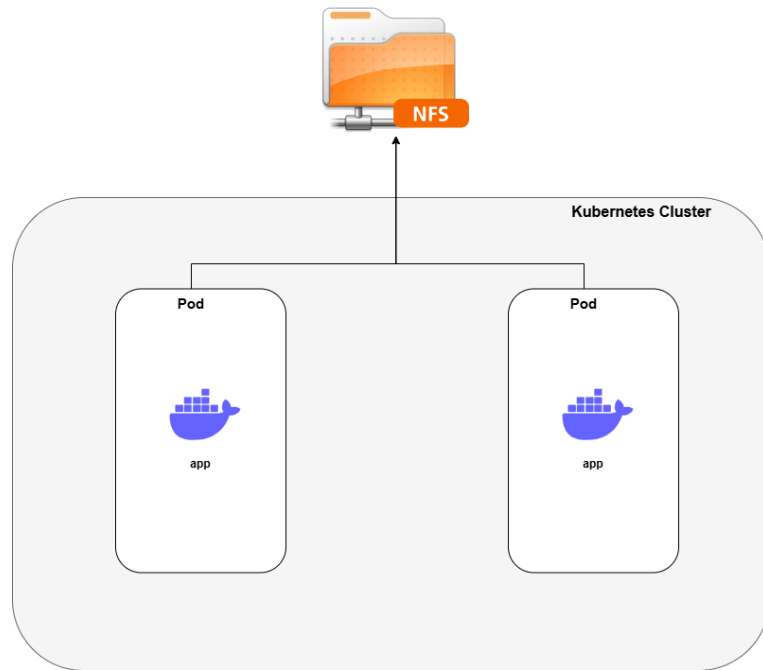


Figure 3.7: Network File System Volume

3.4 Kubernetes High Availability

High availability (HA) refers to the ability to maintain the functionality and accessibility of a system or application even during failures or interruptions. In Kubernetes, high availability aims to reduce downtime by ensuring that services and applications continue to run despite failures at different levels, such as nodes, pods, or network components. High availability (HA) in Kubernetes relies on several key mechanisms to ensure resilience and continuity of operations in clusters. By combining these features, Kubernetes provides a robust platform for ensuring high availability :

1. **ReplicationController:** Essential components such as the API Server, Controller Manager, and Scheduler are replicated in master nodes to ensure continuous availability. If one master node fails, another automatically takes over.
2. **Services:** Ensure interaction between different components inside and outside the Kubernetes cluster. Services facilitate connectivity between pods and external applications by providing static IP addresses and DNS names.

3. **Deployment:** A Deployment in Kubernetes manages a set of replicated pods, ensuring that an application is always available and can be updated or scaled without any interruptions. It allows for the deployment of an application and updates its image in a controlled manner. In case of issues, a Deployment can be rolled back to a previous version or deleted and redeployed as a new one, this ensures that applications can dynamically adapt to demand without compromising availability.
4. **Load balancing and fault tolerance:** Kubernetes offers integrated load balancing and fault tolerance. Services are exposed via load balancers that distribute traffic between replicated pods, while Kubernetes automatically detects and manages node failures by redeploying pods to other nodes.
5. **Persistent storage :** Persistent storage plays a crucial role in ensuring high availability, as discussed in the previous sections. Kubernetes utilizes Persistent Volumes to guarantee data availability and consistency, even when the pod restarts or dies. To enable dynamic and automatic storage provisioning, especially when dealing with a large number of pods, Kubernetes employs the Storage Class component to ensure that the storage backend remains available and resilient for the application. Additionally, Kubernetes provides data redundancy options, such as storage volume replication using technologies like Ceph, NFS, and cloud provider storage, to ensure that data is replicated across multiple nodes and prevent data loss in the event of a failure.

3.5 Integration approaches for Kubernetes and Slurm

In this section, we will explore how Slurm can be integrated with Kubernetes. Slurm is designed to manage large computer jobs, while Kubernetes is responsible for handling numerous small tasks within containers. **SchedMD**, the main developer of the Slurm workload manager, has looked into the possibility of integrating Slurm into Kubernetes. This integration would allow HPC and Cloud Native workloads to converge. They have proposed different approaches to integrating Slurm into Kubernetes, including scenarios where Slurm has full control, Kubernetes has full control, and where the two collaborate equally. **SchedMD** has identified 4 possible approaches to integrating Slurm into Kubernetes: Over, Distant, Adjacent, and Under. Let's take a closer look at each approach and how it functions [5].

1. **Over:** Slurm is deployed on top of Kubernetes, this approach aims to have Slurm manage all cluster resources. In this setup, Kubernetes clusters are

created temporarily within Slurm batch jobs. However, the Kubernetes control plane (Kubernetes Master Node) is only available once the job has been launched, or it must be hosted outside the traditional cluster. **SchedMD** find this solution to be not particularly useful beyond test and development environments [5].

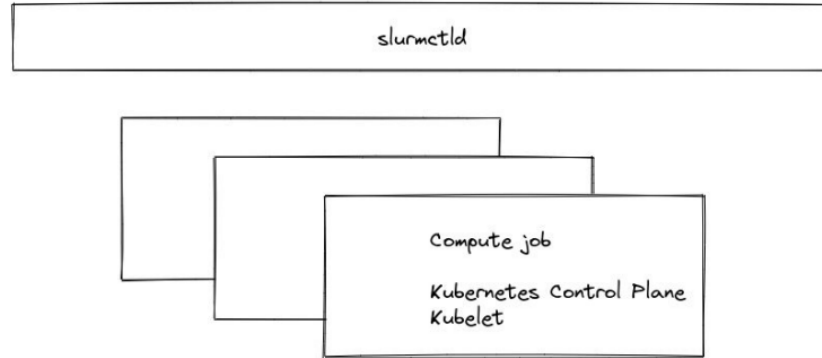


Figure 3.8: Over approach [5]

2. **Distant:** Kubernetes and Slurm operate independently without direct integration, with each managing its own resources and workloads. Therefore, neither Slurm nor Kubernetes is aware of the current resources and demands of the other environment. An additional management tool such as **Dell's Omnia toolkit**, Omnia is open-source software for deploying and managing high-performance clusters for HPC, AI, and data analytics workloads, it installs Slurm and/or Kubernetes for managing jobs and enables installation of many other packages and services for running diverse workloads on the same converged solution [6], can be used to handle assignment of resources between both environment [5].

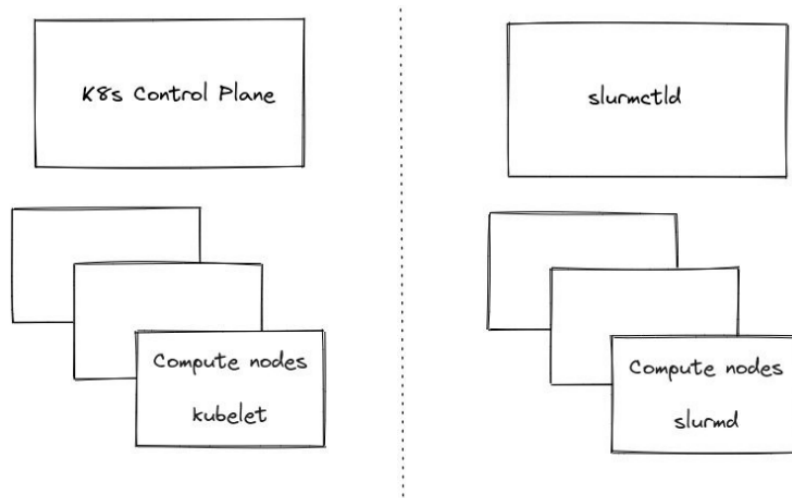


Figure 3.9: Distant approach [5]

3. **Adjacent:** For the Adjacent approach, **SchedMD** proposes to run Slurm and Kubernetes together by installing a Kubernetes scheduling plugin for Slurm. This enables Slurm to manage the workloads of both systems. Kubernetes jobs are managed by the Kubelet, offering all the functionality of Kubernetes, while Slurm jobs are managed by Slurm. However, there are limitations: Kubernetes scheduling is still done at the node level, and some aspects of Kubernetes scheduling, such as affinity (where Nodes/Pods are scheduled together or close to each other) and anti-affinity (where Nodes/Pods are scheduled apart from each other) are difficult to integrate into Slurm [5].

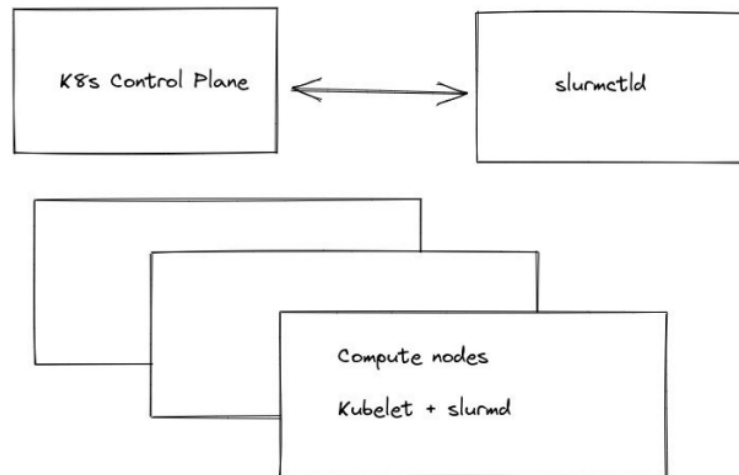


Figure 3.10: Adjacent approach [5]

4. **Under:** This approach aims to run a Slurm cluster within a Kubernetes environment. The goal is to provide a familiar experience to Slurm users while taking advantage of the benefits of container orchestration. Kubernetes-native cloud providers are becoming increasingly common, and major cloud providers are offering managed Kubernetes services. In this approach, persistent connection nodes are running as Kubernetes Pods, preserving the traditional Slurm user experience and improving user isolation. Kubernetes can implement automatic scaling, a key feature for efficient resource utilization. Slurm 22.05+ simplifies this process with its dynamic nodes feature [5].

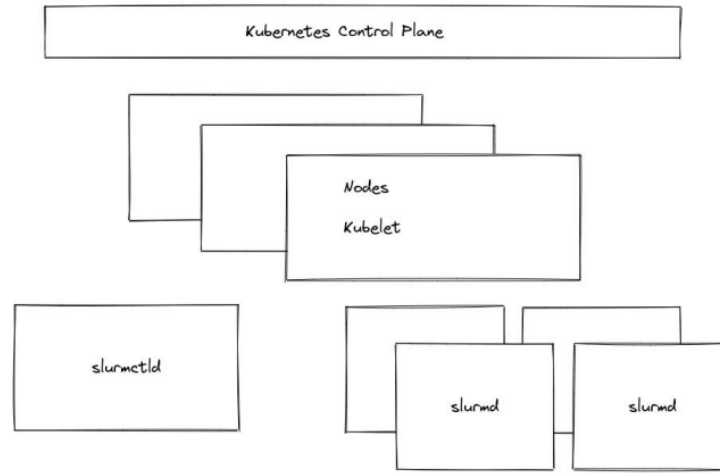


Figure 3.11: Under approach [5]

SchedMD and CoreWeave[7], a cloud service provider, are working in collaboration on an innovative project called **SUNK**. CoreWeave uses Kubernetes to manage its bare-metal clusters. The aim of “SUNK” (“[S]lurm o[n] [K]ubernetes”) is to use Kubernetes to deploy and manage Slurm clusters on bare-metal hardware. The Kubernetes operator deploys and monitors the Slurm cluster via the API server, which enables nodes to be dynamically added and removed from the cluster to adjust capacity as needed. The Kubernetes Scheduling plugin (Sched) also facilitates the integration of Kubernetes workloads into the Slurm cluster, enabling unified resource management. This model combines the Under and Adjacent approaches for maximum efficiency.

3.5.1 Existing implementation of Slurm integration on Kubernetes cluster

StackHPC [8] has developed an approach similar to the SUNK project. This approach combines the planning and scheduling capabilities of Slurm with the orchestration and container management capabilities of Kubernetes. This hybrid implementation efficiently provisions and manages HPC resources using Kubernetes while leveraging Slurm’s advanced task management features. However, there is a small difference between the solution proposed in the **SUNK** project and **StackHPC**’s approach. **StackHPC** has focused more on the “Under“ approach, where Kubernetes has full control of the cluster. As shown in the figure, all Slurm components run within a Kubernetes cluster.

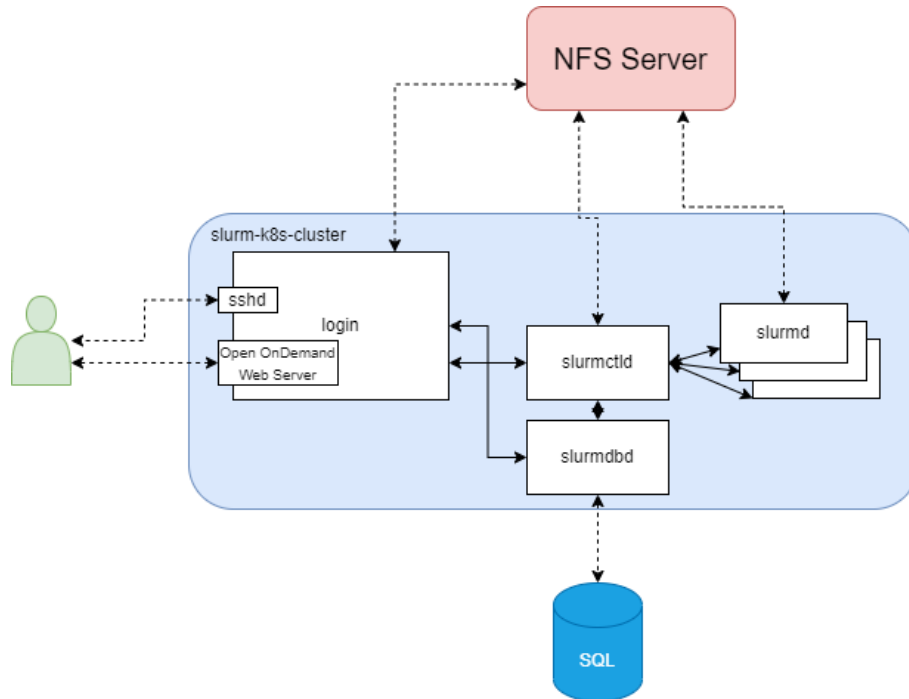


Figure 3.12: Exploring Slurm on Kubernetes cluster [9]

StackHPC has identified several potential benefits to this integration, including simplified deployment and management of Slurm clusters using Kubernetes features, as well as the ability to leverage Kubernetes’ auto-scaling capabilities for hybrid cloud environments.

StackHPC has opted for a full implementation approach with **Kubernetes**, exploring the integration of Slurm with **Kubernetes**. The idea is to provide a separate pod for each Slurm daemon, as shown in Figure 3.12. In addition to

running Slurm entirely within a Kubernetes cluster, **StackHPC** has also added other components to mimic important aspects of a traditional Slurm cluster. For example, they have introduced a login Pod that gives users access to Slurm commands and a shared work directory. This demonstrates that Kubernetes can encompass the entire Slurm cluster and related components, such as login and access to Slurm.

- **Slurmctld**: a single pod running the control daemon responsible for managing the job queue, monitoring work nodes, and allocating resources
- **Slurmd**: several pods running the daemon for working nodes.
- **Slurmdbd**: a single pod providing the interface to the database used to persist accounting records.

Slurm requires a shared filesystem, so **StackHPC** developed a HelmChart to deploy RookNFS, an open-source cloud-native storage orchestrator. RookNFS provides the platform, framework, and support for integrating a diverse set of storage solutions with cloud-native environments [10]. It is connected with Kubernetes persistent volume and provides data persistence to the **Slurmctld** daemon.

In this thesis, and according to the requirements expressed by CISM, the entire system runs on a bare-metal cluster, and the aim is not to migrate Slurm completely to a Kubernetes cluster, but rather to consider a hybrid solution to build a Bare-metal - Kubernetes cluster, where the **Slurmctld** and **Slurmdbd** components run in a Kubernetes cluster, while the other daemons remain on bare metal. The aim is to guarantee robust communication between the two environments and provide high availability for **Slurmctld** and **Slurmdbd**, while not forgetting storage persistence.

Chapter 4

System Architecture

In this chapter, we will present the implementation of the Kubernetes-Slurm cluster, providing a detailed explanation of the implementation for each component, including the choice of approach for integrating Slurm into a Kubernetes cluster.

4.1 Chosen Approach and Methodology

To implement the Slurm Kubernetes cluster, it was essential to choose an efficient approach that combined the Kubernetes and Slurm environments. CISM opted for the idea of integrating part of Slurm into the Kubernetes environment, while the other part would continue to run on bare-metal servers. To achieve this, CISM found that combining the Adjacent and Under approaches would be robust and efficient. By combining these two approaches, we can leverage the advantages of each environment. The Adjacent approach allows us to maintain Kubernetes-specific functionality while integrating Slurm, while the Under approach allows us to benefit from container orchestration and automatic scaling while maintaining a consistent user experience for Slurm users. This combination guarantees solid, efficient integration of Slurm and Kubernetes, optimizing resource and workload management.

The components that will be implemented into the Kubernetes cluster are Slurmctld the controller daemon and its database Slurmdbd. Each component will be containerized and run into a unique pod, so one pod for Slurmctld and one for Slurmdbd. these two pods need to communicate with each other so, a service type ClusterIP was created and ensure communication between them. Each pod has its own PVC which is created during the deployment those PVCs are connected to the global PV which uses the local storage of the node, so if a pod is restarted the data is still there.

4.2 Kubernetes-Slurm cluster implementation

In this section, we will provide an overview of our Kubernetes-Slurm cluster architecture, detailing each component of the cluster and explaining our implementation choices.

4.2.1 Cluster architecture

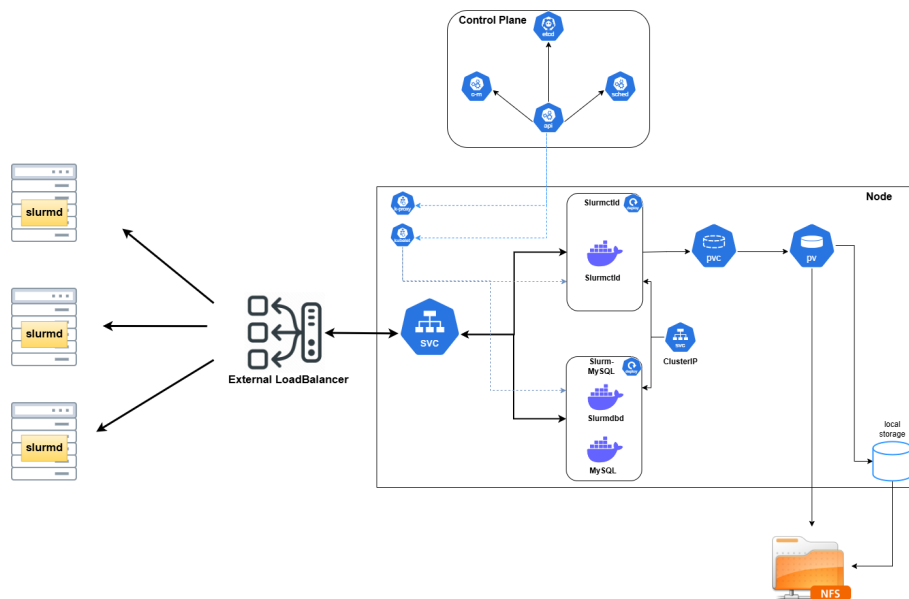


Figure 4.1: Kubernetes Slurm architecture

In this Cluster architecture, we integrate Slurm into a Kubernetes cluster using both approaches the “Under” approach which aims to run Slurm within a Kubernetes cluster, and the “Adjacent” approach which aims to run Slurm and Kubernetes at the same level by installing a Kubernetes scheduling plugin for Slurm, specifically the **Slurmctld** and **Slurmdbd** components. We create individual pods for each Slurm daemon, with communication facilitated via a **ClusterIP** service, enabling **Slurmctld** to communicate with **Slurmdbd** and vice versa.

For **Slurmdbd**’s pod, we have a container for **Slurmdbd** itself and another container for **MySQL**. To ensure data persistence for **Slurmctld**, we utilize a **PersistentVolumeClaim (PVC)** and **PersistentVolume (PV)** with a **hostPath** for storage. This setup ensures that the data remains on the node, providing **Slurmctld** with continuous access, even after restarts or failures.

The **Slurmds** running on bare metal clusters need to connect to the **Slurmctld** controller. However, due to being outside Kubernetes, there is an issue with changing pod IPs. To address this, we set up a **LoadBalancer** service to expose

Slurmctld on port 6817, providing a static IP using **MetalLB**, a load-balancer implementation for bare metal Kubernetes clusters, using standard routing protocols [11]. Simply adding this static IP to the **Slurmds** configuration establishes the connection.

1. **Nodes** : Our Kubernetes cluster consists of one master node and three worker nodes. The control plane(Master node) is responsible for managing the cluster and its resources, including the worker nodes and pods. The master node receives information about cluster activity, internal and external requests, etc. Based on this information, the control plane ensures that the cluster is always in the desired state. The worker nodes host the pods, manage resources for the pods according to their needs, and communicate the current state of the containers and pods to the control plane via the API.

2. Pods

- Slurmctld: The Slurmctld controller daemon runs in a Docker container, which operates within a pod named Slurmctld.

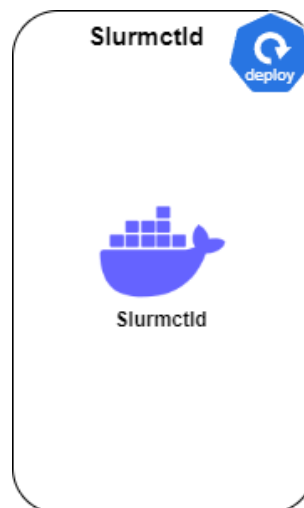


Figure 4.2: Slurmctld Pod

- **Slurm-MySql**: In the slurm-mysql pod, there are two containers. The first container runs the Slurmdbd daemon, which needs a MySQL database container for data storage. The two containers communicate with each other via the pod's IP address. During the build of the Slurmdbd Docker image, the pod's IP address is retrieved and automatically added to the Slurmdbd configuration.

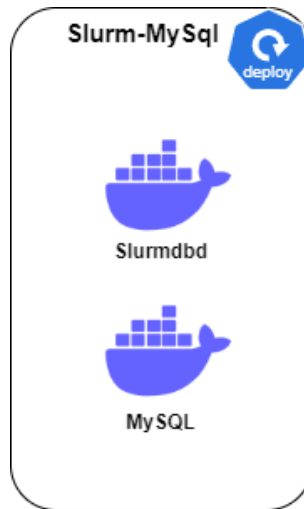


Figure 4.3: Slurm-MySql Pod

3. **Deployment configs:** It is possible to create a pod via YAML configurations using **Kind: Pod**, in the Kubernetes YAML configuration file, the **kind** field indicates the type of Kubernetes resource being defined within the file. We chose the **Deployment** type instead of Pod because it allows us to create replicas of the pod, and in case of interruption or crash, the pod is automatically recreated or restarted. We will discuss this in more detail in the High Availability Strategy chapter.

- Slurmctld deployment

4. Services

- **ClusterIP:** The ClusterIP service allows internal communication between pods. As mentioned, the Slurm daemons Slurmctld and Slurmdbd need to communicate with each other. Slurmctld needs Slurmdbd's host in its configuration. For that, we configured a service type ClusterIP which is exposed to Slurm-MySql on port 6819. Thanks to this service we have a persistent IP, which provides efficient communication between the two pods. We can see in Figure 4.4 how the service ClusterIP is implemented.

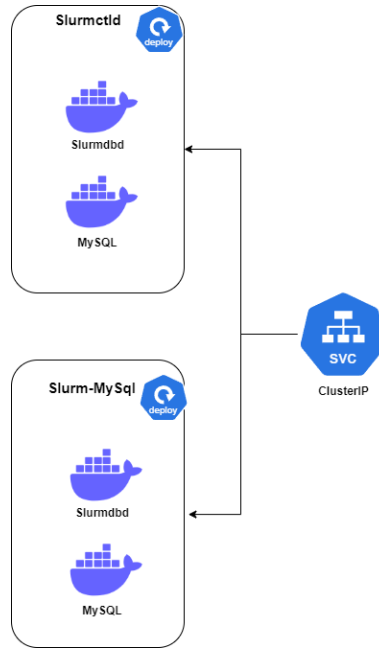


Figure 4.4: ClusterIP service for internal communication

- **LoadBalancer:** The Slurm daemons always require a Slurmctld controller, as seen in the Slurm cluster architecture. Therefore, it is essential for Slurmctld to be reachable. In the Slurmd configuration, we need the Slurmctld host to be accessible either via a DNS or an IP address. The Slurmd daemons are separate from Slurmctld in our cluster; the Slurmds run outside of the Kubernetes cluster while Slurmctld runs in a Kubernetes pod. To establish communication between these two environments, we need to set up an external connection and as a solution, we implemented a LoadBalancer service with a fixed IP address provided by the MetalLB tool which is exposed to the Slurmctld application on port 6817.

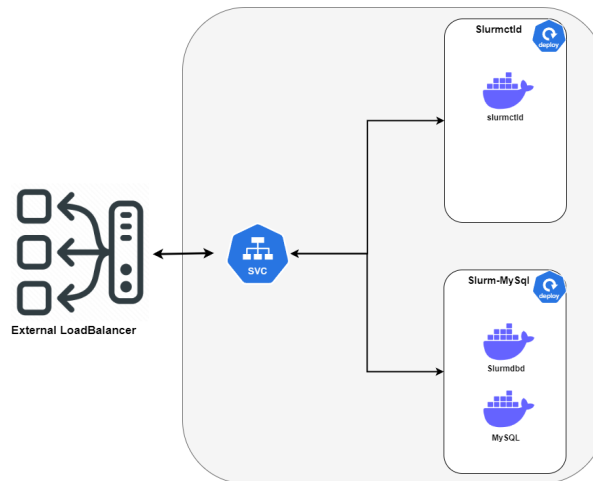


Figure 4.5: Loadbalancer service for external communication

5. Volumes

- **Persistent Volume Claim** To ensure that the data from Slurmctld remains persistent, it was necessary to create a Persistent Volume Claim (PVC). A PVC can be created via a YAML file as follows :

```

1  apiVersion: v1
2  kind: PersistentVolumeClaim
3  metadata:
4    name: slurm-pvc
5  spec:
6    accessModes:
7      - ReadWriteMany
8    resources:
9      requests:
10     storage: 5Gi

```

As shown in Figure 4.6, the pod Slurmctld is linked to a PVC, for that, in Slurmctld deployment configuration we add the following block to persist Slurmctld's data :

```

1      ....
2      volumeMounts:
3        - name: slurm-state-volume
4          mountPath: /var/lib/slurmd #(8)
5  volumes:
6    - name: slurm-state-volume

```

```

7     persistentVolumeClaim:
8       claimName: slurm-pvc  #(9)

```

In this code snippet, we see how a Persistent Volume Claim (PVC) is associated with a pod to ensure data persistence. The **volumesMounts** section defines a volume named **slurm-state-volume**, which is linked to a PVC called **slurm-pvc**. This setup ensures that the volume is backed by persistent storage. In the volumeMounts section, the slurm-state-volume is mounted to the **var/lib/slurmd** path inside the container.

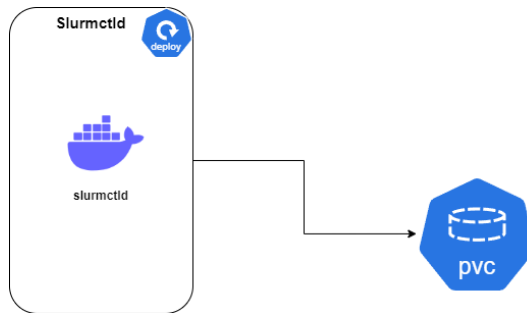


Figure 4.6: Slurmctld Persistent Volume Claim

- **Persistent Volume** To ensure data persistence efficiently, it is mandatory to link the PVC to a PV. PVCs are ephemeral and disappear when the pod dies, so to persist Slurmctld data, we create a Persistent Volume that uses the local storage of the node. The PVC is then linked to this PV to maintain data persistence.

```

1  apiVersion: v1
2  kind: PersistentVolume
3  metadata:
4    name: slurm-pv
5  spec:
6    storageClassName: standard
7    capacity:
8      storage: 5Gi
9    volumeMode: Filesystem
10   accessModes:
11     - ReadWriteMany
12   persistentVolumeReclaimPolicy: Retain
13   hostPath:
14     path: "/var/lib/slurmd"

```

The YAML configuration defines a Persistent Volume (PV) named `slurm-pv` with a storage capacity of 5 GiB. It uses the standard storage class and operates in **Filesystem** mode. The volume has a **ReadWriteMany** access mode, allowing multiple pods to read from and write to it simultaneously. The reclaim policy is set to **Retain**, meaning the data will be preserved even after the volume is deleted. As shown in Figure 4.7 the volume uses the local `/var/lib/slurmd` directory on the Node for storage.

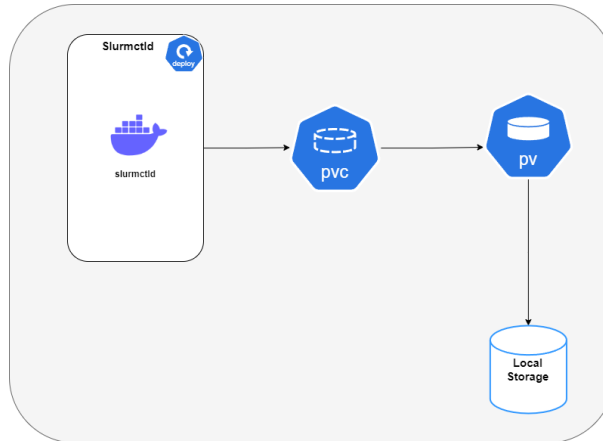


Figure 4.7: Slurmctld Persistent Volume

4.2.2 Slurm Images with Docker

To run the Slurm application in a pod, it was necessary to create a Docker container with a Slurm image. The images available on DockerHub were not compatible with the version of Slurm installed outside the Kubernetes cluster. So, we took an existing Slurm image[12], adapted the Slurm version, and modified it to make it compatible with the desired version and to fit the configurations of our Kubernetes cluster.

Slurmctld and Slurmdbd each need their own configuration: Slurmctld uses “`slurm.conf`” and Slurmdbd uses “`slurmdbd.conf`”.

For example, in “`slurm.conf`” configuration, Slurmctld needs the slurmdbd host for storage in the field **AccountingStorageHost**. As explained before, Slurmctld and Slurmdbd communicate via a **ClusterIP** service, so we added the service name in the `slurm.conf` configuration, especially in “`AccountingStorageHost`”.

```

1 AccountingStorageType=accounting_storage/slurmdbd
2 AccountingStorageHost=slurmdbd-service <--- ``ClusterIP service``

```

```

3 AccountingStoragePort=6819
4 #AccountingStorageLoc=slurm_acct_db

```

Slurmdbd communicates with the MySQL container, which is in the same pod, using the pod's IP address. However, the pod's IP address is ephemeral and changes frequently. To solve this, we gave the variable **StorageHost** the value `@storagehost@`, which is automatically replaced with the pod's IP address during deployment by executing the following "command" :

```

1 name: slurmdbd
2 image: badreddine970/slurmctld-prod:latest
3 command:
4   - "sh"
5   - "-c"
6   - |
7     sed -i "s/@storagehost@/$(hostname -i)/" /etc/slurm/slurmdbd.conf

```

When the container is created, `@storagehost@` will be changed to the pod IP, which will be obtained by running the command `hostname -i`.

Before deploying :

```

1 # Database info
2 StorageType=accounting_storage/mysql
3 StorageHost=@storagehost@
4 StorageUser=slurm
5 StoragePass=password
6 #StorageLoc=slurm_acct_db

```

After deploying :

```

1 # Database info
2 StorageType=accounting_storage/mysql
3 StorageHost=10.244.1.2
4 StorageUser=slurm
5 StoragePass=password
6 #StorageLoc=slurm_acct_db

```

We configured the `Slurmctld` and `Slurmdbd` image during the build process based on the specific needs and resources of our cluster. In the future, CISM can use the same Dockerfile and adjust the version of Slurm for compatibility with `slurmd` running on bare-metal servers, `slurm.config`, and `slurmdbd.config` for internal and external cluster communication.

Chapter 5

High availability strategy

In this chapter, we will discuss the strategy we implemented to ensure the high availability of the Slurmctld and Slurmdbd components using the appropriate Kubernetes components. We will also cover the strategy followed to ensure storage persistence and communication with the external environment, especially with the Slurmd daemon which is running on a bare-metal cluster.

5.1 Tools and technologies required for the study

To study the high availability of our implementation and develop a solid strategy, we built our cluster using a Kubernetes tool called Kind. **Kind** (short for Kubernetes in Docker) is a set of tools for creating local multi-node clusters where each node is a Docker container. It is designed for testing purposes [13]. We also used the **kubect**l tool, a Kubernetes tool, to deploy the different components of the Slurm-Kubernetes cluster. Kubectl lets the user run various commands on the Kubernetes cluster such as deploying applications, managing cluster resources, and monitoring logs. For automating the deployment of the cluster infrastructure, we created a **YAML** file configuration for each component (deployment configuration, service configuration, storage configuration, etc.).

These configurations were deployed using an Ansible playbook. **Ansible** is an open-source IT automation engine that automates provisioning, configuration management, application deployment, orchestration, and many other IT processes[14], which utilizes the **kubect**l command in a defined logical order, effectively making infrastructure as code. Infrastructure as code or IaC, involves managing and provisioning infrastructure using lines of code rather than manual processes.

5.2 High availability Architecture for Slurm

In this section, we will discuss how high availability for Slurm is ensured by implementing the necessary Kubernetes components and integrating them with Slurm. We will detail the specific Kubernetes resources used, such as Deployments, Services, and Persistent Volumes, and explain how they help maintain the availability and reliability of the Slurm workload manager. Through this approach, we aim to provide a comprehensive strategy for achieving and maintaining high availability for Slurm in a Kubernetes cluster.

5.2.1 Multi-node cluster

We created a multi-node cluster with a master node and worker nodes. The worker nodes host the Slurmctld and Slurmdbd pods and continuously report their status to the master node using the Kubelet component. As shown in Figure 5.1 if a worker node fails, either by reboot or shutdown, the communication with the master node stops. The master node then recognizes that this node is no longer ready to host pods, changes its status to “Not ready“, and automatically reschedules the pods to other nodes that are still ready. This ensures that the pods remain available even if some nodes fail. This can also apply to the master node. Therefore, it is preferable to always have another master node as a backup. In case of a failure in the active master node, Kubernetes will switch automatically to the passive master node, making it the new active node.

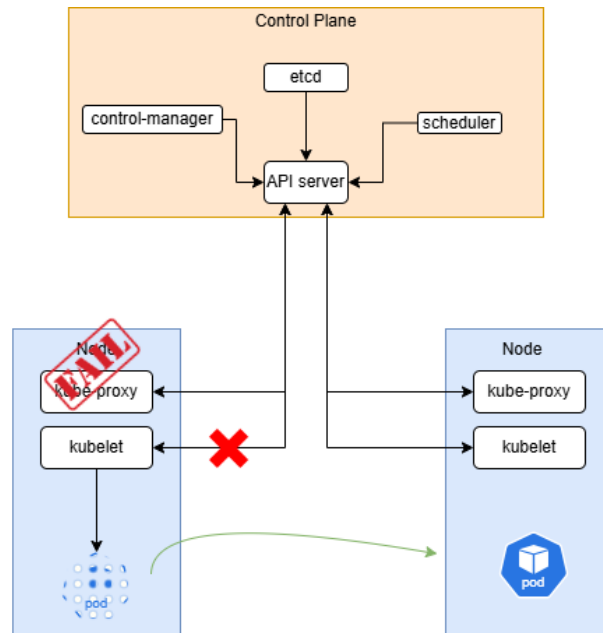


Figure 5.1: Worker node failure

5.2.2 Deployments

For both the Slurmctld and Slurmdbd daemons, we chose to use the **Deployment** type instead of the Pod type or Statefulset type. This allows us to create the Slurmctld and Slurmdbd pods and contribute to the high availability of both pods by **automatically** redeploying them if a failure occurs or if the pod is deleted as shown in Figure 5.2. This is facilitated by the Kubelet component, which continuously monitors the state of the pod. In the event of a failure, Kubelet will stop incoming requests to that pod, restart the pod automatically, and make it available again. This ensures the availability of the pod despite any failures that may occur, whether within the container running in the pod or the pod itself.

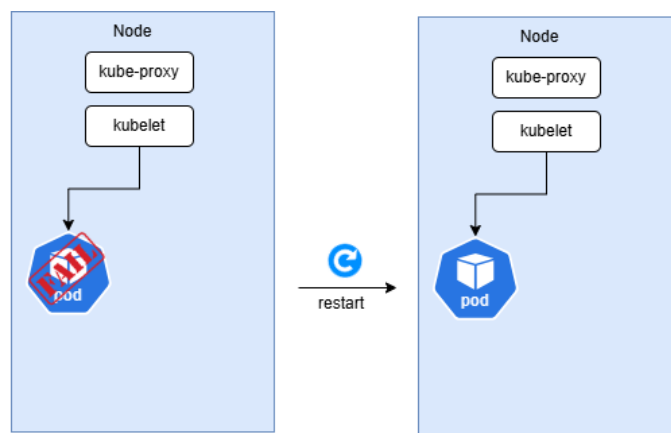


Figure 5.2: Pod failure

5.2.3 Services

As we explained in previous chapters, communication is crucial within a Kubernetes cluster and for Slurm components. Unfortunately, communication using pod IPs in a Kubernetes cluster is not considered reliable because these IPs are ephemeral. In case of a failure, the IP changes, interrupting communication and causing Slurm to become unavailable. To solve this issue, we implemented two types of services.

1. The first is a ClusterIP service, which is used for internal communication within the Kubernetes cluster, especially between the Slurmctld and Slurmdbd pods. This service is accessed using the service name specified in the Slurm configuration file.
2. The second type is a LoadBalancer service, which is used for external communication with the Slurmd daemons. It provides a fixed IP address thanks to MetalLB, used in the Slurmd daemon configuration to connect to Slurmctld.

Even if containers, pods, or nodes fail, the services remain stable, ensuring connectivity both within and outside the cluster.

5.2.4 Persistent Storage

Data persistence for Slurm is crucial, and a well-implemented storage persistence also contributes to its high availability. As explained before, when a pod is deleted or restarted, it loses its data, which can be detrimental to Slurm's data management. To address this issue, we first implemented Persistent Volume Claims (PVCs) and Persistent Volumes (PVs) using the local storage of the node where the pods are running. However, if the node is deleted and the pods are rescheduled to another node, they will no longer be connected to the original node, resulting in data loss. To overcome this challenge, we deployed an NFS server outside of our cluster, and all worker nodes are connected to this NFS server. During node configuration, we mounted the paths that we wanted to persist. Therefore, even in case of pod or node failures, we will still have access to the data. It is important to note that an NFS server is already an integral part of a Slurm cluster. Therefore, it is either already set up with redundancy, preventing a single point of failure (SPOF, which means if a system fails, all the system will stop working), or a single point of failure itself. In either case, this setup is sufficient for our needs.

With the combination of all these mechanisms, we are confident that our strategy ensures the high availability of Slurm Workload Manager. This approach guarantees the continuous availability of Slurmctld and Slurmdbd within and outside the Kubernetes cluster.

Chapter 6

Result and Analysis

In this chapter, we will analyze the results of our solution to ensure high availability for Slurm using Kubernetes. To achieve this, we will conduct test scenarios and collect metrics to evaluate the robustness of our solution. The goal is to confirm that Kubernetes provides high availability and to ensure that Slurmd and Slurmdbd are still available for users despite failures.

6.1 Test Scenarios

In this section, we will present the different test scenarios designed to evaluate the high availability of our solution. Each scenario has been repeated 5 times. These tests will simulate various failure conditions, such as container failures, pod crashes, node failures, and node deletion to test data persistence.

6.1.1 Metrics

To evaluate the high availability of our solutions, we used several metrics defined below and shown in Figure 6.1

- **Reaction time:** This is the time between the moment of failure and when Kubernetes detects the failure.
- **Fixing time** Is the time between detecting failure by Kubernetes and when the pod is running again.
- **Restart time** Is the time between the fixing time and when Slurm's state is **idle**, which indicates that Slurm has reestablished its connections with the Slurmd on the bare metal nodes, but it was already responding to requests before that, and the "IDLE" status will arrive at different times for all compute nodes in the cluster.

- **Outage time:** is the time between failure and when Slurm's state is **idle**.

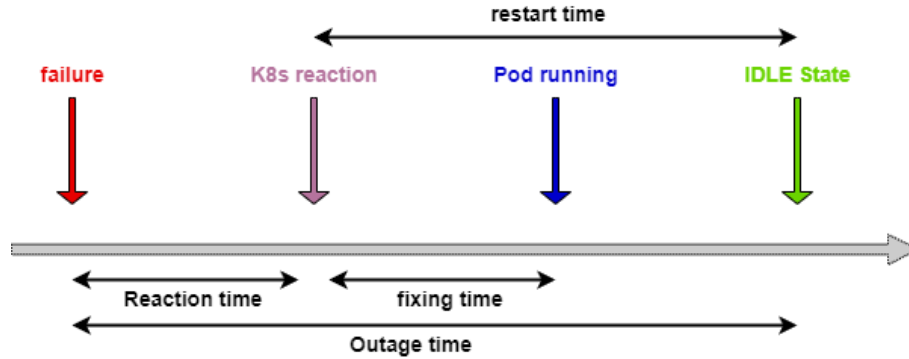


Figure 6.1: High availability metrics for failure scenarios

6.1.2 Slurmctld container failure

In this section, we will simulate the first test scenario by killing Slurmctld container.

1. Experiment:

In this scenario, the failure occurs when the Slurmctld container is terminated. As previously discussed, the component responsible for detecting node failures in the Kubernetes cluster is the "Kubelet." When the container is terminated, the Kubelet detects the failure and reports it to the master node. Consequently, the pod hosting the container enters a state where it is unresponsive to new requests. At this point (referred to as the "fixing time"), the Kubelet restarts the Slurmctld container. Once the container is restarted, the Kubelet updates the pod's state, making it ready to receive new requests again (Restart time).

2. Results:

During the experiment, we measured time in seconds for each metric, as shown in Figure 6.2. We observed an outage time of 5 seconds. Before the failure, we executed the "**sinfo**" command to retrieve the current state of Slurm. In this scenario, throughout the entire outage time, Slurm remained available. However, its state briefly changed from "idle" to "unknown" for one second, specifically between the "**Pod running**" and "**IDLE state**". We focused on the "idle" state as it represents the optimal state for Slurm.

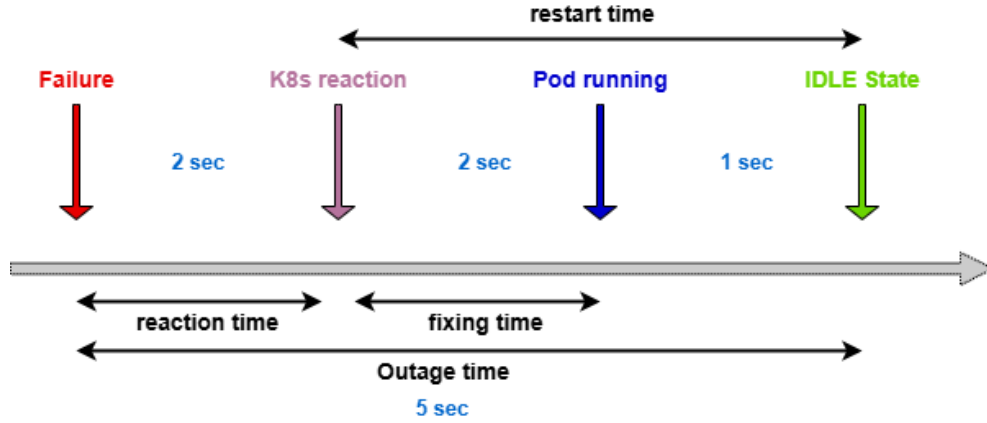


Figure 6.2: Container metrics failure

6.1.3 Slurmctld Pod failure

In this section, we will simulate the second test scenario by killing the Pod container.

1. Experiment:

Within a Kubernetes cluster, when the containers are deployed within a pod, an additional container named "the pod container" is created. This additional container is susceptible to crashing. In this scenario, we will simulate a pod failure by killing the pod container. When the pod container is killed, the Kubelet detects the failure, this is marked as the reaction time. Subsequently, a new Slurmctld pod is created, and its associated Slurmctld container is restarted. At this moment, the Kubelet updates the state of the pod, marking it as ready to receive new requests, which signifies the restart time.

2. **Results:** For this scenario, we followed the same procedure as in the previous scenario for measuring the times and the status of Slurm. In this experiment, we observed an outage time of 5.5 seconds. After the fixing time, we recorded a downtime of 1.5 seconds as shown in Figure 6.3.

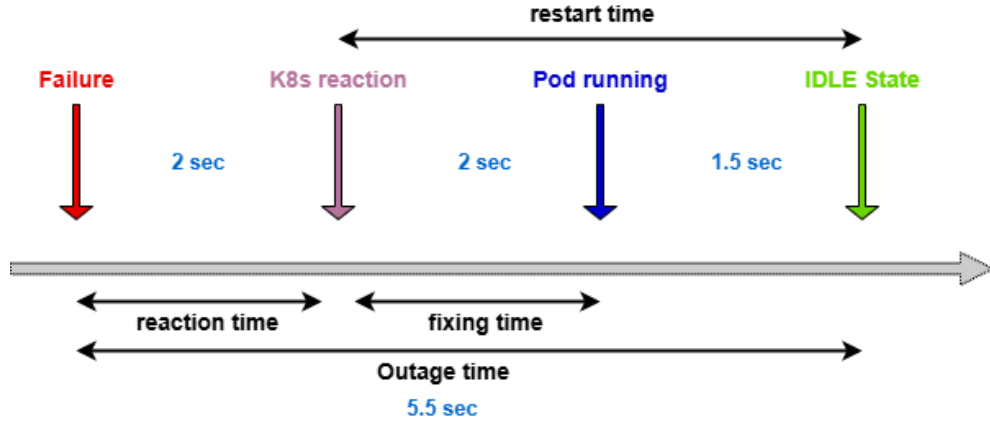


Figure 6.3: Pod metrics failure

6.1.4 Node failure

In this section, we will simulate the third test scenario by rebooting the worker node.

1. Experiment:

In this scenario, we simulate a node failure by rebooting the node hosting the pod. To achieve this in our Docker-based cluster (kind cluster), we execute the "docker restart" command. The node controller of the Master is responsible for detecting this failure. As the node hosting the Slurmctld pod is on reboot, it stops communication with the Master node and is marked as "not ready," this is the reaction time. When the node becomes "not ready," the Slurmctld pod is terminated, and a new one is created on a ready node, this is the fixing time. The restart time is marked when the pod is up and running again, able to receive new requests on the new node, and when Slurm's state is **idle**.

2. Results:

Following the same procedure as in the previous scenarios for measurement, after executing the docker restart command, the node's status changed to "Not ready" after 3 seconds (reaction time). Subsequently, the Slurmctld pod transitioned from running to complete, awaiting 3 seconds for the node to change its status to "Ready". Consequently, we observed a downtime of Slurm for 2.5 seconds. Thus, for this experiment, we recorded an outage time of 8.5 seconds as shown in Figure 6.4.

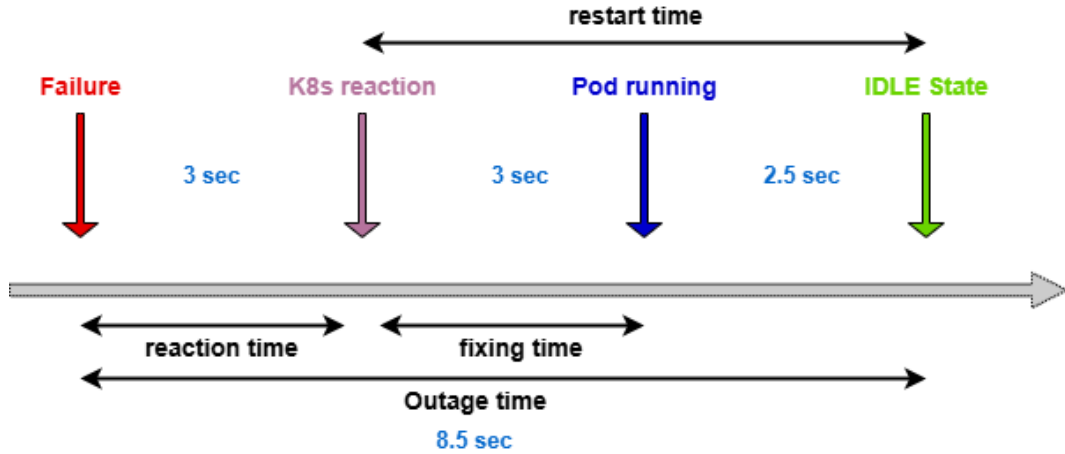


Figure 6.4: Node metrics failure

6.1.5 Extra Measurements

After repeating the experiment for each scenario five times (The first scenario occurs when the pod has not been restarted), we obtained different results for each scenario. First, as shown in Figure 6.5 and Figure 6.6, which show the failures for the container and the pod, we noticed that the time increases with each new restart. These figures show the evolution of outage time relative to the number of restarts of the same pod. We observe that outage time significantly increases as the number of pod restarts rises, whether in the scenario of a container failure or a pod container failure.

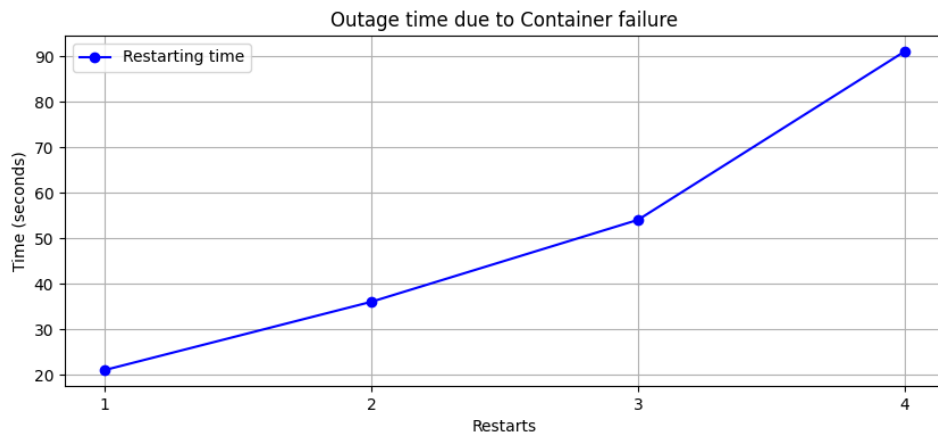


Figure 6.5: Container failures

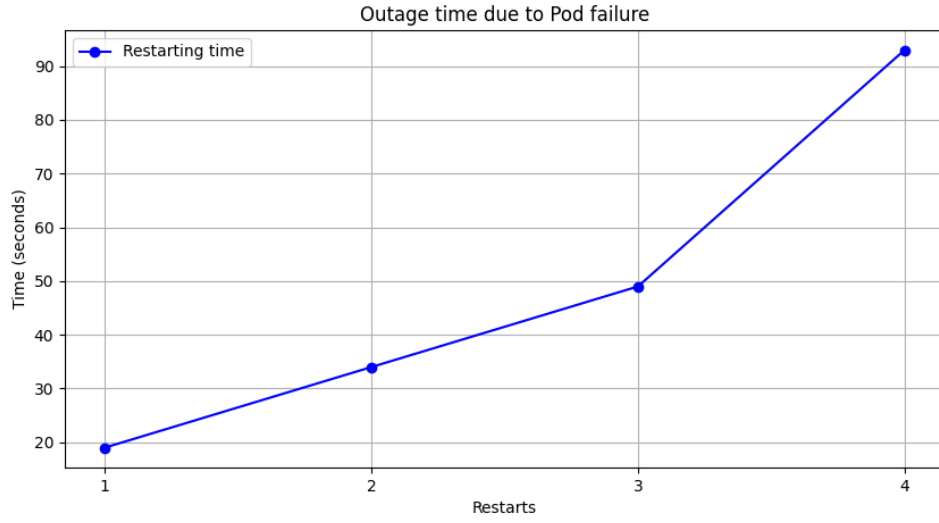


Figure 6.6: Pod failures

The same is observed for node failure in Figure 6.7, where the time increased up to 31 seconds after a single restart.

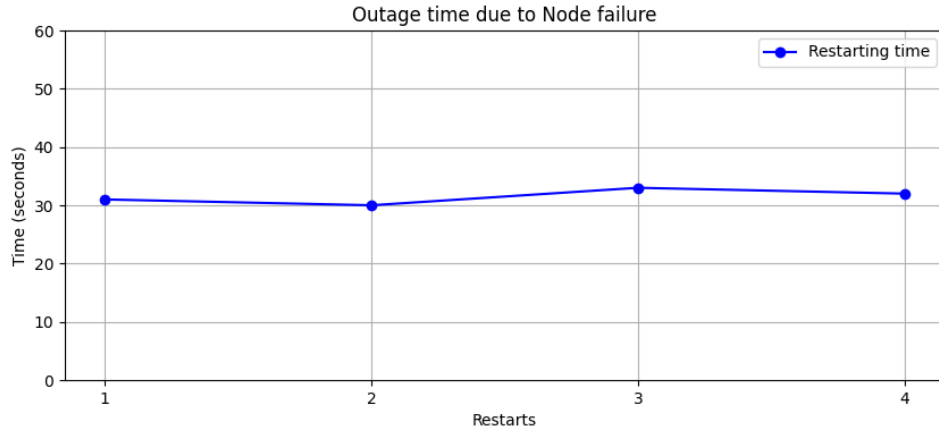


Figure 6.7: Node failures

We have calculated the average of repeated experiences 5 times for each metric and summarized the results in table 6.1. We observe that the average reaction time and restart time remain nearly the same as the first experience. However, the average repair time is significantly higher than the first repair time.

Component failure	Reaction time	Repair time	Restart time	Outage time
Container	3 sec	35.5 sec	3.8 sec	42 sec
Pod	2 sec	33.8 sec	4.4 sec	40.2 sec
Node	14 sec	5.33 sec	8.33 sec	31.5 sec

Table 6.1: Component Details

In conclusion, it would be beneficial to limit the number of restarts for each pod. Once a pod reaches a predefined number of restarts, it should be deleted and redeployed, as deleting and redeploying a pod takes the same amount of time as observed in the first pod failure experiment Figure 6.3.

Chapter 7

Conclusion

In this master's thesis, we studied, prototyped, and analyzed high-availability solutions for an HPC cluster, especially for the workload manager Slurm using the Kubernetes container management framework. This master's thesis has shown that it is possible to integrate Slurm into a Kubernetes cluster to ensure high availability.

First, we explored various integration approaches and chose a hybrid solution combining the two approaches, "Under" which aims to run Slurm within a Kubernetes cluster, and "Adjacent" which aims to run Slurm and Kubernetes in the same level by installing a Kubernetes scheduling plugin for Slurm. This allowed us to successfully integrate the essential Slurm components, Slurmctld and Slurmdbd, into the Kubernetes cluster.

Second, we created and configured the cluster, integrated the Slurm components, and managed the networking aspects for smooth communication and storage persistence. We then implemented a high availability strategy to ensure that Slurm services remain available even in case of failures.

Third, our tests in different failure scenarios confirmed the strength of our solution. The results show that Kubernetes provides an effective platform for ensuring the high availability of Slurm. This solution will allow CISM to reuse decommissioned servers from their existing infrastructure and convert them into Kubernetes nodes.

However, while our solution ensures functional high availability, there are ways to make it better. For example, using Infrastructure as Code (IaC) to automate the entire cluster setup, including application deployment, could simplify manual

configuration and updates. Additionally, integrating a monitoring tool would be beneficial for tracking performance and detecting issues. This would enable alerts for excessive restarts or crashes, and allow automatic corrective actions using scripts or cronjobs.

In summary, our master thesis proves that using Kubernetes to ensure the high availability of Slurm is an advantageous solution. Kubernetes provides a free and automatic high availability and will offer financial benefits for CISM by allowing the reuse of old machines to build a Kubernetes cluster.

I hope this master's thesis will be beneficial for CISM as it has been very fruitful for me.

Bibliography

- [1] CISM, “About the cism.” https://www.cism.ucl.ac.be/doc/_contents/About/index.html.
- [2] SCHEDMD, “Slurm components.” <https://slurm.schedmd.com/overview.html>, 2021.
- [3] S. L. S. Lixin Shen a, Tong Liu a, “Ha-oscar cluster hardware architecture.” <https://www.sciencedirect.com/science/article/pii/S0167739X04001931#fig2>.
- [4] Kubernetes, “Volumes|kubernetes.” <https://kubernetes.io/docs/concepts/storage/volumes/>.
- [5] T. Wickberg, “Slurm and/or vs kubernetes.” <https://slurm.schedmd.com/SC23/Slurm-and-or-vs-Kubernetes.pdf>, 2023.
- [6] DELL, “Omnia open source software.” <https://www.delltechnologies.com/asset/en-nz/products/ready-solutions/technical-support/omnia-solution-overview.pdf>, 2023.
- [7] “Coreweave.” <https://www.coreweave.com/>.
- [8] StackHPC, “About stackhpc.” <https://www.stackhpc.com/pages/about.html>.
- [9] S. B. William Tripp, “Exploring slurm on kubernetes cluster.” <https://www.stackhpc.com/slurm-k8s-cluster.html>, 2023.
- [10] R. Author, “Rook.” <https://rook.io/docs/nfs/v1.7/>, 2022.
- [11] METALLB, “Metalb.” <https://metallb.universe.tf/>, 2021.
- [12] G. Torres, “slurm-docker-cluster.” <https://github.com/giovtorres/slurm-docker-cluster/blob/master/Dockerfile>, 2023.

- [13] T. K. Authors, “Kind initial design.” <https://kind.sigs.k8s.io/docs/design/initial/>, 2023.
- [14] Asible. <https://www.ansible.com/>.

Appendix A

Slurm Docker image

A.1 Prerequisites

1. Install Docker
2. clone the git repo <https://github.com/badreddine2/Slurm-k8s-cluster>

A.2 Build Slurm image

To build the Slurmctld, slurmdbd, or even slurmd image, you need a Dockerfile containing all the commands necessary for constructing this image. Accessing the Slurm-K8s-cluster/conf repository, you'll find the Dockerfile and the Slurm and slurmdbd configurations. The variables with ARG are arguments of the Dockerfile. In this case, the first argument "SLURM_TAG" specifies which version of Slurm to install for our image, and the second argument specifies the version of the gosu command. At the end of the file, especially in the COPY section, Docker will copy the Slurm and slurmdbd config files and place them in the image at the specified path. Before the build, ensure that you have modified both configs to suit your needs.

To build the image, you need to execute the command:

```
docker build -t <image_name>:<tag> .
```

In my case, it was:

```
docker build -t badreddine970/slurmctld-prod:latest .
```

If you have space on Docker Hub, you can push your image using the following command:

```
docker push <image_name>:<tag>
```

Appendix B

Ansible playbooks

B.1 Prerequisites

1. Make sure to install **Ansible**
2. Make sure to install **Kubernetes** <https://kubernetes.io/releases/download/> and **Kind** <https://kind.sigs.k8s.io/docs/user/quick-start/>
3. Clone <https://github.com/badreddine2/Slurm-k8s-cluster/tree/main>

B.2 Create Kind Cluster

Once kind and Kubernetes are installed, you can create a multi-node cluster and deploy Kubernetes-Slurm component such as Pods, Services, Persistent volumes. By executing this command:

```
ansible-playbook -i /path/inventory/hosts /deployment/deploy_cluster.yml
```

You can also create a kind cluster by executing the following command :

`kind create cluster -config kind.yaml`, make sure that `kind.yaml` is available. We can specify the number of worker nodes and master nodes on this file.

B.3 Deploy Cluster

If the Kind cluster is already created, you can use `deploy.yaml` playbook which will deploy only the Kubernetes-slurm cluster component

```
ansible-playbook -i /path/inventory/hosts /deployment/deploy.yml
```

B.4 Delete Cluster

If you want to delete all components, the `delete_all.yml` playbook can be used

```
ansible-playbook -i /path/inventory/hosts /deployment/delete_all.yml
```

UNIVERSITÉ CATHOLIQUE DE LOUVAIN

École polytechnique de Louvain

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | www.uclouvain.be/epl