DPR Cluster: An Automated Framework for Deploying Resilient Stateful Cloud Microservices

by

Nikola Raicevic

B.S. Computer Science and Engineering, Mathematics
Massachusetts Institute of Technology, 2021

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2022

Author .....
Department of Electrical Engineering and Computer Science
August 5, 2022

Certified by .....
Samuel Madden
College of Computing Distinguished Professor of Computing
Thesis Supervisor

Accepted by .....
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

DPR Cluster: An Automated Framework for Deploying Resilient Stateful Cloud Microservices

by

Nikola Raicevic

Submitted to the Department of Electrical Engineering and Computer Science
on August 5, 2022, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Recent advances in distributed recovery protocols enable application builders to achieve strong prefix recovery guarantees in distributed systems of cache-stores (pairs of fast cache backed with persistent storage to answer storage requests) with low overhead. Specifically, Distributed Prefix Recovery (DPR) is a general-purpose protocol that implements prefix recovery guarantee for an arbitrary cluster of cache-stores with the help of a centralized management node. However, deploying such a cluster is still challenging, as it involves timely detection and restart of failed nodes, incremental roll-out of new cache-store implementations and deployments, and routing requests in a dynamic cluster with failures. Cluster administrators must manually configure DPR with this information and program cache-stores with the necessary capabilities in a fault-tolerant manner. In this thesis, we introduce the DPR cluster – an automated framework for quickly and easily deploying clusters of DPR-enhanced cache-stores. DPR Cluster utilizes Kubernetes as its cluster manager and features a declarative Python management API for scripting. Cluster administrators merely specify the desired cluster, and Kubernetes automatically deploys and manages the relevant components and restarts them on failure. Clients can dynamically discover a cluster and its components and communicate with them with DPR Cluster's dynamic, fault-tolerant networking layer based on DNS. Additionally, DPR Cluster implements a suite of functionalities for fault-tolerance in addition to cache-store consistency, such as automatic reconnects. Our evaluation shows that DPR Cluster is highly resilient and functional with a simple API, and significantly lowers the barrier of entry for DPR deployments.

Thesis Supervisor: Samuel Madden
Title: College of Computing Distinguished Professor of Computing

Acknowledgments

First, I want to thank Tianyu Li, without whom this thesis would not be possible. Thank you for all the feedback and guidance provided, as well as for extreme patience during my lazy slumps from the beginning of the project. Your help and insights were appreciated more than you know.

I also want to thank my advisor, Professor Samuel Madden, for his support and feedback as well as accepting to be my advisor and connecting me with Tianyu.

I want to thank my parents, Natasa and Srdjan, for all their support, unconditional help at every step and for always tirelessly working for my benefit. Last but not least, I want to thank my brother Ivan, for always being there for me and having my back, no matter the situation.

Finally, I want to take the chance to dedicate this thesis to my mom, without whom it would most definitely not exist, for many reasons.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Modern cloud storage solutions, such as Azure Storage [1] and Amazon S3 [20], work with data distributed across multiple machines. Consequently, accessing data is much more expensive compared to local storage. Modern applications often deploy a layer of distributed caches in front of cloud storage to mitigate this increased latency [11]. Caching allows for data access at local/in-memory latency, and applications asynchronously flush cached content to cloud storage for persistence. Unfortunately, caches lose data on failure; coupled with asynchronous persistence to cloud storage, this results in inconsistency in the recovered state as the portion of state stored on failed caches can roll back to an unspecified previous state.

Distributed Prefix Recovery (DPR) [18] is a novel, distributed fault-tolerance scheme proposed by Li et al. that addresses this problem. DPR provides a protocol for consistent commits of a set of cache-store pairs. More concretely, DPR offers *prefix recoverability* [19] – for each client issuing operations to DPR-supported cache-stores in a sequential or linearizable session, DPR guarantees that any recovered state corresponds to a valid prefix of the session. DPR automatically orchestrates participants to roll back to such a prefix on a cache failure. DPR assumes a failure-restart system model [18] where failed nodes undergo well-behaved restarts to restore persisted state and notify the rest of the cluster of the event.

Unfortunately, in reality, an external cluster manager needs to detect failures and launch new replacement instances to assume the identities of failed nodes. This pro-

cess may not be perfect and could lead to multiple instances competing to assume the identity of the same underlying cache-store. Additionally, such restarts and failures may change the network identity of cache-stores and impact connectivity during failure restart. For example, a restart instance may relocate to a different IP location, and existing clients may experience a connection failure. Finally, clients to DPR cache stores must be able to maintain an up-to-date and dynamic view of the changing cluster and correctly dispatch requests.

In this work, we propose an automated management suite called **DPR Cluster** to simplify these challenges of DPR. In DPR Cluster, users specify the cache-store implementations to deploy, and DPR Cluster automatically deploys a cluster with the necessary infrastructure to provide DPR guarantees across the cache-stores. DPR Cluster also provides an automated networking management layer to ensure resilient communication to restarting and dynamically relocating nodes, tackling the issue of IP change due to relocation. For clients, DPR Cluster provides service discovery capabilities, so they can discover and communicate with previously unknown DPR cache-stores at run time which is something they were unable to do. Underneath the hood, DPR Cluster utilizes Kubernetes [8]. We design and wrap all DPR Cluster components into proper Kubernetes objects and utilize Kubernetes' intra-cluster DNS network [7] to allow the components to communicate. We then utilize Kubernetes to detect failures and restart the cluster components. Each cluster component has its own persistent storage that it can seamlessly use and its own DNS address through which it can be contacted inside the DPR Cluster. Last but not least, DPR Cluster features a Python-based management SDK for clients to easily deploy clusters.

The rest of this thesis presents our detailed design and evaluation of the DPR Cluster. In Chapter 2, we provide the necessary background on the DPR paper and Kubernetes, allowing readers to follow the thesis. Chapter 3 overviews the initial system state and what we have achieved. In Chapter 4, we discuss why each cluster component is represented the way it is inside Kubernetes. Chapter 5 explains how we achieve DNS communication between the cluster components and how a client communicates with the DPR Cluster. In Chapter 6, we explain how exactly automatic

deployment is achieved. Chapter 7 describes how we achieved fault tolerance inside the cluster. Finally, in Chapter 8, we present some experiments that we have run that confirm the fault tolerance and stability of the DPR Cluster, while Chapters 9 consists of a conclusion and future work.

Chapter 2

Background

We now discuss background information on DPR, its motivation, technical details, and some hidden assumptions that make deployment non-trivial. We then introduce Kubernetes – the centerpiece of our solution to automated deployment of DPR, with particular emphasis on concepts and tools we rely on.

2.1 DPR Overview

On a high-level, a DPR system consists of a centralized coordinator node (**DPR Finder**) and a set of stateful servers (**workers**). Each worker encapsulates some application state backed by storage but cached in memory, forming a *cache-store*. Note that a DPR system may consist of different types of workers exposing different client interfaces. DPR Finder is the system coordinator, and it is responsible for keeping track of the state of the cluster and rolling back the workers to a safe version when they fail.

The DPR guarantee [18] is defined in terms of sequential (with generalization to linearizable) client sessions issuing operations to various cache-stores. Such operations must be either completed or not started in any recoverable state (i.e., atomically recoverable). Then, we can define a DPR-consistent recoverable state of a set of cache-stores as prefixes of client sessions. Given any client session, the recovered state corresponds to a prefix of its issued operations, including the effect of all operations in

the prefix and excluding the effect of all others. Such prefix is incrementally computed and made known to participants and clients, with the promise that the system recovers to at least the promised prefix in response to any worker failure. Operations in this prefix (called the DPR cut) can therefore be considered *committed*. Meanwhile, operations not in the prefix are merely completed with its effects in the cache, and could potentially be rolled back if the relevant cache failed. DPR automatically orchestrates a rollback to a consistent prefix in such an event and sends an explicit signal to client sessions about the rollback.

Just like any worker, it can also happen that the DPR Finder fails. For this reason, DPR Finder also needs a way to persist storage. Persisting storage is achieved using a **PingPong Device** [4]. This storage consists of two file versions, front and back. We can write whatever we want in these files, in any format we want. It is up to the user of the PingPong Device to write inside the files and assemble information from what they read from the files. Hence, they can be used to store anything, but DPR Finder uses them to store the current distributed prefix among the workers. The back file represents the most current version of the data. When we want to write something new, it is written atomically to the front file. When writing whatever the user specified to the PingPong Device, it also attaches a current version number of the data. If the write happens successfully, we atomically switch which file is the front and which one is the back, once again letting the back one be the correct, newest version. If a failure occurs during the write, the back file will remain unchanged and still represent the latest complete write. When the DPR Finder starts up for the first time, it forms these files and the PingPong Device. Every time the DPR Finder starts up after a failure, it sees that the files already exist and thus knows it has restarted. It then reads the versions it has been writing from both files and sets the file with the larger version as the back file since it is newer than the other file. It then proceeds to load the necessary information from the back file.

Apart from storing regular data, which is handled by the worker implementation, the workers also need to store additional information about the data they have committed. Each worker asynchronously commits data and remembers each commit it

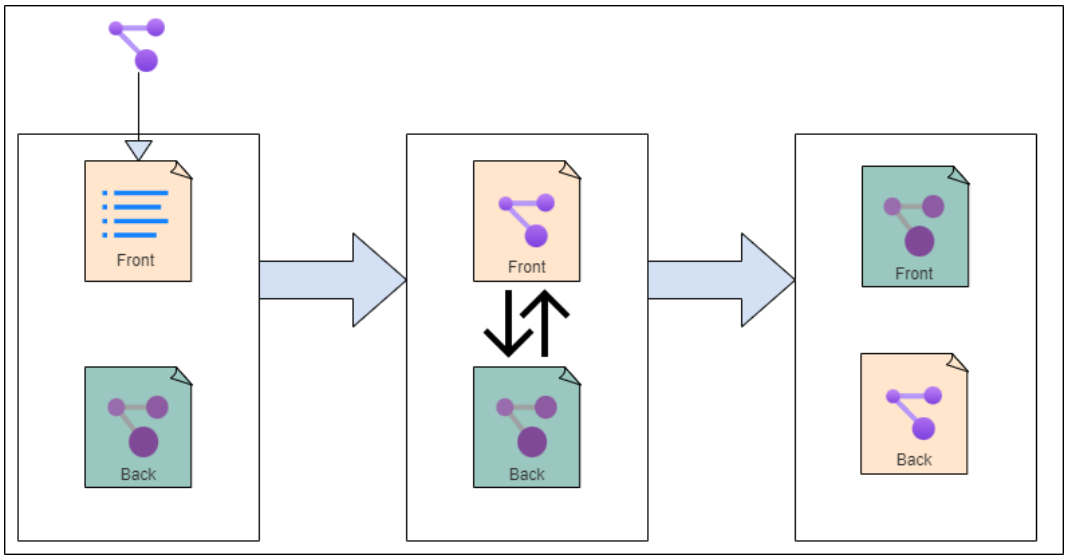

Figure 2-1: A visual example of writing to the PingPong Device. A new distributed prefix is first written to the front file while the back file is safe. Once done, we atomically swap the files. Finally, the front file with the newest version becomes the back file and safely keeps the newest prefix.

made. It then informs the DPR Finder about the commits that it has made. Once the DPR Finder incorporates a worker's commit into its distributed prefix, the worker can forget all of the commits before that one since it will never recover back to them again. However, it still needs to remember every commit after the one in the distributed prefix since those are yet to be committed to the DPR Finder. Workers do not need anything as complicated as a PingPong Device – they can atomically write down their commits. If a write fails, the commit will be treated as if it did not happen.

Finally, each worker has their own globally unique ID for communicating [4] with the DPR Finder. The distributed prefix that the DPR Finder keeps track of is then mapping each worker's ID to their recent commit. We do not go into detail on how these IDs are assigned to the workers at the moment and tackle that later in the thesis. We summarize the capabilities of the DPR Finder below:

- NewWorker() — When a worker starts up, the first thing it does is send this type of request to the DPR Finder while also including its ID. When the DPR Finder receives such a request, it checks whether it is already mapping

the specified ID to some commit in the distributed prefix. If it does not, it simply adds the ID to the mapping, maps it to a dummy commit, and sends an ACK back to the worker confirming that it was successfully added. However, if the ID is already in the distributed prefix, the DPR Finder knows that the worker had to have failed and then restarted. The DPR Finder then rolls back the workers inside the system to their commits from the distributed prefix. We elaborate on how this is done in other operations.

- DeleteWorker() — Similarly to NewWorker(), this method also sends the worker ID to the DPR Finder and it tells the DPR Finder that the worker with the specified ID is leaving the cluster. The DPR Finder removes the worker's ID from the mapping and any other info it keeps about the system and sends an ACK to the worker signifying that it can freely leave the cluster.

- Refresh() — This is a generic message that any worker can send to the DPR Finder and does not require identification. As explained, the DPR Finder always keeps the current distributed prefix. Along with the prefix, the DPR Finder also has a precomputed answer for communicating that prefix upon request. When it receives a Refresh() command the DPR Finder responds with its precomputed distributed prefix response. Each time the DPR Finder modifies the distributed prefix, it also recomputes its response.

- NewCheckpoint() — This method is sent from the worker with its newest commit version. Hence, when the worker successfully commits something, it uses this method to let the DPR Finder know it has done so. In return, the DPR Finder takes note of this and updates its distributed prefix and precomputed response. Refresh(). Once this happens, the DPR Finder sends the worker an ACK. If the ACK does not get through or the DPR Finder fails before it even sends an ACK, the worker will not consider the commit correctly sent to the DPR Finder and will try to send it again later.

- ResendGraph() — After the DPR Finder fails and recovers it lets all the

workers know. Before the DPR Finder, and the cluster as a whole, can recover each worker needs to send the ResendGraph() request to the DPR Finder. While the DPR Finder is down, each worker keeps forming new checkpoints that it cannot send to the DPR Finder. As part of the ResendGraph(), the worker first sends all the stalled NewCheckpoint() requests to the DPR Finder, followed by the ResendGraph() message itself, signaling that the worker has sent all the information necessary for recovery to the DPR Finder. Please pay attention to this request since it is sent out when the cluster needs to recover and most bugs arise around it. We have provided a simplistic view of the request but sufficient for understanding the thesis.

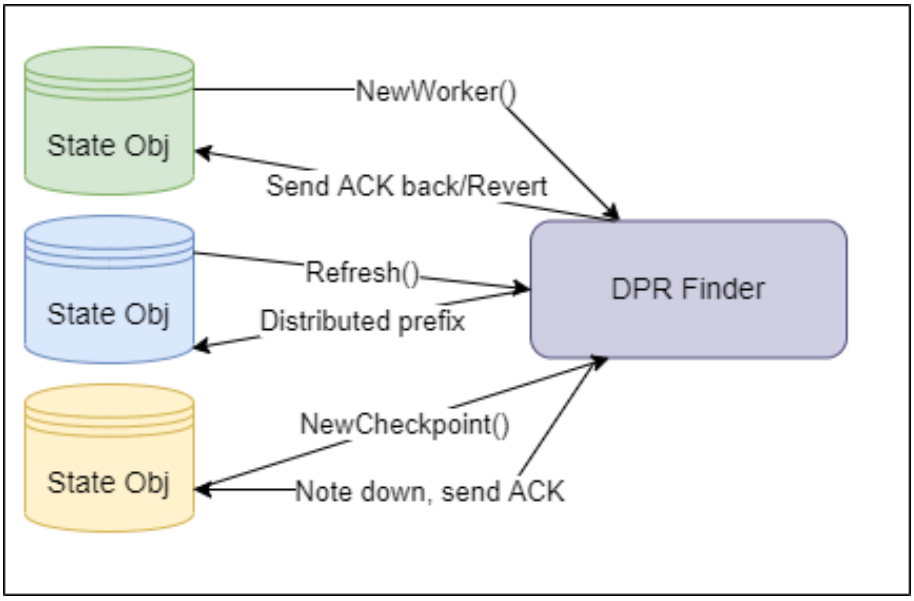

Figure 2-2: A simple example of the DPR System. Each State Object on the left is one of the workers and each of them is sending one of the relevant operations we described to the DPR Finder.

2.2 Kubernetes Overview

Kubernetes is a popular open-source framework for automating containerized applications' deployment, scaling, and management. A container is a lightweight version of a virtual machine that isolates an application's environment. Systems such as Docker

[3] combine the application environment and the executable into a container image for easy, plug-and-play deployment to various environments. Such containers, although powerful and modular, are often challenging to manage in the face of failures and the complex, multi-container environment of modern cloud applications. Kubernetes addresses these issues by providing a relatively easy way to specify container workloads to run and a set of machines (Kubernetes cluster) to run them on; Kubernetes then automatically deploys these containers onto the machines and provides a way to utilize intra-cluster communication [7].

For users, Kubernetes exposes several specialized objects, which we now describe. Note that we are focusing on the specification and usage of these objects rather than the implementation.

2.2.1 Pods

The basic building block of Kubernetes is a **pod** [10]. A pod consists of one or more containers acting and being controlled as a single application. They share the environment, the networking layer and the filesystem. Even though it can run multiple containers together, a pod usually runs a single container and we are only running single containers inside pods in this project. The best way to conceptually think about a pod is simply an application running on its own on a separate mini-machine as part of a larger machine, somewhat similar to a thread.

A pod is simply running a container and therefore is not much different from it. Kubernetes does not handle failures of pods automatically, neither if code inside them fails nor if the machine it put them on goes down. For those purposes, Kubernetes include more complex objects whose only job is to manage and scale pods. These are called pod controllers [2] and we will be going over two of them: ReplicaSet and StatefulSet.

We communicate pods and all other objects that we are about to discuss to Kubernetes utilizing YAML [17] formatted files. These files allow us to specify all the necessary properties of objects [15] we want to run. For example, a YAML file would contain a specification that we want to run a Pod, where we can find that pod's

Docker image, its name, and its resources. YAML files get more complicated with more complex objects used but the principle stays the same. We specify in the YAML file and Kubernetes executes.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: dnsutils
  namespace: default
spec:
  containers:
  - name: dnsutils
    image: k8s.gcr.io/e2e-test-images/jessie-dnsutils:1.3
    command:
      - sleep
      - "3600"
    imagePullPolicy: IfNotPresent
  restartPolicy: Always
```

Figure 2-3: A simple YAML deploying a pod. Specified basics such as the pod name, container name and the location of the container image. Of course, there are additional attributes specified, but we do not go into detail on those

2.2.2 Pod Controllers

A ReplicaSet [12], as its name suggests, is used to manage multiple replicas of the same pod. The ReplicaSet treats the pods as actual replicas of each other and does not differentiate between them in any way. While it does assign them unique names, since this is a general Kubernetes requirement, it uses the user-specified name of the ReplicaSet and adds a randomly generated suffix. In its YAML file, we specify the number of replicas we want the ReplicaSet to keep running and a pod template specifying how to generate each replica. Of course, we can still specify the resources we want the pods to have. Once ReplicaSet is started, it deploys the specified number of pods on cluster machines on its own and continues to manage them. If a pod fails, it restarts it to keep the number of pods as specified.

A StatefulSet [14] is also used to manage multiple copies of the same pod. It

is also specified similarly through YAML and takes the number of pods and their template. However, StatefulSet differentiates between the pod copies once they are formed and, in doing so, it "keeps state." Each pod is assigned a unique ID starting at 0 and ending at $n-1$ where n is the number of pods specified in the YAML. The ID is attached as the name suffix to each pod instead of a random one. The pods are also started in order rather than one at a time. By differentiating between pods, StatefulSet makes networking easier because pod names are not changing. It also allows for an easier way to utilize persistent storage. More word on this is below.

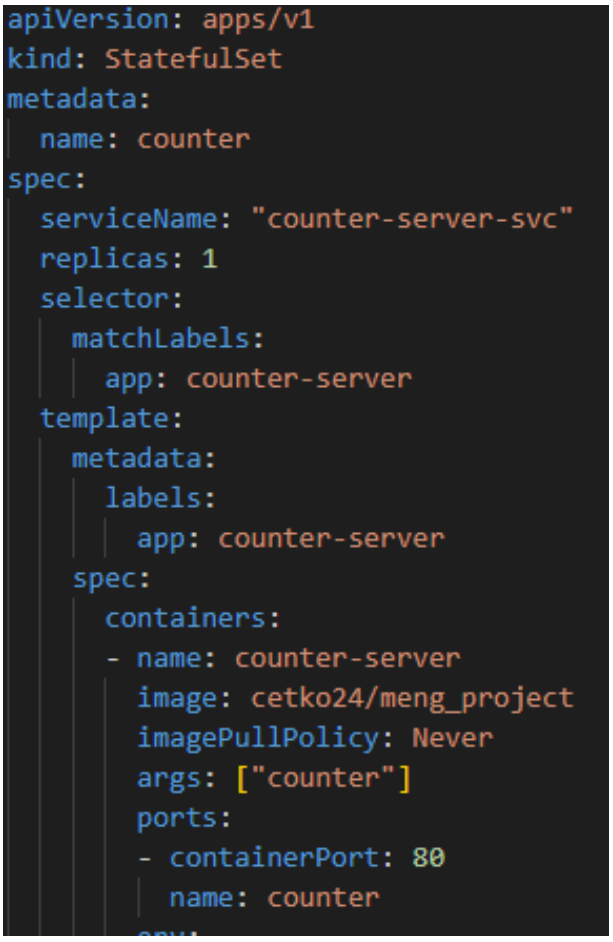

Figure 2-4: A partial YAML deploying a StatefulSet. In this example, we specify that the StatefulSet should only keep one replica at all times. The template part of the YAML is very similar to the figure above since it is also simply describing a pod. Note that we are specifying the name of the Service that represents the StatefulSet.

2.2.3 Services

Pods have their internal IPs [7] that Kubernetes assign and these IPs can be used to reach pods. However, whenever a pod restarts there is no guarantee that the IP will not change. In fact, it most probably will. To overcome this issue, Kubernetes provides us with an intra-cluster DNS network. By utilizing DNS we are trivially immune to IP address changes. To introduce DNS in a cluster, we have to utilize a Service object, the Kubernetes networking object. A Service is a representative of the pods that takes the pod's requests and forwards them to the pod. There are multiple types of Services as well, depending on the user's needs, and we present three different types of Services: regular Service, headless Service and LoadBalancer.

A Service [13] is deployed using YAML files as well, and we specify its name, the ports it is listening on and special attributes that allow it to determine to whom it should forward requests. We do not go into details on these since they are neither relevant nor complicated to set up. While there are ways for Services to point to unrelated objects, they are almost always forwarding requests to a pod controller, i.e. the pods inside it. We also only use it for those purposes.

A regular Service does not differentiate between the pods it is pointing to. When it receives a request, it tries to send it to whichever one of the pods it is forwarding to is currently available, thus maximizing the load it can handle. Because it does not differentiate between pods, this Service is usually used with ReplicaSets but that is not always the case, as will also be demonstrated in this project.

A headless Service also points to a group of pods represented by the same pod controller as a regular Service, but it differentiates between the pods. In contrast to the regular Service, the headless Service acts more like a DNS server and needs additional info about the pods to resolve queries. A regular Service is more similar to an endpoint which makes no guarantees about the pod servicing the request. Due to the ability to differentiate between pods this Service is usually used in combination with a StatefulSet.

LoadBalancer [13] is used in front of a pod controller, same as the other Services,

but it is usually used to expose the cluster to the outside world. It usually works only on cloud providers and has a persistent, public IP at which it can be contacted from the outside. It then takes those requests from the outside and forwards them to the pod, thus allowing cluster components (pods) to talk to the outside world.

2.2.4 Volumes and Persistent Storage

Finally, a natural need arises for some applications to persist storage between failures. For example, someone might be running a database inside a pod. Pod controllers are not directly in charge of providing storage; Kubernetes takes care of that. Namely, there exists a concept of a volume [16] which is simply persistent storage that exists in Kubernetes independently of any pods. How the volume operates precisely depends on the chosen implementation, but we do not focus on that.

```
        volumeMounts:
        - name: counter-storage
          mountPath: /DprCounters/data
volumeClaimTemplates:
- metadata:
    name: counter-storage
  spec:
    accessModes: [ "ReadWriteOnce" ]
    resources:
      requests:
        storage: 1Gi
```

Figure 2-5: A snippet such as this one needs to be added to a StatefulSet's (or any other pod controller) YAML file in order to signal to Kubernetes that we require persistent storage. The first part specifies the name of the claim and the directory at which it is mounted, i.e., the one whose data persists. The second part specifies the storage itself, like specifying that it requires 1Gb of data, for example.

When specifying a pod controller, we also specify a volumeClaimTemplate [14] in addition to the pod template discussed above. This is the template through which we are telling the Kubernetes to provide the pods with persistent storage. Among other things, we have to specify the amount of storage we want. We also specify a directory path where the volume will be "mounted." That means that all the data

written to that directory will be persisted and that that directory will contain the data even after failure.

Different pod controllers handle persistent storage differently. In ReplicaSets, each pod shares the same persistent storage. They share it because there is no way to differentiate between pods so a pod cannot have its own. In StatefulSets, each pod has its storage that is always associated with it. When a pod with a particular ID fails and gets restarted, it is again associated with the same storage.

Chapter 3

System Overview

There are two main parts to this chapter. The first part outlines the problems we were facing and the decision we had to make to get the DPR Cluster up and running. The second part is a high-level overview of our proposed solution and its components, each of which will be discussed in future chapters in greater detail.

On a high level, we want our end product to be a cluster consisting of cache-store nodes (workers) offering DPR guarantees. DPR Cluster is the software kit that allows for a seamless deployment of said cluster that has its own network and is fully fault tolerant while also offering a client side implementation for connecting to the cluster.

3.1 Main Challenges

We identify and present four major challenges we solve with the DPR cluster:

- **System Architecture**: There are two sets of questions that fall under system architecture. Where is the cluster hosted exactly? How does it get there? How is it managed? A short answer to these is that we are utilizing a Kubernetes Cluster and that we are utilizing Kubernetes' automatic deployment possibilities to start up our cluster. Kubernetes also manages the cluster but it is vital to make Kubernetes treat each cluster component properly. We design and decide how we will represent the workers and the DPR Finder in Kubernetes and get Kubernetes to give them everything necessary. In the System Architecture

chapter, we offer insight into the decisions we have faced and why we picked the ones we did.

- **Networking**: There are two parts here. The first part is that the workers and the DPR Finder need to be able to discover each other and communicate. We achieve this by utilizing Kubernetes' intra-cluster DNS network. However, it was still up to us to design the types of services used by the worker and the DPR Finder and, therefore, implicitly set up the DNS naming convention. The other part of networking was designing ways for the client to communicate with the cluster. As mentioned in the Introduction, the DPR paper assumed that the client knew the architecture of the cluster and the way to communicate with each worker. However, neither of these assumptions is realistic, and they need to be addressed. We discuss how we managed to expose the cluster to the public and get it to communicate with the client while also explaining why we went down that path.

- **Deployment**: Our whole cluster can be deployed manually by feeding the appropriate YAML files to Kubernetes and having it spin up our cluster. However, there are too many factors to consider and runtime information to embed in the YAML files for this process to be done by hand. Therefore, we utilize Python scripting and the Kubernetes APIs [9] to deploy whole clusters automatically. The Deployment chapter describes how we achieve scripting and coordination when spinning up the cluster and discussing the cluster administrator part of specifying the cluster.

- **Fault Tolerance**: DPR itself exists to guarantee fault tolerance, but it guarantees it in terms of consistency of the data inside the cluster. There are many other possible points of failure, primarily due to the DPR Finder's communication with the workers. Some of these included classic situations like the DPR Finder failing and the workers failing with it because they were awaiting a response. The others are implementation bugs that were causing cluster components to either continuously crash or get stuck. A significant portion of our

work was spent on the fault tolerance of the DPR Cluster. However, most of our work is too low-level and code-specific to be discussed in the thesis.

3.2 Our Solution

There are three main components of our solution:

1. The Cluster Administrator specification and Python deployment
2. The DPR Cluster Harness
3. Client-side library

The Cluster administrator utilizes a simple Python script to start up the cluster. They can then add workers and specify the worker types (what cache store implementation are they running) and the resources they require, like memory or CPU. They can also specify the DPR Finder resources if they are knowledgeable about how the DPR Cluster functions. After adding the workers, they can simply start the cluster.

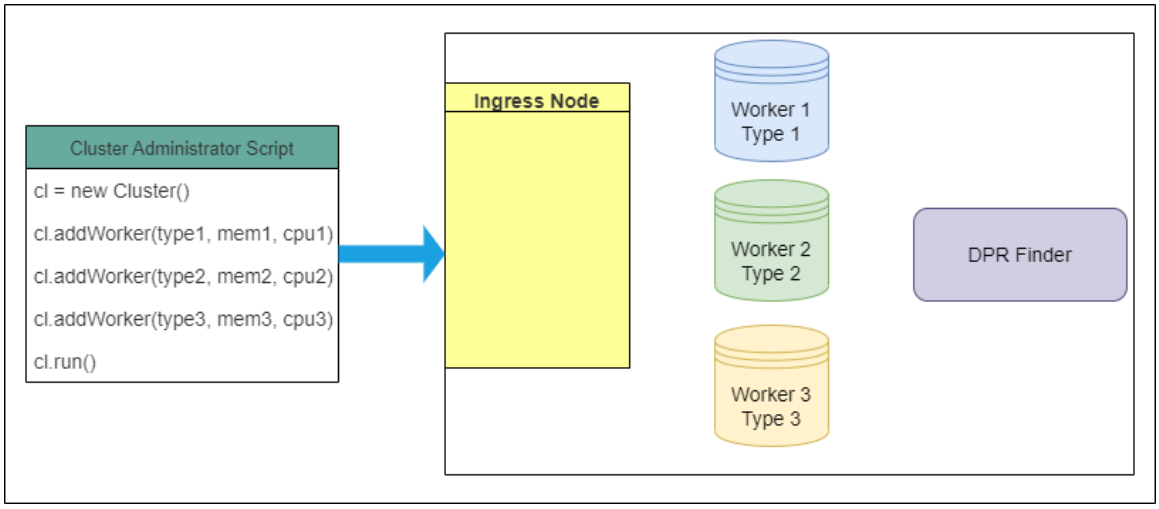

Figure 3-1: On the left, we have a script that the cluster administrator could write to deploy a cluster. The administrator specifies that they want a worker with **type1**, **mem1** memory and **cpu1** CPU and similarly for the other workers (this is a simplified). The blue arrow represents the Python Automation code that takes this script and starts workers with specified types and resources. It also implicitly starts up the DPR Finder and the Ingress Node so that we can communicate with the client.

Cluster specifications are then fed into Python code. In an automated fashion, we ensure that all the proper YAML files are built and that each worker represents the correct type and has the specified resources. Here we also ensure that all the workers are named correctly and have properly assigned IDs. Finally, this code also sets up all the external communication channels with the on its own and without cluster administrator guidance.

Inside the DPR Cluster, each worker is run inside a StatefulSet of its own, while the DPR Finder is also in a StatefulSet. We also correctly attach persistent storage to each worker and to the DPR Finder using Python automation. Surprisingly, we use a regular Service in front of both the worker and the DPR Finder StatefulSets, one per each. Through these Services, the components communicate with each other and the outside world. As the final part of the DPR Cluster, we are using an Ingress Node [5], which has a persistent public IP. In addition to this IP, the Ingress Node can be used to route its incoming requests and we use it to route clients' requests to internal DPR Cluster components. More specifically, the Ingress node would route a request based on which port it has received the request.

Before communicating with the cluster, the client is assumed to know the Ingress Node's persistent IP address. Note that this is a reasonable assumption since there must be some way to contact the cluster. In a fully productionized setting, a public DNS path would exist that resolves to the persistent IP for the clients could use.

For the client to naturally learn the cluster's architecture and how to contact the workers, we designed a FetchCluster() operation that the client issues to the DPR Cluster. In response to this request, the client would receive a message containing the type of each worker and which Ingress Node port maps to which node. Specifically, we designed the clients to contact the DPR Finder through the Ingress Node. Note that this also assumes that the clients know the port mapping to the DPR Finder but that this is also a reasonable assumption.

We piggybacked off the existing communication without adding any additional communication between the DPR Finder and the workers. We made the DPR Finder keep the worker types, ports and all other necessary information. This information

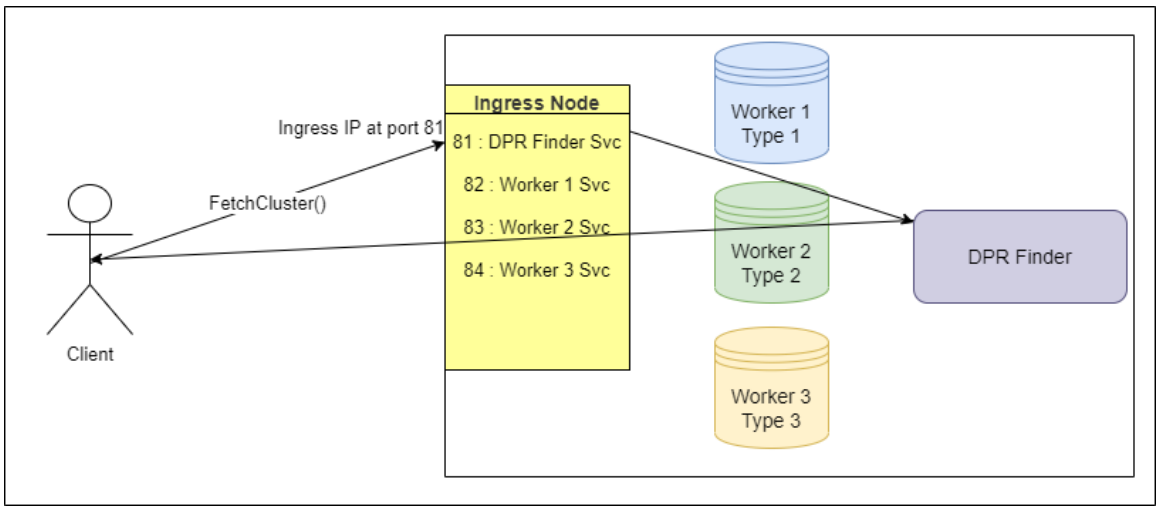

Figure 3-2: In the ingress node, we have a simplistic view of the port to DNS mapping and port 81 is being mapped to the DPR Finder's DNS. The client knows the Ingress Node's persistent IP and DPR Finder's port so it can send the FetchCluster() request to the address. The Ingress node routes the request to the DPR Finder and the DPR Finder sends over the whole cluster architecture, allowing the client to learn about the mappings that we can see inside the Ingress node (among other things).

is also persisted through DPR Finder restarts. Once queried about the cluster architecture, the DPR Finder simply sends the info it keeps to the client.

The FetchCluster() operation and everything that goes into supporting it intersect with almost every major problem area described in Chapter 3.1. Instead of dedicating a whole chapter to it, we divide it between chapters and describe each part of the implementation where appropriate.

Finally, we implemented an elementary version of a worker for all the demo-related purposes of this project and the evaluation. We call our type of worker the Counter Server, and we generally regard its type as just a **counter** worker. This worker starts with an internal counter initially set to zero and only accepts Increment() operations in which it takes an increment **inc** as an argument. In response, Counter Server implements its count by **inc**. Of course, as with any other type of worker, the Counter Server is wrapped in all the necessary DPR logic. We use the example of the Counter Server to demonstrate many of the points we try to make in future chapters when the type is required.

Chapter 4

System Architecture

After deciding to use Kubernetes as our cluster manager, the first step was deciding how to represent each of the DPR Cluster components inside Kubernetes. As mentioned in the System Overview chapter, we decided to represent the DPR Finder as a StatefulSet and each worker as a StatefulSet of its own. This chapter provides further insight into why this decision was made. We also discuss how we provided each worker and the DPR Finder with persistent storage.

4.1 Kubernetes Representation

4.1.1 DPR Finder

There is at most one distinguished DPR Finder inside the cluster at all times (0 in the case of transient failure). This fact will come in handy at multiple places in our discussion. Given that there is a single DPR Finder, a naive approach for representing it in a Kubernetes Cluster would be as a pod of its own. There are multiple bad things about this approach, each highlighting the important aspects of the DPR Finder that we must account for.

First, once a pod fails, it never comes back up which is unacceptable behavior for the DPR Finder. Additionally, the only way to contact a pod is by using the pod's internal cluster IP, which is only assigned when the pod is created. This would require

us to keep hard-coding IPs. Even if we could somehow automatically fetch the IP, there is no guarantee that it would remain unchanged, making it hard to contact the DPR Finder reliably. Finally, if a pod was to go down and we manually got it back up, all of its data would still be lost, which is another no-go for the DPR Finder.

The above shortcomings mean we must use a pod manager like a ReplicaSet or a StatefulSet. Either of these will detect when the DPR Finder pod fails and restart it automatically, addressing the first issue from above. We do not need to use any of the inherent properties of the ReplicaSet and DPR Finder needs persistent storage, both of which mean that StatefulSet is the right approach.

The StatefulSet approach gives us everything we are looking for and more. It assigns each pod a unique identifier, starting from zero. This means that the DPR Finder pod's name inside the cluster is always **dpr-finder-0**. This name also stays constant upon each restart which helps with debugging. Additionally, the StatefulSet can assign persistent storage to a specific pod, which is also precisely what we need. However, using a StatefulSet slightly complicates communication, as discussed below.

4.1.2 Workers

Tracing almost identical steps as in the DPR Finder discussion above yields the conclusion that we should also represent the workers as StatefulSets. However, it is pretty reasonable to expect DPR Clusters with only workers of a single type or at least multiple workers of each type represented inside the cluster. A question arises of whether we should represent each worker as a StatefulSet or bundle workers of the same type together and have a StatefulSet represent all of them. Although these two converge on the same thing when there is only a single type of each worker, it is worth exploring whether grouping workers has any benefits.

At first glance, grouping the workers seems much more natural and like the right choice. Grouping the workers would lead to a much cleaner name resolution. In general, we went with the straightforward approach of naming each worker after the type of the worker. For example, the base of the Counter Server's name would be **counter**. If we grouped all the Counter Servers, then the StatefulSet under which

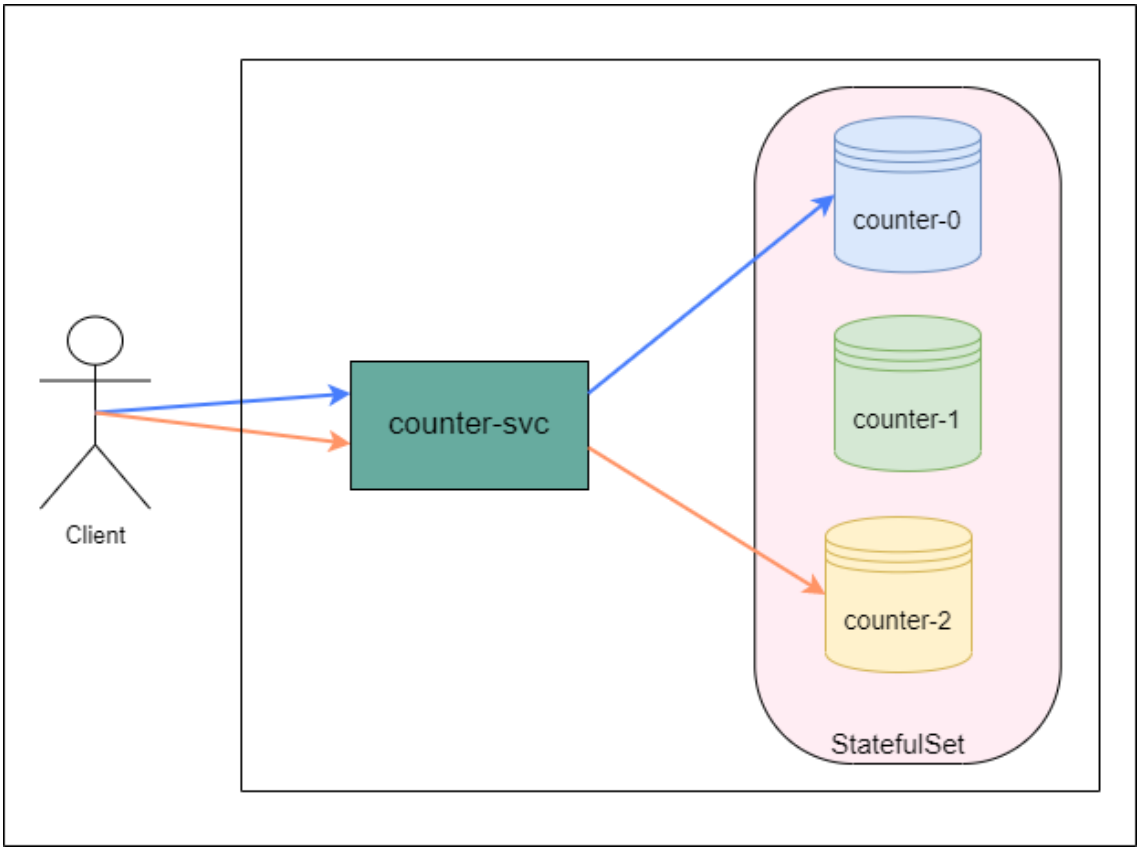

Figure 4-1: On the right, we have three pods, each a part of the same StatefulSet, under the same Service. The client contacts them through the Service, which routes the requests.

they are grouped would assign them unique IDs (in order), making it much simpler for us. For example, if we had 3 Counter Servers then the StatefulSet would name the pods **counter-0**, **counter-1**, **counter-2**. On the other hand, if each Counter Server were a StatefulSet of its own (of size 1), then each of these StatefulSets would assign the ID of zero to its pod resulting in each of the three pods being named **counter-0** which is a clear conflict in Kubernetes. This requires augmenting the names of each StatefulSet with IDs generated by us and then letting the StatefulSet attach the zero. For example, it would look something like: **counter-0-0**, **counter-1-0** and **counter-2-0** where we pick the first number.

Similarly, grouping workers would allow us to have a single Service pointing to the StatefulSet called **counter-svc** which could then be used for DNS. On the other hand, each of the Services in front of the standalone StatefulSets would need to have

its name modified to avoid clashing, similarly to the situation above. On top of this, we would be using a single Service to access a bunch of pods over having a Service per pod, which sounds more natural.

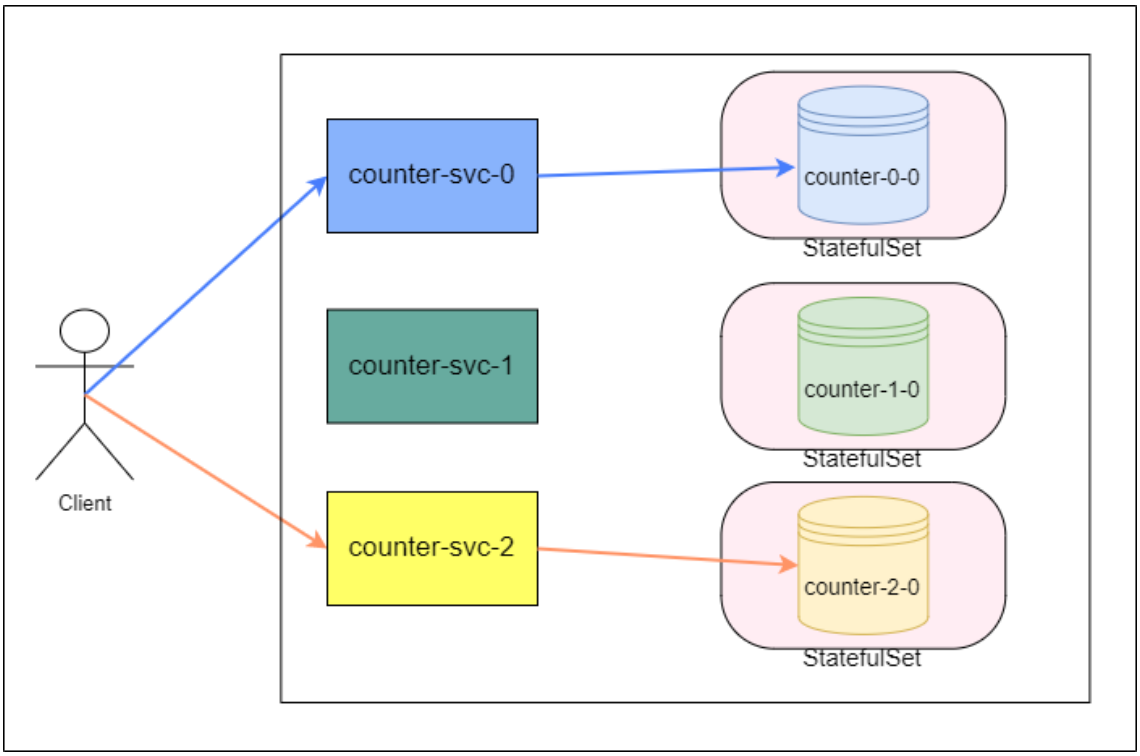

Figure 4-2: On the right, we have three pods, each in its own StatefulSet and each with its own Service. The client contacts them through their respective Services.

While it is simpler to group workers, it is not the optimal choice. First of all, a StatefulSet starts up its pods one by one. Just this fact increases the start-up time of the cluster. If one of the earlier pods fails during start-up, it worsens the effect. Sequential deployment is how StatefulSets operate and is not something we can change. Starting up each pod on its own is better from this angle.

A Service is an abstract concept introduced by Kubernetes and not an object physically running anywhere. As such, for example, it can not fail. It is essentially just a way to define DNS inside Kubernetes. Since this is the case, it does not pose a problem if we run many Services as opposed to a few. Hence, it does not matter that we would be using a single Service as opposed to many if we were not grouping. Arguably, even if Services could fail, it would still not be beneficial to use a single

one over many.

The final and most important reason why it is inconvenient to group workers is the fact that we need to be able to add and remove workers from the DPR Cluster without taking it down. When the workers are not grouped, we can instruct Kubernetes to remove (shut down) the worker's StatefulSet and, in doing so, we would remove exactly the worker we wanted. In contrast, removing a pod in the grouped setting is equivalent to removing a pod from a StatefulSet, which is not simple and there is no go-to way for doing this in Kubernetes. While it would be possible to do this, we would likely end up killing the whole StatefulSet and re-establishing it without the pod we are trying to remove. This would require us to copy the data or re-take ownership through Kubernetes and it would also lead to all of the pods with a higher ID having their ID decreased, which is highly inconvenient.

The discussion above shows that the only real advantage of grouping workers together, although it does initially sound intuitive, is added convenience in naming. However, naming is a much simpler problem than deleting a specific pod from a StatefulSet, which is why we decided not to group the workers. Implementing the StatefulSet decision would probably be a performance bottleneck and a potential source of corrupted data.

The naming problems highlighted above are real and quite inconvenient to deal with manually. However, this is where Python automation comes in. Among other things, this part of the system ensures that the name clashes above are avoided.

4.2 Persisting Storage

4.2.1 Classic Storage in DPR Cluster

Before diving deeper into how we achieve persistent storage in Kubernetes, we provide a quick overview of how pod file systems work with Kubernetes' persistent storage. The pod is essentially unaffected by whether or not it has persistent storage attached to it and is unaware of it. It simply proceeds to use its file system as it usually would.

Of course, some pods are using their file systems assuming that the files they are writing will be there if the pod fails and restarts. If we did not mount persistent storage to this folder then the application inside the pod would see the folder empty each time it starts up and this will cause a straight-out error, or it will cause the pod to think that this is the first time it has been started. We do not concern ourselves with how this data survives in persistent storage since Kubernetes guarantees that.

We now discuss how to achieve persistent storage in Kubernetes. We discuss the DPR Finder first and then expand our discussion to workers. Inside DPR Finder's YAML file, we need to specify a volumeClaim [16] which will make Kubernetes ensure that appropriate persistent storage is available to the pod whenever it is running. We also need to specify where the storage is going to be mounted. From the pod's perspective, it can always read and write things in its own file system and is not affected by persistent storage. However, only the data inside the folder on which we mounted the persistent storage will still be there upon restart, while the other data will be lost.

As explained, the DPR Finder uses PingPong Devices to keep its local state. Fortunately, both files from the device are written inside the same specified folder in the file system. This folder is specified to the DPR Finder when it is being started up and all we need to do is make sure the storage is mounted to this folder. Then, our data will be safe, as discussed in the first paragraph. We are in complete control of the DPR Finder implementation and the code starting it up. Hence, we can specify the same folder at the YAML file upon DPR Finder start-up and solve the problem.

However, this is not the case with the workers because they are implemented by someone other than us. While the workers are certainly using persistent storage (or rather, assuming their storage is persistent), the DPR Cluster has no way of knowing to which folders they are internally writing to. Unfortunately, there is no way for the DPR Cluster to obtain this information from the worker's Docker container image, which is all that is provided. Therefore, this is a part of the system that has to be manually communicated to the DPR Finder. Whoever is trying to deploy a certain cache-store as a worker must know where that cache-store stores the data it wants

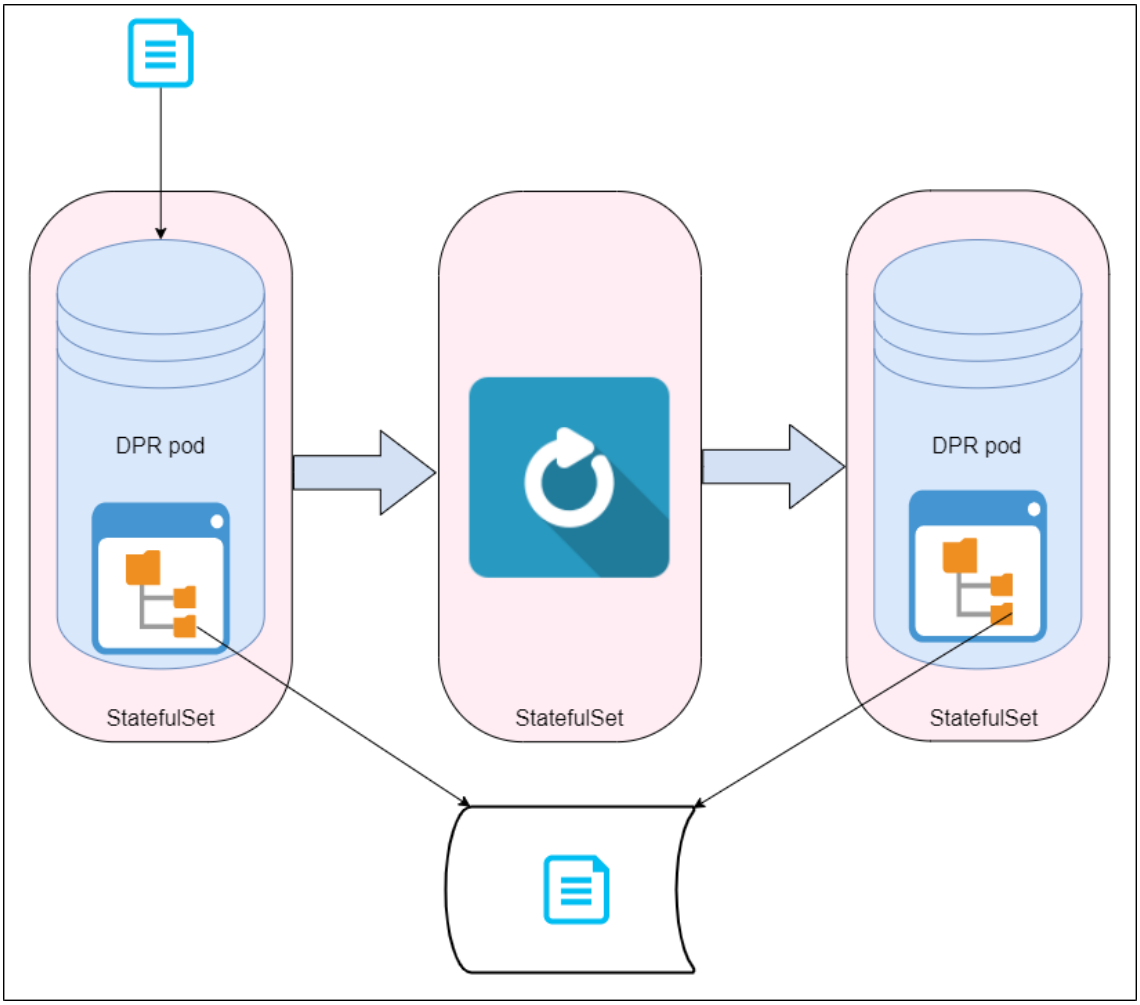

Figure 4-3: A path of a written file being persisted. If written to the right folder, then it ends up inside persistent storage that always exists. When the pod fails, like in the second example, the StatefulSet will take note and reset it while persistent storage remains. Finally, once a new pod is up, the same folder is pointing to the same persistent storage and the file is waiting inside.

to be persisted. They, as the cluster administrator, specify that information when deploying the cluster, and only then can we make sure to assign the worker persistent storage. More details on what exactly happens once the cluster administrator provides this information are in the Deployment chapter since the Python API handles it.

4.2.2 Persisting Cluster Architecture in DPR Finder

As mentioned in the System Overview chapter, we are also extending the DPR Finder to keep track and persist the cluster architecture so it can be communicated to the client. This consists of keeping track of each worker's ID and mapping them to each worker's type and the port they can be contacted on through Ingress. Any inconsistencies about how the cluster looks are undesirable, and we store the cluster architecture in a PingPong Device.

The only question here was whether to store the cluster architecture in a PingPong Device of its own or store it in the same device as the distributed prefix. Storing it in the same PingPong Device makes intuitive sense since we would have single persistent storage. It would also be slightly easier to implement in code since we would only have to modify what we put inside the original PingPong Device rather than add a completely new device and then write to it.

However, there were also problems with using the same PingPong Device. First of all, we would have to write one piece of data in the file, followed by the other. Writing in such a fashion means that when the length of the first piece of data, whichever one we choose, changes, we need to rewrite the whole file. Our other option is to introduce a buffer zone between the two writes so that we can freely modify only a single one of them.

Whenever we write a new version to the front file of the device, we are actually only changing one piece of data, either the architecture or the distributed prefix. This means that we have to copy over the second piece of data from the front file to the back file. While this is doable, it also means that we are two times slower at writing than before we kept track of the architecture. On top of that, if we used two separate PingPong Devices, we would only write each piece of data when appropriate and keep the same performance.

Hence, using two PingPong Devices is better performing. Therefore, we decided to have two separate devices and coordinate them separately, even though implementing them was less straightforward. We point the second PingPong Device to the same

folder as the first one so that we do not have to persist any additional storage.

Chapter 5

Networking

We tackle two types of networking in this chapter: inside the cluster and outside the cluster. The former concerns how the DPR Cluster components talk to each other, while the latter concerns how the client communicates with the cluster. In addition, recall that we are keeping the cluster architecture for FetchCluster() by piggybacking off the usual cluster communication. We also elaborate on how exactly we achieve this in the intra-cluster section.

5.1 Intra-cluster Networking

5.1.1 Routing Requests with Services

Recall that we represented the DPR Finder and each worker as a StatefulSet of its own. Therefore it makes sense to provide them with DNS in the same way. The discussion that follows is regarding the DPR Finder, but the same logic can be applied to the workers, and all of the decisions are the same.

For intra-cluster communication, we are using the standard approach of having a service in front of the StatefulSet, allowing us to use intra-cluster DNS. There were two choices: a regular Service and a Headless Service [13]. A regular Service cannot distinguish between pods and sends traffic to an arbitrary pod determined by the Service (useful for, e.g., load-balancing between replicas). On the other hand,

a Headless Service strictly distinguishes between pods and can only be used with a StatefulSet, acting more like a DNS server with a record of each pod's DNS and IP.

In our situation, we only have a single pod. This means that the regular Service would always lead precisely to the pod that we wanted to contact. Hence, it is easier to use the regular Service since the workers do not have to know the actual name assigned to the DPR Finder pod in the YAML file. This removes the need for this information to be passed down to the workers and allows cluster administrators to change the pod's name without creating any issues. This choice also helps us fail faster in the case of a bug in which the cluster administrator or we accidentally deploy multiple DPR Finders. With a Regular Service, we will immediately know something is wrong, whereas the issue would be much more subtle in the other case. Finally, using a regular Service is also more convenient due to client-DPR Finder communication and this argument is provided in the incoming section regarding the client.

5.1.2 Cluster Architecture Maintenance

DPR Cluster must maintain additional metadata such as cluster membership and worker names. Such metadata must also be resilient and reflect a consistent view of the cluster state in the face of component failures. We may store this information implicitly as part of maintaining DPR guarantees using the DPR Finder or explicitly on a separate code path with persistent storage.

Mechanism

Whenever a worker joins the cluster for the first time, it will send a NewWorker() command to the DPR Finder. The DPR Finder now knows that this specific worker has just joined the cluster. Of course, if it is not the first time the DPR Finder has received such a request from a worker, then that means that the worker was and remained a part of the cluster. The DPR Finder then adds this worker to its own copy of cluster architecture. The issue is that the DPR Finder does not generally

need information about what type of worker it is talking to and the NewWorker() command only sends worker IDs. In addition, each worker has a port at which it needs to be contacted, which the DPR Finder does not know.

These issues can not be circumvented because the client needs this information to properly use the DPR Cluster, which means that the DPR Finder needs to have it so that it can eventually send it to the client. Because of this, we extend the NewWorker() command to send the worker's port and the worker's type to the DPR Finder. The worker already knows its own "type" but does not know its port. In this chapter, we assume that it also knows its port and explain how it knows it in the Deployment chapter. Note that extending the request is extremely cheap since the message is only extended by 16 bytes. Upon receiving these modified commands, the DPR Finder maps each worker's ID to the worker's type and port.

The DPR Finder also needs to persist the cluster architecture. This is because the workers do not resend NewWorker() commands to the DPR Finder when it restarts. Hence, the DPR Finder would have no way of learning the architecture again. To persist the cluster, we utilize another PingPong Device and mimic the existing mechanisms of persisting storage inside DPR Finder. We design our own protocol for writing and reading the cluster architecture from the PingPong Device fails. We also ensure to write down the cluster architecture every time it changes so that we always have the latest version in storage. Finally, we only send an ACK of a worker being added as part of the cluster once we have persisted the cluster architecture. We do this to avoid the situation in which the DPR sends the ACK and then fails before persisting the storage. Then, once it comes back up, it will "forget" about the worker.

Design Trade-offs

Persisting the cluster architecture in a separate PingPong Device required a solid amount of coding changes, enough to make us wonder if there is a simpler approach. The alternative approach was not to persist the cluster architecture at all. Rather, we could augment the ResendGraph() request with the same information as the

NewWorker() request. Then, whenever the DPR Finder is restarted, it can assemble the cluster architecture from all the ResendGraph() requests. While much simpler to implement, this approach also takes longer because the DPR Finder has to re-learn the cluster each time. In addition, if a worker is down for some time, the DPR Finder will not find out about it until it recovers. In contrast, the original approach is immediately ready and does not depend on the state of the cluster, making it a better choice.

5.2 Client Networking

There were two solutions we considered for setting up client networking. The solution we ended up using is utilizing an Ingress Node which we describe in the following subsection in greater detail. The second solution utilizes LoadBalancers in order to make the cluster public. This solution would be much cleaner because LoadBalancer [13] is a basic Kubernetes object and an Ingress Node is not. We discuss why this approach cannot work at the end of the chapter for anyone interested.

5.2.1 Ingress Node

An Ingress Node [5] mediates communications between the clients and the DPR Cluster components; it has its own persistent IP that is public at which it can be contacted. The Ingress Node's original purpose is to reroute HTTP requests. Ingress Node takes in several **rules** that tell it how to route incoming HTTP requests based on the request's prefix or suffix. These are specified inside the node's YAML file. The rules are incredibly user-friendly and flexible. Unfortunately, DPR is utilizing TCP connections in order to communicate. Fortunately, we employ an Ingress Node backdoor for establishing TCP connections.

In addition to an IP address, every TCP connection also requires a port to which it connects. Ingress Node allows routing TCP connection requests through these ports. It is possible to open listening TCP ports on the Ingress Node and specify a mapping from these TCP ports to internal DNS addresses inside the DPR Cluster. We use

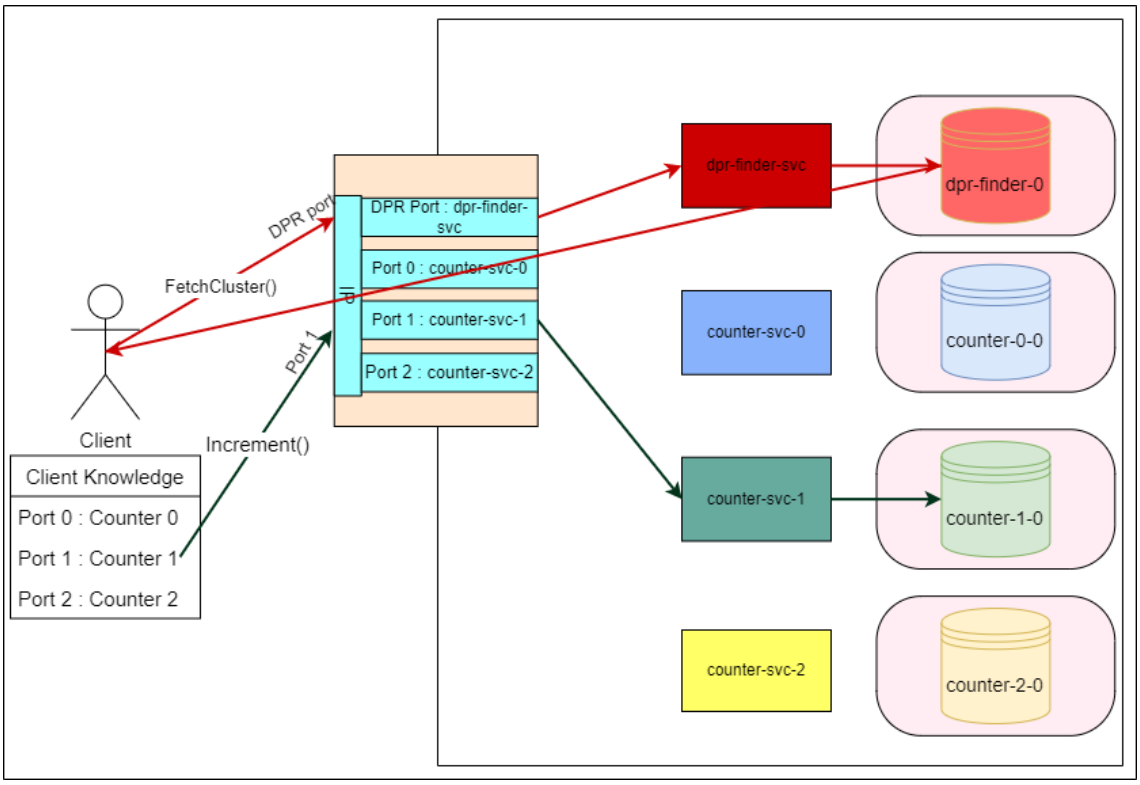

Figure 5-1: Full overview of Ingress Node communication. It contains the mapping, and the client knows the Ingress Node's IP and the DPR port. It sends the FetchCluster() request to the DPR Finder using this knowledge and then learns the port to worker mapping below the worker through the response. It then utilizes the knowledge and contacts appropriate ports on the Ingress Node to contact the workers it wants, like Counter 1 in the Figure.

these features to allow the client to communicate with the cluster.

While deploying the DPR Cluster, we open free ports on the Ingress Node (detailed description of how in the Deployment chapter) and each cluster component gets a port mapping to its DNS. The IP of the Ingress Node and DPR Finder's port is assumed to be known by the client. The client then uses these two to connect to the DPR Finder and fetch the cluster architecture like in Figure 5-1. Afterward, the client knows the cluster architecture and can begin sending operations. We also make the client fetch the cluster architecture immediately upon connecting to the DPR Cluster.

Finally, there are multiple implementations of the Ingress Node available. We utilize the NGINX implementation. This implementation is the most documented and most tested implementation except for maybe the implementations from major

cloud providers. Each cloud provider's implementation is available only on that cloud provider's platform, whereas the NGINX one can be deployed to all three of them, as well as locally.

Finally, it is worth pointing out that the Ingress Node will not hinder the performance of the DPR Cluster as it is a commonly used tool in major distributed systems much bigger than our own.

5.2.2 Load Balancing Approach

It is possible to make a LoadBalancer external and, in doing so, get Kubernetes to expose the LoadBalancer to the public and provide it with a public IP. Hence, it is possible to attach this Service type to a worker or a DPR Finder and effectively expose them to the outside world. Then the client could use this public IP to communicate with each component. The issue with this approach is that the Load Balancer usually assigns the IP on its own and randomly. A simple but unacceptable fix would be to manually observe these IPs and then feed them to the client. It would have been truly convenient if this approach worked since LoadBalancers are a basic component of Kubernetes, easy to deploy and use. Both other proposals are more complex objects and they are not Kubernetes objects.

Another approach is to have each worker sends the IP address to the DPR Finder through an NewWorker() augmentation as described above. The problem with this is that the worker's IP is volatile and subject to change, and sending that IP would be wrong. Sending the LoadBalancer IP would be the right approach and everything would work correctly. However, the workers and their LoadBalancers (Services in general) are entirely separate entities and the worker has no way to observe the LoadBalancer's IP. Generally, it is possible to pass information to the worker through its environment (discussed in the Deployment chapter). However, the LoadBalancer is tailored to the specific worker or the DPR Finder and is formed after the worker, thus not allowing us to pass the IP address properly. Of course, there are ways to circumvent this, like starting up a Kubernetes Job to let each component know its IP. However, approaches such as this are over-complicated and artificial and would

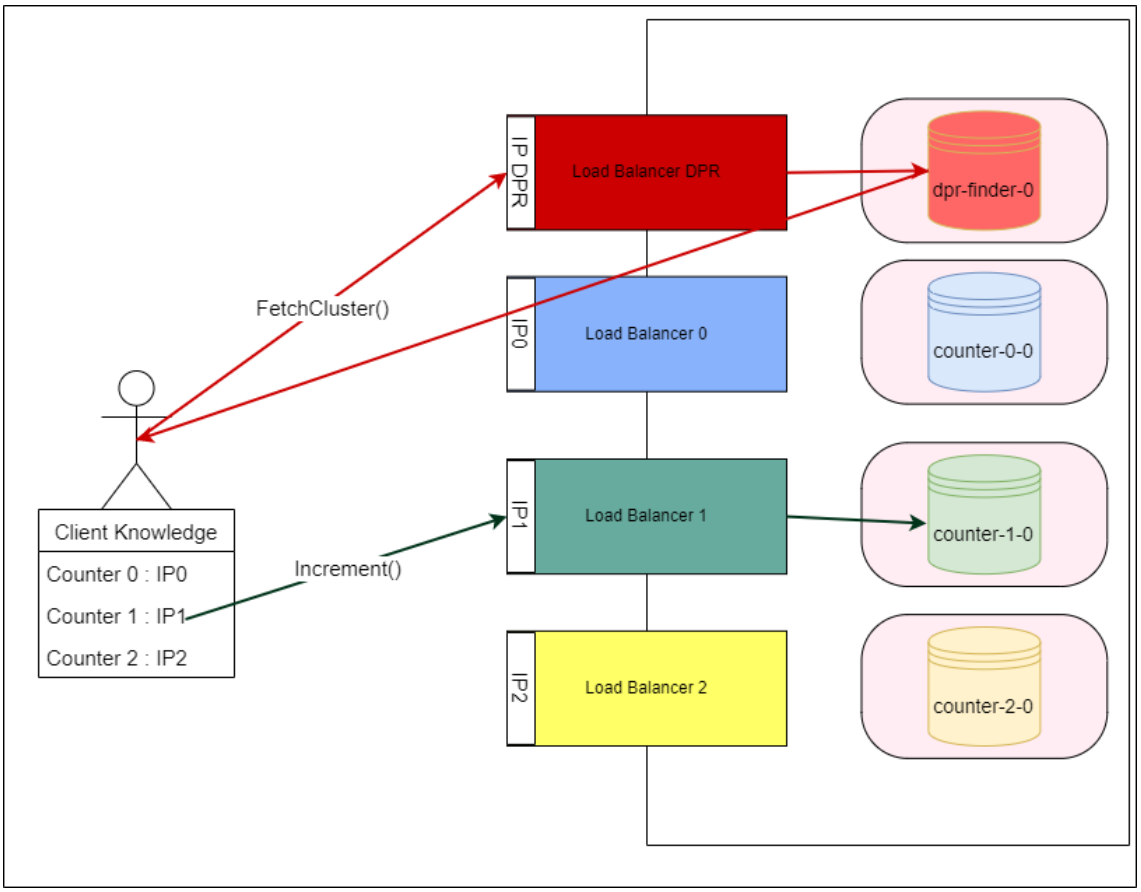

Figure 5-2: The client first sends the FetchCluster() operation to the DPR Finder by using DPR Finder's Load Balancer IP. This request is then forwarded to the DPR Finder through its Load Balancer, and the DPR Finder then responds. Only when the DPR Finder responds the client gathers the knowledge displayed in the Figure. After that, the client may use their knowledge to contact Counter 1, for example, like in the Figure.

potentially significantly increase the startup time.

The final attempt at circumventing the problem with IPs being assigned too late is to make all the IPs custom and static. Specifically, we would reserve a bunch of IPs that are never used for other purposes except when we put them to use in the DPR Cluster, and we would know these IPs. Then, we could pick the static IP of each LoadBalancer before we deployed any parts of our cluster and we could have passed that IP to each DPR Cluster component. Then the workers would let the DPR Finder know their IPs and everything could proceed smoothly. While this approach works, it is neither portable, easy to use, or scalable. Namely, reserving IPs

in such a manner can only be done on cloud providers such as Azure, Amazon, and others. They have to be reserved manually and they have to be manually input into the Python deployment code. This would also have to be done for every new cluster started separately instead of simply working out of the box. In addition, the number of IPs we can reserve is fairly limited. Thus, if we had a large cluster that we wanted to deploy, we would not be able to do so. Finally, while it is not a deal breaker, the fact that we have to use a cloud provider to deploy the cluster is inconvenient for testing the cluster and development in general while also costing money. For all the reasons mentioned, this approach was unacceptable, although it would have worked.

Chapter 6

Deployment

Cluster deployment is handled automatically by utilizing Kubernetes Python APIs, the Core API and the Apps API [9]. The difference between these two is simply in the objects they can deploy and manage. This chapter discusses how each cluster component is deployed, from the workers to the Ingress Node. We outline various options for customizing the cluster that we support. Some of these options are basic, but we also provide some options that seem unexpected at first glance. Finally, we explain how to add and remove cluster components while the cluster is running.

6.1 Component Deployment

6.1.1 Cluster Administrator Side

Like the Kubernetes CLI, each API command for starting a Kubernetes Object takes in a YAML file. However, the API calls take in the YAML file as a nested dictionary. For both the workers and the DPR Finder, we keep a template YAML file that we load and convert to a dictionary of dictionaries. Each of these files consists of primary YAML fields that each of the two cluster components needs to have, such as the kind of the Kubernetes Object to be used, for example. We utilize Python to fill in the rest of these files with either default values or the values provided by the DPR Cluster administrator.

Some of the more basic options the cluster administrator can specify when adding a worker to the cluster:

- CPU request, CPU limit — The first option tells Kubernetes the smallest amount of CPU that it has to assign to a pod. Kubernetes will report a failure if that is impossible to achieve. The second option tells Kubernetes the maximum amount of CPU it should assign to a particular pod. The administrator does not have to specify either of these; their default options are simply empty, thus leaving it to Kubernetes to assign resources as it deems appropriate.

- Memory request, Memory limit — These two are the same as the above options except regarding memory. Their default values are empty as well.

- Storage capacity — This option is entirely dependent on the actual worker implementation and the type of data the worker itself stores. This option can not be left empty and needs to be specified inside the volumeClaim before the pod is even deployed. We do not force the administrator to input this value; we use a default value of 10GB in this case, but it is risky not to do so.

Each of these can also be specified for the DPR Finder.

```
def startCluster():
    cluster = KubernetesCluster(cloud=False)
    cluster.addServer("counter", image_location="cetko24/meng_project", storage_location="cetko24/meng_project",
                      listening_port=80, storage_capacity="2Gi", memory_request="800M", cpu_request="1")
    cluster.addServer("counter", image_location="cetko24/meng_project", storage_location="cetko24/meng_project",
                      listening_port=80, storage_capacity="5Gi", memory_request="700M", cpu_request="2")
    cluster.start()
```

Figure 6-1: An example of deployment from the cluster administrator side. They specify everything we discussed for each worker and deploy the cluster.

There are additional choices that the administrator can specify, and they are regarding the workers. We can not expect a worker implementation to be changed in order to work within the DPR Cluster, and that would defeat the purpose of the DPR Cluster. Because of this, we try to allow the administrator to tailor the DPR Cluster to the workers that it is hosting. We now list these options:

- Storage Location — It might be hard to change the storage location inside worker implementations. Even if it was not, the fact that it does not have to be done still lowers the barrier of entry to the DPR Cluster. Whatever the administrator specifies here is where persistent storage will be mounted. This attribute has to be specified by the administrator.

- Image Specification — The administrator should provide the Docker image of the worker implementation they want to use and publish it online. Once they have done that, they need to specify this location to the cluster so that it can load the correct implementation. This attribute also has to be specified.

- Listening Port — A worker might not only be communicating with the DPR Finder and the other workers. For these reasons, it might not be possible to change the port on which it accepts requests. However, there is no way for the cluster to know this port unless specified. This is why the listening port is also a mandatory attribute.

6.1.2 From Specification to Cluster

Once everything above was specified, we take it all in and start building the cluster. The first and simplest step is starting up the DPR Finder. Since we completely control its implementation, we can have an almost ready YAML template file for deploying the DPR Finder. We load the YAML file into code and incorporate the administrator's specification of resources at appropriate places. Once done, we simply feed it into the Core API and we have it create the DPR Finder. The DPR Finder's Service YAML requires no changes, so we can load it and feed it to the App API to set up DNS for the DPR Finder. Note that these template YAML files being references have the classic Kubernetes attributes for connecting and recognizing each other. These details are not as important and are not discussed in the thesis.

When deploying the workers, we load the template YAML files as usual and embed the specified resources in their respective YAML dictionaries the same way as for the DPR Finder. However, deploying at this point would cause all the pods and their

Services to have the same name, which would not work, as discussed in the System Architecture chapter. At this point, Python gives a random order to the workers and assigns each of them their unique ID. It then proceeds to embed these IDs into respective YAML files and thus names each pod differently and evades collision. It renames the Services in the same way as well. At this point, we are still not ready to deploy the workers. We still need to tell them the Ingress Node port they are listening on so they can later pass it on to the DPR Finder, as discussed in the Networking chapter. We pass these ports to the workers through the YAML file by making the port an environment variable. Then, when the worker starts, it can simply get the port.

When deploying the Ingress Node, we first select available Ingress ports and map them to the Services. The first thing we need to modify is Ingress Node's ConfigMap **tcp-services** that is, among other things, used to specify the port to DNS mapping. We form the necessary mapping with Python and format it appropriately for the ConfigMap. Once we have made the mapping, we need to ensure that the Ingress Node is listening on all the ports we are mapping. For this, we need to modify the Ingress YAML file with all of this port info. We utilize Python to properly format what this would look like inside the YAML file. At this point, we have the Ingress Node listening on the specified ports and the mapping that specifies how to route the requests to those ports.

Recall that we do not usually send requests to Kubernetes Objects but their Services. When deploying a cluster on a local machine, the Service will not have an external IP at which it can be contacted. Instead, what is used is the local (persistent) IP of the whole cluster. Hence, in this setting, we directly contact the Ingress Node pod since that is the only way. When deploying on Azure (or other providers), the Service is the one that actually has the external IP and the one that the client needs to contact. Therefore, we also need to open the ports on the Service and have them route the requests to those same ports on the Ingress Node. Note that we can map any Service port to any Ingress Node port, but, in our specific case, the mapping is an identity. Finally, we form the appropriately formatted string for the Service's YAML

file.

It is important to note that, once set up, the Ingress Node is constantly running as a part of our cluster and waiting for outside requests. For these reasons, we can not manipulate its YAML files the same way as other components and must resort to **patching** them through the APIs. Patching modifies these files on the fly. It is a bit harder to specify the patching format in code correctly, but we take care of that for the administrator. We then apply all the properly formatted changes above as patches and finally set up the Ingress Node.

Note that even if there was a way to take the Ingress Node down before deploying the cluster, we could not do that every time. For example, the Ingress Node needs to be updated and patched every time we add a new worker to the cluster, which does not only happen when we initially deploy the cluster.

6.2 Dynamic Cluster Changes

We do not want to take the whole cluster down because we want to add or remove a single piece from it which is why it is possible to modify the cluster on the fly. Adding the worker is the more straightforward of the two tasks and almost painless, while deleting a worker is slightly more complicated.

When a worker is deployed and becomes a part of the cluster, it proceeds to send a NewWorker() request to the DPR Finder. This is an already defined situation for the DPR Finder that can recognize that the worker indeed just got added and let the other components know as well. Hence, the only question that remains is how to deploy the worker. We follow the same process as during the initial deployment of attaching the proper resources to the worker and ensuring that the worker's name and its Service's name do not clash with the existing ones. We also assign an Ingress Node port to the worker and patch the Ingress Node's Configmap and YAML files the same way as above. A good thing about patching is that it will simply extend these files and will not overwrite their contents with only information about the new worker. At this point, we deploy the worker and let the DPR logic take care of everything, as

explained.

A DeleteWorker() operation exists that a worker can send to a DPR Finder along with its ID to signal it is being turned off. In response, the DPR Finder would know that the worker is no longer a part of the cluster, and it could proceed to remove it from the distributed prefix and slowly distribute this information to others. Unfortunately, there is no way for us to let the worker know that it should indeed send this operation. When removing a worker from the cluster, we can not simply delete the worker's pod since it will only get reset by the StatefulSet that manages the worker. Instead, we need to delete the whole StatefulSet. While there are ways to define behavior for graceful shutdown or deletions of a pod, it is impossible to specify what the pod should do when the whole StatefulSet is deleted. Therefore, from the worker's perspective, it simply goes down and never comes back up, entirely unexpectedly.

Because of the outlined reasoning, the only thing that we can do is let the DPR Finder know that the worker has been deleted through alternate means. Note that we do know the ID of the deleted worker and that the ID of the worker is the only argument in the DeleteWorker() request. Hence, if we can still manage to send this request to the DPR Finder some other way, everything will still be fine. We propose deploying a Kubernetes Job [6] that only sends the DeleteWorker() command to the DPR Finder, waits for it to be successfully executed and then finishes. Setting up and starting up the job is straightforward; we can pass it the worker ID that it needs, and we will be sure that the worker has been deleted once the Job executes (since Kubernetes will keep restarting it until it successfully finishes). At this point, there will be no trace of the worker inside the DPR Cluster.

Finally, note that we can not naively only delete the worker from the cluster and ignore the DeleteWorker() command. The worker will still be there each time the client fetches the cluster. This might lead to clients contacting the worker even though it will never become available again and potentially waiting on it.

Chapter 7

Fault Tolerance

Even though the original DPR protocol is fault-tolerant in the sense that it guarantees prefix consistency, DPR Cluster must correctly trigger DPR recovery, and additionally be resilient against code-level failures (e.g., broken sockets elsewhere due to a worker failure). In this chapter, we focus on these additional fault-tolerant mechanisms in DPR Cluster.

7.1 Resilient Connection

In our implementation, components of DPR Cluster communicate with each other through sockets. Normally, socket implementations throw exceptions and terminate upon connection failures. This is problematic as applications may not properly catch such exceptions; a failed DPR worker may cause all other connected components to crash due to uncaught exceptions. Other stateful connection protocols such as WebSocket can display similar behavior. To address this, DPR Cluster hardens each worker, client, and the DPR Finder with retry and reconnect logic. For DPR Finder, it only receives requests and it is never the one to initiate contact. Once it receives a request, it takes care of it asynchronously. If the worker that has sent the request to the DPR Finder fails in the meantime, the asynchronous thread will also fail in replying to it. However, the fact that there was an error in the asynchronous thread does not affect the main thread running the DPR Finder. In addition, because the

worker failed before getting any sort of response, we know that the request will be re-sent at some point. There is nothing for the DPR Finder to do here and it can simply ignore whatever happened asynchronously.

Workers on the other hand, will experience a SocketException when the DPR Finder it is connected to fails. Note that this can happen for any request the worker sends. Hence, we catch for SocketException at every single request. If we catch the exception, we initiate the re-connecting protocol, after which we re-send the request. Note here that such connection failures may also not throw an exception. For example, when the request it has sent to the DPR Finder goes through but the DPR Finder fails before (or in the middle) sending a response back. Trying to receive on a socket is usually a blocking call but, because we are communicating on Kubernetes intra-cluster network, whenever the DPR Finder crashes the worker gets a graceful disconnect signal. Specifically, the worker is trying to receive a certain number of bytes on the socket, and the receiving call starts returning zero as the number of bytes received (instead of blocking), signaling that the connection is down. We wrote a wrapper around the receiving method for the socket that throws a SocketException if it ever starts receiving zero bytes. The exception gets caught the same way as above and we ensure we detect the problem as quickly as possible. To reconnect a worker to the DPR Finder, we try to connect the socket to the same DNS that the DPR Finder always has. Until the DPR Finder is up and running, we will keep getting a SocketException. This method must not be recursive since it might be a while until we can re-connect.

The final piece left to discuss is the client and how we achieve fault tolerance in communication with the DPR Cluster. Similar to the communication between the workers and the DPR Finder, the client could also fail if the worker it is trying to contact is not available or block if the worker fails before responding. Therefore, we needed to implement safety precautions here as well. When a worker or the DPR Finder is down when the client contacts them, it gets a SocketException and we catch those and try to re-connect. However, when these go down after receiving the client's request but before responding, the client blocks forever since there are no

graceful signals and socket receiving is blocking. For this reason, we had to introduce a receiving timeout for the socket, which makes it throw a SocketException if it has not received anything for our defined time threshold. Once it fails, we re-connect the same as above.

7.2 Interfacing with DPR

When Kubernetes triggers a restart of a worker, DPR Cluster must correctly convert that signal into a rollback for the DPR protocol. We now briefly describe how these two pieces connect. A worker restarted by Kubernetes is guaranteed to have exclusive access to its assigned storage volume and DNS address, and executes its normal startup routine by sending NewWorker to the DPR Finder. The DPR Finder, as discussed, resiliently stores information about cluster membership, and can detect that an already recognized worker id is executing its start up sequence, which implies that a restart has occurred. At this point, the DPR Finder computes the surviving prefix and sends out rollback signals as the DPR protocol prescribes.

To complicate matters, the DPR Finder may also experience restarts during this process. Logically, the DPR Finder aggregates information across persistent storage of workers, which can always be reconstructed on failure. To ensure correctness, the DPR Finder always queries each worker about their state before completing recovering and beginning normal operations. However, because worker restart depends on a functional DPR Finder, as described before, this creates the possibility for a deadlock situation. To handle this situation, we implement two mechanisms:

- **Preemptive resend:** recovered workers preemptively sends its local DPR information to the DPR Finder in parallel with registration attempts, in case the registration attempt is blocked because of a DPR Finder restart.
- **Recovery log:** The DPR Finder logs any rollback it wishes to orchestrate on persistent storage (provided by Kubernetes) to guarantee progress in the case of repeated DPR Finder failures.

Chapter 8

Evaluation

The project's two main goals are portability and fault tolerance. By portability, we mean easy local deployment of DPR Clusters on most devices and straightforward deployment to major cloud providers. Achieving this goal significantly decreases the barrier of entry for utilizing DPR applications since it makes it possible for virtually anyone with an internet connection to deploy a DPR Cluster of their own.

DPR itself offers fault tolerance from the data perspective, but that does not make it fault tolerant. When run in a distributed setting like the one we provide, we had to ensure that there were no unhandled failures of any kind. It was important for the deployed application to work entirely for the project to make sense.

Of course, goals such as system performance and having high throughput and low latency of requests still matter. However, these mainly depend on the DPR implementation and not so much on the DPR Cluster. DPR Cluster does not modify the behavior of system components and therefore does not affect their performance. There might be an effect on throughput and latency of requests. However, that effect should be negligible and, if anything, positive since Kubernetes intra-cluster network is faster than a regular network.

It is not clear how to evaluate portability. However, we do demonstrate the simplicity of deploying our system locally. We also deploy the cluster on Azure with minimal effort.

In order to test for fault tolerance, we have designed extensive testing that tested

every part of the DPR Cluster, from the client to the way data is stored. We offer a closer look into these and an argument for fault tolerance in the section on fault tolerance below.

8.1 Portability

8.1.1 Local Deployment

Deploying the cluster locally was done on Linux as the most common development platform, especially since the development of WSL. Deploying the DPR Cluster requires the user to have Docker and Kubernetes installed on Linux, both of which are easy to do. At this point, we provide a startup script that sets up the user's environment for running the DPR Cluster. For example, this script sets up the initial version of the Ingress Node. We also provide a script for building Docker container images from the provided code and putting them in the local cache to be used with Kubernetes. This removes the burden from the user of having to publish the images to a public Docker container registry. Once the above is done, the user simply has to access the python code for deploying the cluster and run it. Running the script deploys the fully working cluster.

With a total of three steps in total in order to deploy a DPR Cluster, we claim for this process to be highly flexible and easy to use. In addition, it works on any somewhat recent Linux distribution.

8.1.2 Cloud Deployment

With minimum effort, we have successfully deployed the DPR Cluster on Azure as an example of a major cloud provider. We see no reason for the other providers to be more complicated. The user does need to put in more work than when deploying a local version of the DPR Cluster, but that work is still relatively simple.

The user needs to set up a Kubernetes Cluster and a Docker registry on the cloud provider of their choice. Note that we could not have done this automatically for the

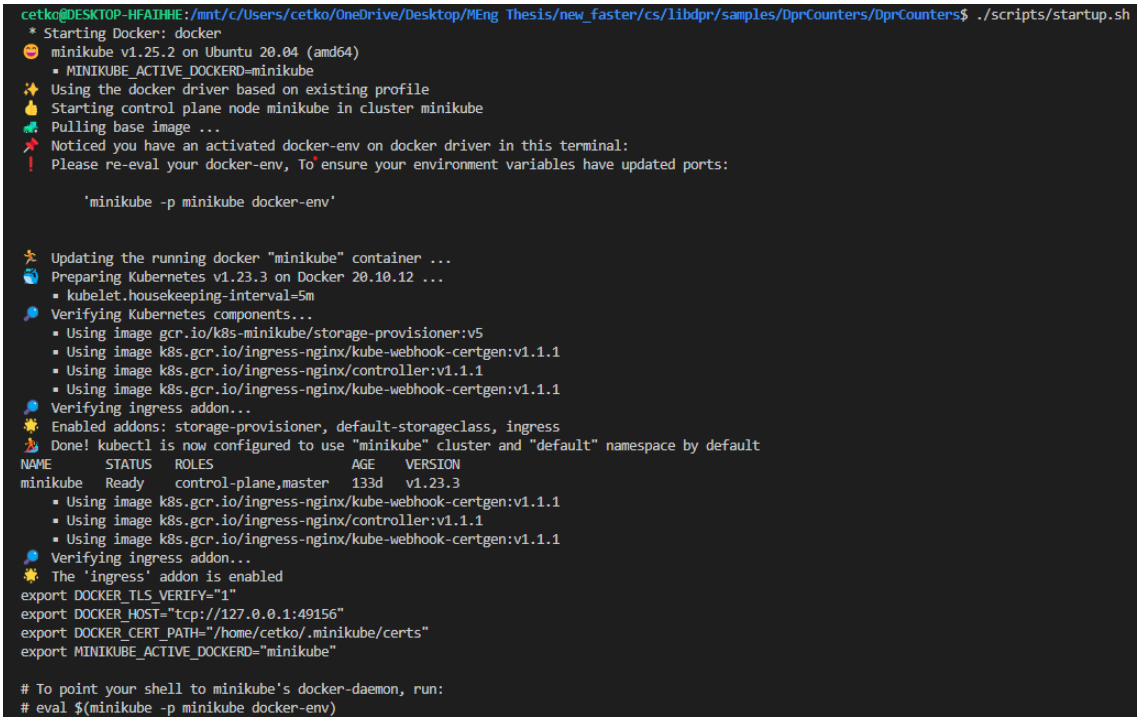

Figure 8-1: An example of the startup script used and the environment being set for the user. Note that the fields that seem like they could have been hard-coded were, in fact, not. The exported variable values are automatically taken and put in, and they are not something we have inputted manually

user even if we wanted to. The user must also push Docker container images to their own Docker registry. Finally, the user needs to follow the instructions for deploying an NGINX Ingress Node in their Kubernetes Cluster. However, most users of this project likely already have all of this set up except possibly the Ingress Node.

Once everything is set up, there is no need to run any scripts and the user can run the same Python deployment as it does locally. Figure 8-2 provides an example of deployment on Azure. Once again, deployment does not get simpler than this.

Finally, it is important to note that the client is always run from the local machine and that it can contact the DPR Cluster when it is deployed on Azure. Thus, the cluster is immediately available and reachable from anywhere, not just the same local machine. Figure 8-3 shows the Ingress Node services deployed on Azure and the Ingress Node's (and therefore our cluster's) IP.

```
nikola@Azure:~/FASTER/cs/libdpr/samples/DprCounters/DprCounters$ kubectl get pods
No resources found in default namespace.
nikola@Azure:~/FASTER/cs/libdpr/samples/DprCounters/DprCounters$ python3 python/start_cluster.py
Pod dpr-finder-0 has been deployed
Pod counter-0-0 has been deployed
Pod counter-1-0 has been deployed
Pod counter-0-0 is running
Pod dpr-finder-0 is running
Pod counter-1-0 is running
nikola@Azure:~/FASTER/cs/libdpr/samples/DprCounters/DprCounters$ kubectl get pods
NAME           READY   STATUS    RESTARTS   AGE
counter-0-0    1/1     Running   0          2m37s
counter-1-0    1/1     Running   0          2m37s
dpr-finder-0   1/1     Running   0          2m37s
```

Figure 8-2: As you can see on the left, we are indeed deploying on Azure. No pods are running, and we then spin everything up only by utilizing Python. Of course, an actual cluster specification is made as described in the Deployment chapter inside the Python file.

```
nikola@Azure:~/FASTER/cs/libdpr/samples/DprCounters/DprCounters$ kubectl get svc -n ingress-nginx
NAME                                 TYPE           CLUSTER-IP    EXTERNAL-IP     PORT(S)
ingress-nginx-controller             LoadBalancer   10.0.31.78    20.223.12.243   6379:31100/TCP,6380:31101/TCP,6381:31102/TCP
ingress-nginx-controller-admission   ClusterIP      10.0.11.68    <none>          443/TCP
```

Figure 8-3: As evidenced, a real public IP of the DPR Cluster indeed exists. Also, take note of the three seemingly random ports that are open on the Service. These are the ports that are being mapped to each one of the DPR Cluster components.

8.2 Fault Tolerance

Demonstrating fault tolerance is much more objective than portability. We demonstrate DPR Cluster's fault tolerance by conducting thorough testing that exposes the cluster to all sorts of failure scenarios, all while the cluster is communicating with the client. We deem the test successful if the client can eventually execute their whole batch of operations and if the workers' states at the end of the operations look like they should. First, we outline how we have designed our test and then describe the scenarios we tested for. Then, we provide an example of a test and the results we have obtained from it.

8.2.1 Testing Framework

We designed one flexible testing framework that we were able to modify to test for all sorts of behavior called **ChaosMonkey**. As its name suggests, we essentially use it to create chaos. We continuously kill off the pods, usually after a certain time. It

is possible to specify a sequence of pods to be killed when trying to replicate a bug for example, and we can also kill the pods at random.

It is important to note that once a pod has been killed, the ChaosMonkey waits for it to come back up. There are multiple reasons for this. First, if we keep killing pods after some short time interval, we will eventually reach a state where most of the pods are constantly down. This happens because the more we kill the pods, the slower they are to come back up (not the case in general, only when killing pods aggressively). Thus, we manage to avoid our cluster being constantly down. In addition, this allows the ChaosMonkey to adjust to the average pod downtime that, as mentioned, keeps increasing. The second reason for waiting is because killing a pod that is not yet back up is not testing anything. It only kills the new pod that was still restarting and not visible. From the point of view of all the other cluster components, this pod never comes back up at all, and it only seems like it has been down for a very long time, which is missing the test's purpose.

```
nikola@Azure:~/FASTER/cs/libdpr/samples/DprCounters/DprCounters$ python3 python/start_cluster.py
Pod dpr-finder-0 has been deployed
Pod counter-0-0 has been deployed
Pod counter-1-0 has been deployed
Pod counter-0-0 is running
Pod dpr-finder-0 is running
Pod counter-1-0 is running
Killing pod: counter-0-0
Killing pod: counter-1-0
Killing pod: counter-0-0
Killing pod: counter-0-0
Killing pod: counter-1-0
Killing pod: counter-1-0
Killing pod: dpr-finder-0
Killing pod: counter-1-0
Killing pod: dpr-finder-0
Killing pod: counter-1-0
Killing pod: counter-0-0
```

Figure 8-4: The deployment side with ChaosMonkey turned on. In this version, a random pod is selected and killed every so often.

On the client side, we only send and commit one operation at a time. This means that once an operation is sent, we wait for it to be committed to the DPR finder before sending new operations. If a rollback occurs and our operation has not and will not be committed, we can resend it. While it was indeed possible to send operations

continuously and then roll back whole batches of them, we did not need to implement that kind of logic on the client side for testing purposes of the DPR Cluster. In addition, sending many operations would make it significantly more complicated to figure out what exactly is going on when a bug does occur.

For every operation, we send it to every worker. Note that it does not make it easier for the cluster to work in any way. We do it because it is easier to track what exactly was happening with each worker. In the end, their counts should increase by the same amount at every spot and add up to the same tally after the client is completely done executing. The client is run from the local machine and talks to the DPR Cluster on Azure, while ChaosMonkey is killing off cluster components.

We have run all sorts of tests using the described framework. We have run tests that only kill the DPR Finder, only kill the workers, kill the workers at the same time and so on. Of course, the most important test was killing the pods entirely at random and checking that everything still works. We have tested the ChaosMonkey by re-introducing bugs, that we have already fixed, back into the code. It has caught every single one of them and the DPR Cluster has not worked correctly with any bug re-introduced. Due to this fact and the fact that this approach should theoretically expose any bugs eventually, we are quite confident that we have created a thorough tester.

8.2.2 Test Example

At the end of the project, DPR Finder does not crash through any one of the above tests, and the client always executes until the end, which is excellent news and confirms that DPR Cluster is fault tolerant. When the code is working, all the tests converge to the same behavior, and there is no point in showing the same result for each of them. We show and discuss the results of a short test in which we are killing pods off randomly, one at a time, every five seconds. Note that this barely gives the DPR Cluster any time to be fully operational. There are two workers, Counter Servers. Finally, the client sends simple increments of 300 until it goes over 70000 to both workers.

After running the test, we wait and the client finishes, which is already good news. We then check the logs of the two Counter Servers to ensure that they both add up to the same number, 70200 in this case. Consider Figure 8-5.

```
nikola@Azure:~/FASTER/cs/libdpr/samples/DprCounters/DprCounters$ kubectl exec -it counter-1-0 -n default -- bash
root@counter-1-0:/DprCounters# tail data/basic.txt
New value: 69300
New value update:
Old value: 69300
New value: 69600
New value update:
Old value: 69600
New value: 69900
New value update:
Old value: 69900
New value: 70200
root@counter-1-0:/DprCounters# exit
exit
nikola@Azure:~/FASTER/cs/libdpr/samples/DprCounters/DprCounters$ kubectl exec -it counter-0-0 -n default -- bash
root@counter-0-0:/DprCounters# tail data/basic.txt
New value: 69300
New value update:
Old value: 69300
New value: 69600
New value update:
Old value: 69600
New value: 69900
New value update:
Old value: 69900
New value: 70200
root@counter-0-0:/DprCounters# exit
exit
```

Figure 8-5: We access the end of each Counter Server's log file and see that the final value in both of these is indeed 70200 as expected.

This is already proof enough that the test has finished successfully. However, we provide some further insight into everything that has happened and show that ChaosMonkey is properly testing all types of behaviors. In Figure 8-6 we show the DPR Finder's log. Using this log, we can reconstruct the order in which pods were killed depending on what is logged down. The fact that there are no errors also means that each recovery ran smoothly.

Finally, it is also important to cause all sorts of situations to arise while testing. For example, simple tests were never able to cause a Counter Server to decrease its counter to a previous value because an operation has not been committed. While this is normal, specified behavior it was very hard to cause by manually killing pods or manually sending client requests. Figure 8-7 demonstrates that Chaos Monkey also provokes this correct behavior. With this, we can finally conclude that the testing is

```
nikola@Azure:~/FASTER/cs/libdpr/samples/DprCounters/DprCounters$ kubectl exec -it dpr-finder-0 -n default -- bash
root@dpr-finder-0:/DprCounters# cat data/basic.txt
Worker added. Id: 0
Worker added. Id: 1
Worker added. Id: 1
Recovery Complete
Worker added. Id: 0
Worker added. Id: 1
Recovery Complete
Recovery Complete
Recovery Complete
Recovery Complete
Worker added. Id: 0
Worker added. Id: 1
Recovery Complete
Worker added. Id: 0
Recovery Complete
Worker added. Id: 0
Worker added. Id: 1
Recovery Complete
Worker added. Id: 0
Worker added. Id: 0
Recovery Complete
Worker added. Id: 0
Worker added. Id: 0
Recovery Complete
Worker added. Id: 1
Worker added. Id: 0
Recovery Complete
root@dpr-finder-0:/DprCounters#
```

Figure 8-6: Each worker re-addition means that that worker's pod failed, i.e., was killed by ChaosMonkey. Each time we see "Recovery Complete," it means that the DPR Finder has successfully recovered itself, i.e., the DPR Finder is the pod that was just killed. Hence, this means that ChaosMonkey first killed Counter 0, then Counter 1 twice and then the DPR Finder, and so on.

sufficiently thorough and that the DPR Cluster indeed works.

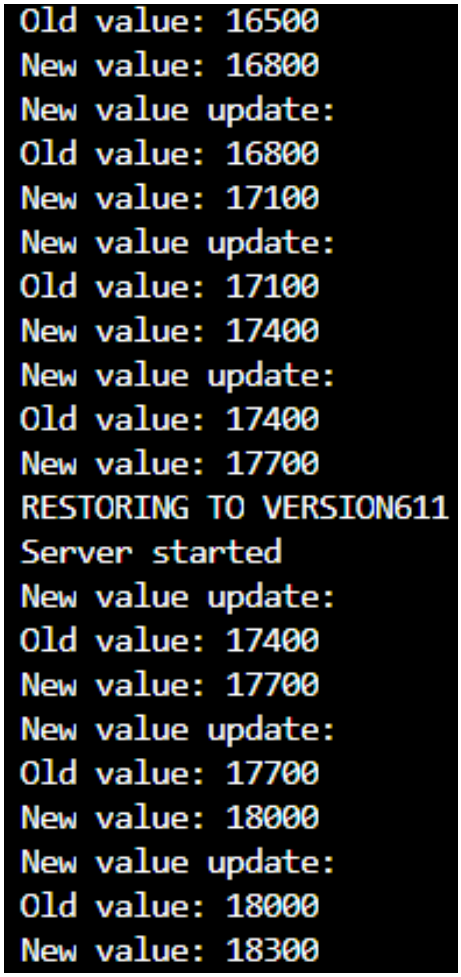

Figure 8-7: Each new value is put inside the log when the counter increases, not when that change gets committed. In this example, the worker went down and then was restored to a previous version by the DPR Finder. When it receives a new operation, it becomes clear that the previous operation has not been committed and the desired behavior has occurred.

Chapter 9

Conclusion and Future Work

We have successfully made deploying DPR applications a reality through DPR Cluster. We have designed Python scripting that allows for seamless deployment of a DPR Cluster. We also wrap each cluster component into a carefully selected Kubernetes object which manages the component for the duration of the cluster and restarts it upon failure. These components communicate using DNS that we automatically set up inside the Kubernetes cluster. We have fixed many bugs and overcome many problems to make the DPR Cluster truly fault tolerant and usable in production. We have designed thorough testing to detect those bugs and later confirm fault tolerance. DPR Cluster is also very portable, being extremely easy to deploy on most local devices and any major cloud provider. We have reached the project's primary goals without compromising in other areas (like performance, for example), and we conclude the project is finished successfully.

The next step in this project would be deploying a true cache-store into the DPR cluster instead of the Counter server implementation that we have used for testing and demo purposes. This step would ultimately fulfill the purpose of the DPR Cluster and pave the way for many other potential deployments. Unfortunately, we did not have time to achieve this ourselves, but we identify it as the next logical step in this project.

Bibliography

[1] Azure storage - secured cloud storage. https://docs.microsoft.com/en-us/azure/azure-sql/database/serverless-tier-overview?view=azuresql.

[2] Controllers in kubernetes. https://kubernetes.io/docs/concepts/architecture/controller/.

[3] Docker. https://www.docker.com/.

[4] Dpr finder code. https://github.com/tli2/faster/tree/master.

[5] How nginx ingress controller works. https://docs.nginx.com/nginx-ingress-controller/intro/how-nginx-ingress-controller-works/.

[6] Jobs in kubernetes. https://kubernetes.io/docs/concepts/workloads/controllers/job/.

[7] Kubernetes cluster networking. https://kubernetes.io/docs/concepts/cluster-administration/networking/.

[8] Kubernetes. https://kubernetes.io/.

[9] Official python client library for kubernetes. https://github.com/kubernetes-client/python.

[10] Pods in kubernetes. https://kubernetes.io/docs/concepts/workloads/pods/.

[11] Redis. https://redis.io/.

[12] Replicaset in kubernetes. https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/.

[13] Services in kubernetes. https://kubernetes.io/docs/concepts/services-networking/service/.

[14] Statefulset in kubernetes. https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/.

[15] Understanding kubernetes objects. https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/.

[16] Volumes in kubernetes. https://kubernetes.io/docs/concepts/storage/volumes/.

[17] What is yaml? a beginner's guide. https://circleci.com/blog/what-is-yaml-a-beginner-s-guide/.

[18] Tianyu Li, Badrish Chandramouli, Jose M. Faleiro, Samuel Madden, and Donald Kossmann. Asynchronous prefix recoverability for fast distributed stores. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD/PODS '21, page 1090–1102, New York, NY, USA, 2021. Association for Computing Machinery.

[19] Guna Prasaad, Badrish Chandramouli, and Donald Kossmann. Concurrent prefix recovery: Performing cpr on a database. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 687–704, New York, NY, USA, 2019. Association for Computing Machinery.

[20] Derek Rangel. Dynamodb: Everything you need to know about amazon web service's nosql database, 2015.