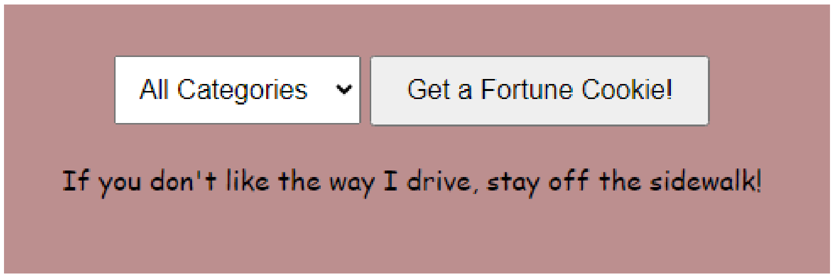
Kubernetes, Team 6

Kristian Hill (202107355), Nikolaj Jakobsen (202105913)

October 2023



If you don't like the way I drive, stay off the sidewalk!

Contents

1 Introduction

In an era where relying on standalone systems is as cutting-edge as using a typewriter for web development, Kubernetes stands tall as an unsung hero of distributed computing. Through a novel use of Kubernetes, a project merging the notion of fortune cookies with the capabilities of distributed systems, we explore why the tool has become so pivotal for managing and orchestrating dynamic environments.

Our objective is quite straightforward: create a web service hosted on Kubernetes that can generate digital fortune cookies on demand. The core of the project is the deployment architecture which is designed to support multiple pods that might join or die dynamically, enhancing the reliability and scalability of the system. Supplanting the default leader election scheme used in Kubernetes [3], we implement a custom version of Héctor García-Molina's Bully Algorithm [1] to elect a new leader pod in case of failure. While the Bully algorithm is a less prominent component of the project, it is nevertheless a key component that plays a vital role in reducing downtime and ensuring continuous service. The pods communicate with each other through an API which is also responsible for serving the website and fortune cookies.

We will only briefly touch on aspects of the bully algorithm[1]. Instead, we provide a high-level overview of the project architecture, delving into details that tie directly to Kubernetes such as .yaml specifications and Minikube.

2 Background information

In this section, we briefly describe certain aspects of the technology stack that are particularly relevant to this project.

2.1 Docker

Docker is an open-source platform that simplifies the process of developing, shipping, and running applications using containerization technology. This allows developers to package all the dependencies inside a standardized unit of software called a container, which unlike a virtual machine does not require a separate operating system. Instead, it shares the OS kernel with the host system, making containers lightweight while still running in isolated environments. One can explicitly expose certain ports. Docker Engine is the runtime that manages containers. The containers are created from static Docker images which in turn are spawned from a custom Dockerfile. Docker conforms to the "Open-Container Initiative" (OCI).

Docker uses a client-server architecture for the interaction between the client and the Docker daemon which is responsible for the heavy lifting associated with building, running, and managing the containers. Essentially these con-

[1]We refer to our first report specifically about the Bully algorithm.

tainers provide a consistent developing environment curing the "it works on my machine" syndrome.

2.2 Kubernetes

Kubernetes, often abbreviated K8s, is an open-source platform originally designed by Google to automate the deployment, scaling, and operation of containerized applications.

At the heart of Kubernetes' functionality are *pods*. Pods are the smallest deployable units that can be created and managed. A pod consists of one or more containers that share storage and network resources. Pods allow tightly coupled containers to exchange resources while abstracting away granular details at the single-container level. The containers in the pod run based on a specification provided by the user.

Kubernetes uses YAML files for configuration. These files specify the desired state of the system such as which images to use, the number of replicas, and network settings among other things. Kubernetes then continually works towards maintaining this desired state across all pods. For instance, if a pod crashes, Kubernetes will start a new one.

Unlike Docker which works on the level of a single container, Kubernetes widens the scope to multiple containers. Furthermore, Kubernetes does not directly use CRI as the container runtime. Instead, it uses a plugin API for containers known as container runtime interface (CRI). This allows runtimes other than the one used in Docker. However, to plug a new container runtime into Kubernetes one must implement a "shim" that translates requests made by K8s into requests that can be understood by the runtime in question. A generic shim (CRI-O) for all OCI runtimes exists.

2.3 Minikube

Minikube is a lightweight implementation of Kubernetes that creates a VM on the local machine. The virtual machine is then able to deploy Kubernetes cluster nodes. Essentially Minikube provides a simple way of running Kubernetes locally and is particularly ideal for testing purposes. Minikube also has some useful commands such as "Minikube dashboard" which accesses the Kubernetes dashboard (web-based user interface) within the Minikube cluster as opposed to repeatedly using all the CLI-commands.

3 Methods and materials

In this section, we briefly discuss our methods as well as the tools we have used for the project and how they are interconnected.

3.1 System Setup

The project is primarily written in Python (version 3.10.2) using certain common libraries such as *requests*, *asyncio*, and *aiohttp* for processing asynchronous requests and sending responses. We opted to avoid using Dockerhub to store the Docker images. Instead of pulling the image from a remote registry, we created a Minikube cluster to run Kubernetes locally and configure the shell environment to use the Docker daemon inside the Minikube instance as opposed to the daemon on the local machine. Of course, one has to be mindful that the images are then also built within the Minikube instance and not on the local machine. Kubernetes could then be configured to dynamically maintain some desired state by applying the YAML specifications[2].

These specifications define deployments, services, roles, bindings, and service accounts necessary to get the docker container up and running with the desired behavior. These YAML files were added to enable a node port service forwarding all requests directly to the leader, circumventing any requests ending up at the follower pods. The pods are deployed with a custom service account in order to give adequate permission to edit pod labels which tells the service which pod is the current leader. There is also an extra environmental variable in the deployment for fetching the deployed pod's own name, which is necessary in the implementation of label mutation, of the pod's own labels.

3.2 Qualitative Assessment

In this project, we make a qualitative assessment of the Kubernetes-based architecture and the custom leader election algorithm. Key considerations are primarily related to runtime efficiency and tolerance. The evaluations are primarily made through code analysis rather than empirical testing with say memory profilers, message watchdogs[3] or other performance monitoring tools. The goal of this assessment is to demonstrate the resilience and consistent operational stability within the Kubernetes environment. This assessment is not made in a singular place, however, we highlight the pros and cons of our design throughout the report.

3.3 Quantitative Assessment

This section details our approach to quantitatively evaluating the performance of the Kubernetes system. Our assessment particularly involves the downtime when the leader fails as well as the performance (response time) during a load test that bombards the system with requests. The aim of this assessment is to tie concrete numbers to the effectiveness of the Kubernetes deployment. This assessment is made in section 4.2

[2]These contain application-specific configuration. For example, it describes the image-pull policy. In our case, we never want to pull the image from a remote registry, but this could easily be changed without any changes to the source code other than the YAML file.

[3]That is, some service that monitors message traffic

4 Implementations, Experiments and Results

In this section, we briefly discuss our concrete implementations, results from our assessments, and any considerations we have made.

4.1 Implementation

We have extended the provided API skeleton written in Python with new endpoints handling additional aspects of the bully algorithm. Furthermore, we introduced endpoints for serving individual fortunes as well as for displaying the website. This website is a simple HTML page with some in-line Javascript for querying the API on events such as button clicks. It also has some CSS. This entire HTML page is given as a response to get requests on the root URI. To serve fortune cookies we wrote a small FortuneServiceClass which is composed of an instance of a FortuneCookieJar class as well as a get method. The cookie jar has a "category" attribute which can be set to provide fortunes related to a particular category instead of entirely random fortunes. The cookie jar has a method that sends a request to an online API[4] which responds with JSON containing the fortune. The service class' get method simply prompts the cookie jar to send a request with its current category[5] and it then gets the response.

In order to forward all incoming requests to the leader pod, a Kubernetes label approach was chosen. Each pod is given a label with key "leader" that contains value "False" at deployment. When the node's election process converges and a leader is chosen, that leader sets its own leader label to "True", and every pod that received a coordinator message sets its own to "False". This is implemented using the Kubernetes library in Python. Specifically, the configuration is loaded and an API instance is created. The loaded configuration is defined by the pod's service account and secret token which is custom and linked via "read-pods" role binding, to the "pod-reader" role. This gives the pod privileges to read and modify pod labels. When the cluster receives a request it is forwarded to the leader by "bully-service-external", which has a selector for "Leader: True".

[4]http://yerkee.com/api/fortune

[5]A dropdown menu is available in the HTML document.

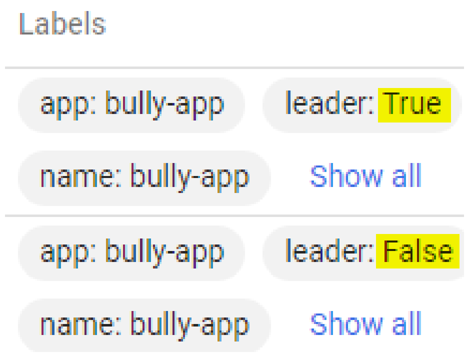


Figure 1: Pod labels showing leader true and false.

That is, "bully-service-external" is a NodePort service that uses a selector to expose only the leader pod to the outside world, providing a single point of entry. In contrast, we also have a "bully-service-internal" which is a headless service, i.e. with ClusterIP set to None which does not provide load balancing or a single IP for accessing the pods. Instead, it enables direct pod-to-pod communication within the cluster. A DNS lookup is performed on the internal service, returning all the pod IP addresses (as it does not have a single IP). The pods can then use these IP addresses for direct communication in the leader election process.

In order for pods to mutate their own labels it was necessary to create a new service account. This is good practice as giving privileges to the default Kubernetes service account can create security issues in larger clusters. In order for pods to authenticate with the Kubernetes API, they need a secret token. This created by Kubernetes itself but is defined in the YAML file "label-secret.yaml". The custom service account "label-sa" does not contain much information in itself, its functionality is created by linking to the "pod-reader" role through the "read-pods" role binding. The "pod-reader" role defines resources and verbs which are allowed for all subjects bound to it. In this implementation the resources are pods and they can be gathered, updated and patched.

4.2 Experiments and results

In this section, we briefly describe each experiment and its associated results.

4.3 Experiment 1: Delete the Leader

We let our system converge and then deleted the current leader pod. This triggers Kubernetes to boot up a new pod to maintain the desired number of replicas, and our leader election algorithm would then find a new leader. We performed this experiment for systems with 5 and 10 pods respectively. We also tried to scale 5 converged pods to 10 pods (introducing 5 new pods to a converged system, which also triggers a leader election). In all cases, it took about 20 seconds for a new leader to be elected. In terms of user experience, this downtime period likewise manifested itself as a roughly 20-second delay. Specifically, during this downtime, we could press the fortune button and it would queue up all the requests for when a new leader was elected to handle it[6].

4.4 Experiment 2: Load Test with HTTP get requests

We created a number of threads, each sending 100 HTTP GET requests to the website endpoint. We compute the average of the response times over the 100 requests in each thread, and finally compute the average over all threads. The results of our preliminary testing can be seen below:

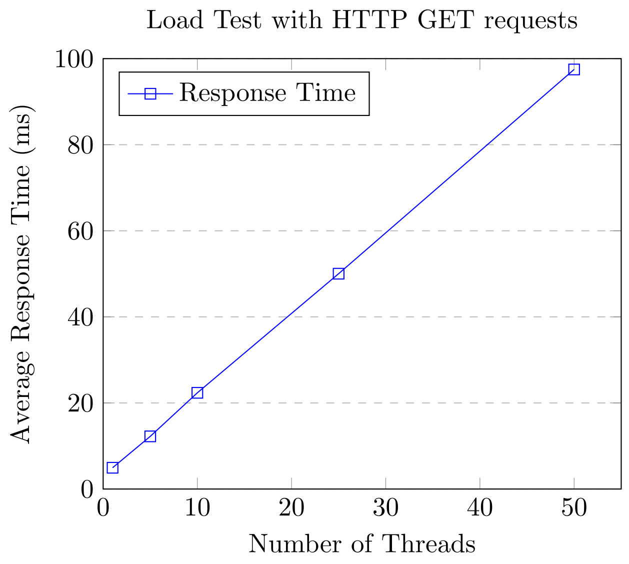


Figure 2: Average response time for varying number of threads

The load appears to scale linearly with the number of threads. This makes sense as all the work is essentially forwarded to a singular leader pod. The response time for the cases with 20 and fewer threads have ideal response times[7]. While the response time is not terrible when there are more threads, the linear correlation highlights that this trend continues and the performance throttling could foreseeably become too much. That is, there is room for improvement. In particular, the burden on the leader pod could be alleviated by leveraging all the other pods for computation too.

[6]In the context of a fortune cookie service, where only one fortune is displayed at a time, this behavior is not really necessary - but it might be handy in other scenarios.

[7]In [2] on page 24, Daniel Jackson claims that most events within a threshold of 30ms result in *perceptual fusion*, where events are indistinguishable from each other. This threshold, however, is obviously context-dependent.

5 Discussion

Our main goal was to demonstrate the usefulness and versatility of the Kubernetes tool for orchestrating complex distributed systems. We have highlighted Kubernetes' strengths in dynamically managing and scaling containerized applications, while also explaining some of the intricacies and inherent challenges of such systems.

Additionally, we also ended up outsourcing the forwarding policy between pods to Kubernetes but this could be handled directly in the code by introducing conditionals that make non-leader pods request responses from the leader. However, this would pollute the codebase a bit more and also introduce a risk of faulty implementations as opposed to relying on Kubernetes which is already a thoroughly tested system.

Of course, a risk of leader election algorithms is that poor implementations can lead to "split-brain" situations where two pods (or more) are elected as leaders simultaneously, each trying to control the cluster and making it unable to coordinate. As leader election algorithms are commonly used in environments that are already quite complex, off-loading parts of the implementation will almost always reduce the likelihood of bugs and errors. In rare cases in our implementation of the custom leader election policy, we found that Kubernetes would mistakenly elect two leaders simultaneously. In our experience, we found that by fiddling with the system parameters (particularly, the sleep statements) we could optimize the convergence speed at the cost of robustness and vice versa. Having long sleep durations would for instance allow the DNS to properly lookup all the associated IP addresses in a timely manner. Lowering the sleep durations would drastically increase the number of messages emitted; sometimes this would cause two pods to announce their leadership at the same time. Of course, unless the system is known to be particularly volatile, one would not expect the leader pod to fail frequently - in which case it might be better to ensure stability rather than speed when maintaining the desired state. Ways of handling the "split-brain" syndrome could be 1) to introduce a more strict leader election policy or 2) occasionally leaders should query all the other pods to ensure it is the only leader. If the pod is not the only leader in the latter case, it should start a new election[8].

Arguably, this project is quite an artificial demonstration of Kubernetes in the sense that pods are commonly used to distribute and otherwise manage the work across a cluster as Kubernetes has built-in support for load-balancing and horizontal scaling (of the number of replicas) based on the workload; yet, all non-leader pods simply forwards their work to the leader. There is no load-balancing and the leader can potentially drown in requests. For example, section 4.2 highlights how response time scales linearly with the number of requests[9].

[8]We saw that starting a new election would cause the system to converge in our testing.

[9]Of course, there is a limit to how much parallel processing that can be done with a finite number of pods

6 Conclusion and perspectives

We have demonstrated the capabilities of Kubernetes in managing and orchestrating a distributed web service. By integrating the custom leader election algorithm and relying on Kubernetes' dynamic environment, we were able to create a *resilient* and *scalable* fortune cookie service. Despite some challenges like the potential for "split-brain" scenarios or the linear scaling of response times under load - both of which we either propose a solution for or are simply a consequence of having a simple pod do all the work - the project highlights how Kubernetes can be used to maintain some desired system state. Future work could explore refinements to the leader election algorithm to enhance robustness or load-balancing strategies to enhance the response time.

References

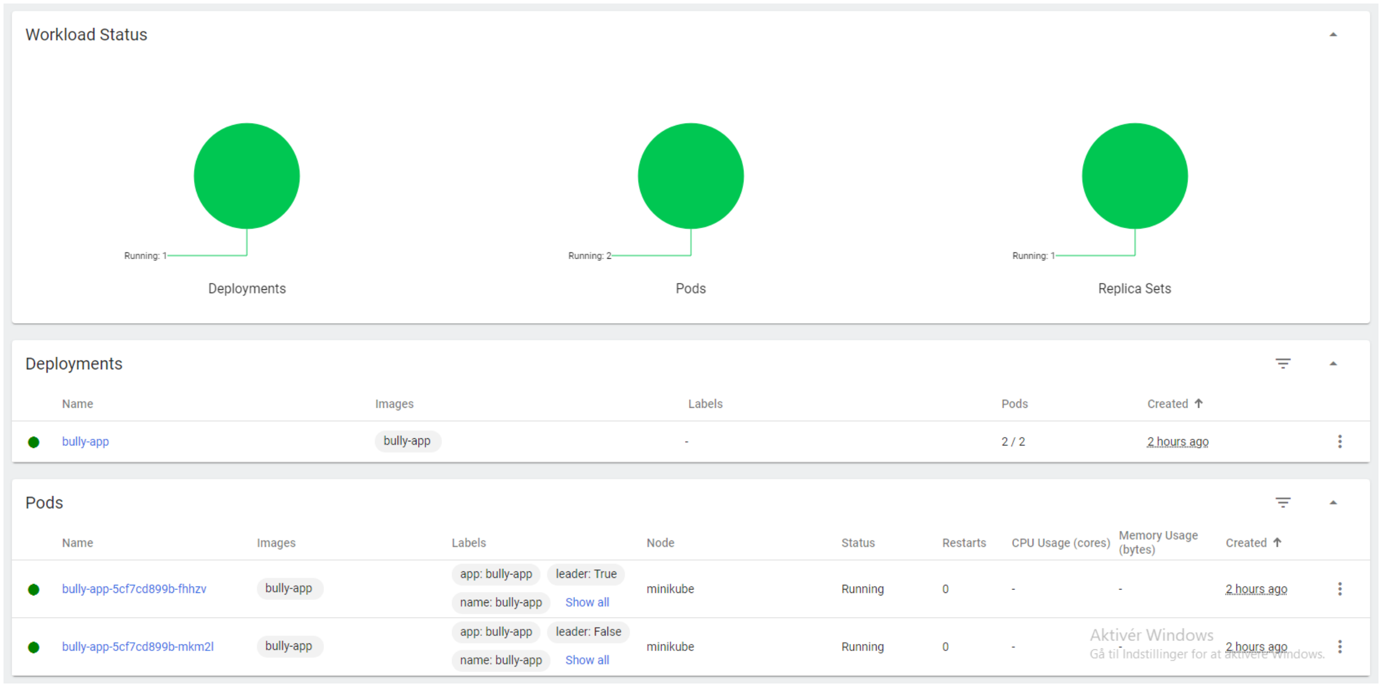
[1] Garcia-Molina. "Elections in a Distributed Computing System". In: *IEEE Transactions on Computers* C-31.1 (1982), pp. 48–59. DOI: 10.1109/TC.1982.1675885.

[2] Daniel Jackson. *The Essence of Software: Why Concepts Matter for Great Design*. Princeton University Press, 2021. ISBN: 9780691225388. URL: http://www.jstor.org/stable/j.ctv1nj340p (visited on 05/09/2023).

[3] *Simple Leader Election with Kubernetes*. https://kubernetes.io/blog/2016/01/simple-leader-election-with-kubernetes/. Accessed: November 2023. 2016.

A Appendix: Screenshot of Dashboard

Example of a Kubernetes cluster with leader labels running. We only have two pods running in this case to keep the picture compact.



Deployments

	Name	Images	Labels	Pods	Created ↑
●	bully-app	bully-app	-	2 / 2	2 hours ago

Pods

	Name	Images	Labels	Node	Status	Restarts	CPU Usage (cores)	Memory Usage (bytes)	Created ↑
●	bully-app-5cf7cd899b-fhhzv	bully-app	app: bully-app leader: True name: bully-app Show all	minikube	Running	0	-	-	2 hours ago
●	bully-app-5cf7cd899b-mkm2l	bully-app	app: bully-app leader: False name: bully-app Show all	minikube	Running	0	-	-	2 hours ago

Figure 3: Kubernetes cluster dashboard