Assignment 3: Report

Final Deployment of ChatHub

6th of June 2024

Authors

103154	João Fonseca
103183	Diogo Paiva
103600	Guilherme Antunes

Professor

Prof. Dr. João Paulo Barraca

Management of Computation Infrastructures - 2023/24

MSc in Informatics Engineering

University of Aveiro

Table of contents

Table of figures

1 Introduction

1.1 Purpose of the report

The purpose of the report is to document the work developed by our group, for the second assignment of the Management of Computation Infrastructures course (2023/24), where we were proposed an initial deployment of the product we chose to operate, ChatHub, our rebranding of TailChat.

1.2 Technologies used

We used Docker to build custom images of the components and used the DETI registry to store them. To deploy the entire infrastructure – static website and ChatHub components - we used K3s [1], a lightweight distribution of Kubernetes, along with Longhorn [2], a cloud native distributed block storage for Kubernetes, all provided to us by the course professor. To centralize logs from our infrastructure, we used Fluentd [3] to collect and redirect them from our pods to a Rsyslog [4] server. Finally, to monitor performance metrics such as CPU and memory usage, we used Open Telemetry [5] to collect and export these metrics to Prometheus [6], that is used by Grafana [7] to create visual representations of this data.

1.3 Code repository

We used a GitHub repository [8] to store the specification of the Docker images and K3s deployments, as well as some scripts to automate the processes of building the Docker images, and deploying the infrastructure. There are README files with some indications on how to use the contents of the repository.

2 Description of the full cluster operation

2.1 Automatic deployment

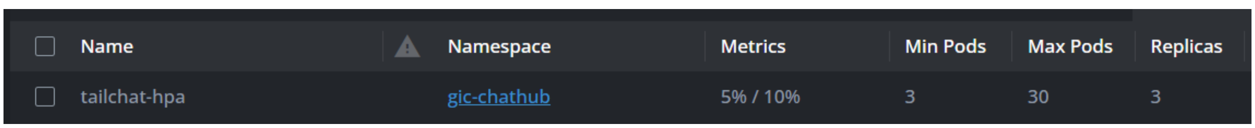
To achieve an automatic deployment we have two files, one to put in the registry all the docker images (`docker-all.sh`) and another to deploy the infrastructure in the Kubernetes cluster (`deploy-all.sh`).

However, our deployment is not fully automated since Mongo needs some manual configurations because we were not allowed to use operators. These manual configurations are explained step by step in the file `mongo-config.md`.

2.2 Autoscaling

2.2.1 TailChat Application

For our main application, we have defined a Horizontal Pod Autoscaler that will aim to maintain the CPU consumption percentage at around 10%, to keep resource usage low during periods of lower activity, with the ability to scale from 3 to 30 pods. Typically, with no active users, the system consumes about 5% of CPU with the minimum number of pods.



Name		Namespace	Metrics	Min Pods	Max Pods	Replicas
tailchat-hpa		gic-chathub	5% / 10%	3	30	3

Figure 1. Autoscaler status showing current consumption and limit for launching new instances.

2.2.2 Other Services

We did not apply additional autoscalers to the remaining services, Minio, MongoDB, and Redis. In the case of Minio, the number of replicas is configured manually, so automatically scaling this number would result in unused instances, as it would be necessary to manually redefine the number of available replicas in the configuration of each service. MongoDB also requires that replicas be added one by one and connected to the master instance, which stays the same, making it impossible to apply autoscaling in this case. Finally, the Redis instances are connected to the Sentinel instances, and increasing the number of one would require increasing the number of the other. This would not be feasible using only the resources available for this project.

2.3 Health checks

2.3.1 Redis

To perform periodic checks on the status of the Redis pods, we opted to perform a verification directly within each pod by executing the "ping" command, which should return "pong" if it is functioning correctly. After some testing, we determined that the ideal waiting time to start the checks should be around 15 seconds, and these checks should be performed every 5 seconds with a timeout of 5 seconds. Up to three failures are accepted before restarting the pod, and a single successful execution is enough to determine that the pod is functioning correctly.

2.3.2 Mongo

For MongoDB, like Redis, direct execution of a ping command was defined to verify if the pod is working correctly. It was also defined that the checks should start after 15 seconds of execution, maintaining the remaining time limits. However, since we are dealing with a database that is intended to store all messages sent by users, which is the focus of the system, we defined only a tolerance of two consecutive failures before considering that the pod is not in proper working condition. However, in case of failure, it is not defined that the pod restarts since if the failure occurs in the master instance, it is not possible to recover automatically from that failure as the configuration must be done manually.

2.3.3 Minio

To verify the proper working of the Minio instances this system has two dedicated endpoints for this purpose: *"/minio/health/live"* for livenessProbe and *"/minio/health/ready"* for readinessProbe. Calls to these endpoints are made with a periodicity of 10 seconds, as it serves primarily as a repository for large files, such as images, which are not accessed as frequently during application usage. Since Minio takes less time to start, the checks begin after 10 seconds of execution. We also defined a 5-second verification timeout and a tolerance policy of up to three consecutive failures to restart the problematic instance.

2.3.4 TailChat Application

For the TailChat application, we followed the recommendations of the developers who also reserved the endpoint *"/health"* for readiness and liveness checks. We also adhered to the developers' recommendations for the verification intervals, setting a waiting time of 10 seconds before starting the checks, intervals of 30 seconds between checks, a timeout of 2 seconds, and the same tolerance of up to three consecutive failures to consider the instance as malfunctioning.

2.4 Redundancy and disaster recovery

2.4.1 Redis

We tried achieving redundancy in two separate ways.

The first approach was using Sentinels, they provide a high availability (HA) setup for Redis, monitor the health of the Redis master and slave instances, ensuring that the system remains operational even if the master node fails. And the part of HA and of monitor the Redis master and slaves instances works well but when using Sentinels we have to pass to the Redis client in the application the service with all the Sentinels instances and we tried doing that but the Redis client has to know that are Sentinels and for that we had to chance the code (The error that the application was showing is: `ERR Only HELLO messages are accepted by Sentinel instances`).

The second approach was using a Cluster, it provides a robust solution for scaling Redis horizontally across multiple nodes, ensuring high availability and fault tolerance. Its automatic sharding and replication features make it suitable for high-performance and large-scale applications. But with this approach we found the same problem, the Redis client needs to know that we are passing a Cluster and for that we had to change the code (The error that the application was showing is: `ReplyError: MOVED 10495 10.42.0.165:6379`).

We end up using the first approach with 3 replicas (3 Redis instances and 3 Sentinel instances) but instead of passing the Sentinels service we had to pass the master directly which means that if the master goes down and the other instances select another instance to become the master the application will fail because it will try to do write operations in a slave.

2.4.2 Mongo

In the case of Mongo, we achieve redundancy but with a problem caused by using manual configurations.

We launch three replicas of Mongo and to turn the first replica in the master we must do some manual configuration that consists of setting the primary server and adding the other 2 Mongo instances.

This means that Mongo turns into a master slave architecture and unlike Redis we can pass the service with all the Mongo instances since if we try to do a write in one of the replicas the replica will reroute the write to the master.

Now for the problem, despite Mongo turning into a master slave architecture if the master instance goes down the slaves don’t select another instance to turn into the master thus if the master node goes down for some seconds the application may not work as expected (we couldn’t test this since after turning down an instance it automatically comes up again).

2.4.3 Minio

In the case of Minio we have 3 replicas, we didn't had any problems since any replica of Minio can handle writes and reads so we passed the service containing all the Minio instances to the TailChat application and it worked perfectly and if a Minio instance goes down the other 2 are still available to the application for any operation.

2.4.4 TailChat Application

As for the TailChat application we also have three replicas so that if one instance goes down we still have two more. And is fault tolerating since if one instance goes down the end user will not realize it and no information are going to be lost.

We also have the same number of load balancers and application instances in Kubernetes as this can offer significant advantages in terms of traffic distribution, fault tolerance, scaling, security, and overall system management. This approach aligns resources directly with application needs, providing a balanced, efficient, and manageable infrastructure.

2.5 Load Balancer

The load balancer was achieved by using a Nginx server. Despite already having a Nginx docker image for the static website, we had to create another one. This is because if we pass the conf file through a secret (as in the static website), the TailChat application was giving us an error. After we created another image with the conf file already inside, the application worked as expected.

In the configuration for the Nginx server, in addition to the `proxy_pass http://tailchat:11000/`, we had to configure three more things for the server to support Web Sockets:

- `proxy_http_version 1.1`
- `proxy_set_header Upgrade $http_upgrade`
- `proxy_set_header Connection "upgrade"`

We configured the Nginx as a deployment with the same number of replicas as the TailChat application and created a service that then we passed to the ingress to be able to connect to the TailChat application.

2.6 Affinity

Affinity in Kubernetes is a powerful tool that enhances the scheduling capabilities of the cluster, leading to optimized performance, improved reliability, better resource utilization, and cost efficiency, because of that we decided to add some affinity policies.

We decided that all the components that are directly connected to the TailChat application must be in a node where it exists an instance of this application. To be able to achieve this and considering that the TailChat application is the last thing that is deployed we had to start in the first component that is deployed, Minio, and we set a policy of podAntiAffinity to match any Minio app. Then for the other components we set a policy of podAntiAffinity to match the same component and a policy of podAffinity to match a Minio app.

The instances of Nginx that serve as load balancers also have policies of affinity to be in the same node as the TailChat applications.

In the future one thing that we liked to do but did not find how we can do it is, make the TailChat application always made reads from the component that is in the same node as it.

fluentd-n6m48	kub05	gic-chathub	1 / 1	Running
minio-1	kub05	gic-chathub	1 / 1	Running
mongodb-replica-2	kub05	gic-chathub	1 / 1	Running
nginx-569c7fcb7f-8wz	kub05	gic-chathub	1 / 1	Running
otel-collector-opente	kub05	gic-chathub	1 / 1	Running
prometheus-87cd55b	kub05	gic-chathub	1 / 1	Running
redis-1	kub05	gic-chathub	1 / 1	Running
rsyslog-5c7694d888-z	kub05	gic-chathub	1 / 1	Running
sentinel-0	kub05	gic-chathub	1 / 1	Running
tailchat-1	kub05	gic-chathub	1 / 1	Running
website-5598d6ddfc-	kub05	gic-chathub	1 / 1	Running

Figure 2. Demonstration of affinity policies on node "kub05".

2.7 Integrated monitoring and observability of product operation

2.7.1 Centralized logging

Centralized logging was achieved by combining Fluentd with a Rsyslog server. Fluentd is configured as a Daemon Set, which means there is one instance running in each node of the cluster. Each instance of Fluentd can collect system logs from every pod in the respective node and forward them to the Rsyslog server. The Rsyslog server is configured to receive these logs from Fluentd and filter them according to the namespace they originated from. In our case, we want to store logs coming from the "gic-chathub" namespace and discard the remaining ones. Once filtered, these logs are stored in a Persistent Volume Claim with a capacity of 500 Mi, mounted on the Rsyslog server `/var/log` directory. Log files are discriminated by container names, and each newly received entry is appended to the respective log file under `var/log/remote/kubernetes`.

To view these logs, one should connect to the Rsyslog server pod using the command **kubectl exec -itn gic-chathub <rsyslog-pod-id> -- sh**, and move to the directory where the logs are stored, as seen in Figure 1.

```
vagrant@ubuntu-jammy:~$ kubectl exec -itn gic-chathub rsyslog-5c7694d888-zwz2h -- sh
/ # cd /var/log/remote/kubernetes/
/var/log/remote/kubernetes # ls -l
total 1544
-rw-r--r--    1 root     root           876 Jun  5 21:38 gic-chathub_grafana.log
-rw-r--r--    1 root     root          7301 Jun  5 21:43 gic-chathub_mongodb.log
-rw-r--r--    1 root     root        164301 Jun  5 21:42 gic-chathub_opentelemetry-collector.log
-rw-r--r--    1 root     root         10187 Jun  5 21:42 gic-chathub_prometheus.log
-rw-r--r--    1 root     root           877 Jun  5 21:36 gic-chathub_rsyslog.log
-rw-r--r--    1 root     root       1380707 Jun  5 21:45 gic-chathub_tailchat.log
/var/log/remote/kubernetes # tail -n 1 gic-chathub_mongodb.log
2024-06-05T21:43:59+00:00 fluentd-8w8l4 fluentd: host:stdout#011ident:F#011message:43:54.473+0000 I NETWORK  [LogicalSessionCacheRefresh] St
arting new replica set monitor for rs0/mongodb-replica-0.mongo:27017,mongodb-replica-1.mongo:27017,mongodb-replica-2.mongo:27017#011docker:{
"container_id"=>"b4b68296ea54ff7af1fb14927959dbe16e0e987a6c22b5a345d9199a1fb49372"}#011kubernetes:{"container_name"=>"mongodb", "namespace_n
ame"=>"gic-chathub", "pod_name"=>"mongodb-replica-2", "container_image"=>"registry.deti/gic-chathub/mongo:latest", "container_image_id"=>"re
gistry.deti/gic-chathub/mongo@sha256:ecd3182faf3533bbbf88c40882cf828016ce52145b5003e9028d7a6a57983bb3", "pod_id"=>"59fdc0e8-6a1b-4fad-bc39-5
a7fc3fb7ec6", "host"=>"kub04", "labels"=>{"app"=>"mongo", "controller-revision-hash"=>"mongodb-replica-66979c67db", "selector"=>"mongo", "ap
ps_kubernetes_io/pod-index"=>"2", "statefulset_kubernetes_io/pod-name"=>"mongodb-replica-2"}, "master_url"=>"https://10.43.0.1:443/api", "na
mespace_id"=>"58d804c2-4c3d-486f-9af7-46b1c69ce183", "namespac
/var/log/remote/kubernetes #
```

Figure 3. Viewing the logs from MongoDB in the Rsyslog server.

2.7.2 Performance metrics

Performance metrics were obtained using Open Telemetry, which runs as a Daemon Set in every node of the cluster. It is configured with the help of Helm, a package manager for Kubernetes [9]. The Open Telemetry receiver collects metrics from the host every 10 seconds and exports them to Prometheus using the remote write functionality, which pushes the metrics directly instead of waiting for Prometheus to scrape them. To allow for this behaviour, we used Prometheus v2.33.3, and enabled the remote write receiver by passing it as an argument in the container specification (**--web.enable-remote-write-receiver**). To create and view charts with the chosen metrics under analysis, we used Grafana and setup Prometheus as the data source. The metrics we monitored were the CPU usage, Memory usage and Filesystem usage, which would allow us to monitor the usage of resources of the system. Another relevant metric is the number of established connections, which would let us know how many users were using ChatHub. The visualizations created in Grafana can be observed in Figure 2, proving that the performance metrics pipeline was successfully implement.

To view the Grafana dashboard, one should access http://grafana.gic-chathub.k3s with the login credentials **admin/admin**. If you are using a local instance of Grafana, the dashboard JSON is available in our repository (**grafana-dashboard.json**). Do not forget to configure Prometheus as the data source, http://prometheus.gic-chathub.k3s.

However, this monitoring setup should only be seen as a proof of concept, since it comes with many limitations, that are discussed in greater detail in the Limitations section. Nonetheless, we believe that overcoming these problems could be achievable by using and modifying the current metrics pipeline.

Figure 4. Performance metrics charts in Grafana.

2.7.3 Limitations

Centralized logging	**Performance metrics**
The logs should be filtered at the source by Fluentd instead of the destination by Rsyslog. This would significantly decrease the amount of information being transported in the cluster and distribute the filtering among Fluentd instances instead of centralizing it on the Rsyslog server.	The current metrics do not include a label with the pod's name; therefore, we are not able to monitor a specific component of our application. The attribute this issue to the lack of documentation and online guidance over this subject, given the novelty of these tools.
The logs could be further discriminated by the specific instance of the component where they originated, instead of being grouped in the same file.	We are not filtering the specific metrics we want to collect; therefore, we are also storing metrics that are not relevant in our case.
The size of the log files is not bounded to any limit. Once the storage capacity of 500 Mi is reached, the Rsyslog server simply stops saving the received logs.	There is the potential to use Open Telemetry to also collect logs from the application, allowing the use of a single technology to collect logs and performance metrics. We would not need to use Fluentd and Open Telemetry.
There is no visually appealing user interface put in place to query and filter the stored logs. Everything should be done using the shell, which can be challenging for unexperienced users.	

3 Conclusion

In this assignment, we were able to finalize the deploy an existing application using the Docker technology and a K3s cluster. We implemented additional features in comparison to the previous deployment, such as autoscaling, health checks, redundancy and disaster recovery, load balancing and affinity policies to our services. These features were put in place to make our deployment more robust.

The automatic deployment was already put in place in the previous deployment; however, we extended it with the newly added components (Fluentd, Rsyslog, Open Telemetry, Prometheus and Grafana). The creation and push of Docker images, and the deployment of the infrastructure in the K3s cluster is automated using shell scripts.

We were able to configure the collection and centralization of logs in a Rsyslog server using Fluentd, and the collection of performance metrics using a combination of Open Telemetry, Prometheus and Grafana. However, we pointed out some of the limitations with both features, which would certainly have to be addressed as future work.

References

[1] "K3s," [Online]. Available: https://k3s.io/.

[2] "Longhorn," [Online]. Available: https://longhorn.io/.

[3] Fluentd, "Fluentd," [Online]. Available: https://www.fluentd.org/.

[4] Adiscon GmbH, "Rsyslog," [Online]. Available: https://www.rsyslog.com/.

[5] "Open Telemetry," [Online]. Available: https://opentelemetry.io/.

[6] "Prometheus," [Online]. Available: https://prometheus.io/.

[7] "Grafana," [Online]. Available: https://grafana.com/.

[8] G. Antunes, D. Paiva and J. Fonseca, "GitHub Repository - Deployment 2," [Online]. Available: https://github.com/GIC-Assignment-ChatHub/deployment2/.

[9] "Helm," [Online]. Available: https://helm.sh/.