SDCI Report

Introduction

In modern distributed systems, ensuring the efficient handling of traffic, especially under high load, is critical for maintaining the integrity of data flows. In this report, we discuss the steps taken to address an issue in an application architecture using Kubernetes and Istio. Our aim was to manage data flow effectively while prioritizing critical information and ensuring smooth communication between components in the system. When working on a project in this domain, it is important to note the difference between application developers and network and application operators. For this project we aimed to take the position of operators rather than developers.

The GitHub containing the full code described in this report and used in this project, can be found at this link: https://github.com/EmilieGreaker1/project-SDCI.git

Design

Nominal Phase

The project revolves around a system that consists of an application that communicates with a server, which in turn interacts with an intermediate gateway (GI). The GI is responsible for handling communication with final gateways (GFs) connected to devices across different zones. For this project, we had a total of three zones with each zone connected to only one device for simplification. However, it could be imagined that each zone could handle multiple devices such as illustrated in the figure below. Figure 1 illustrates the initial setup of the system, including its components and original routes between these.

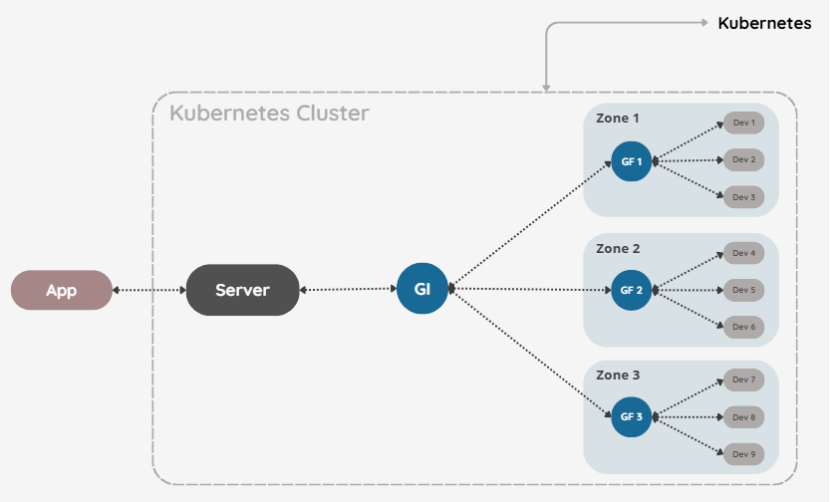


Figure 1: The initial setup of the system

Incident Phase

For this project, we imagined that the issue in the application architecture was that the GI was saturated. We also considered that the data coming from Zone 1 was very critical data that we could not afford to lose or compromise on. However, in the case of saturation, where the GI was overwhelmed by the incoming traffic, there was a risk of potential loss of this high-priority data. This saturation was intensified by the simultaneous communication from multiple zones, not all of which were critical. Our primary goal was to ensure that critical data continued flowing while non-critical data was managed more efficiently.

There are several ways to achieve this in a Kubernetes environment, however for this project we decided to focus on trying to solve this problem by implementing this specific use case: “Adaptation of application flows”. In other words, our goal was to adapt the flow of packages from the non-critical zones to reduce the amount of packages going through the GI in a way that was transparent to the components of the system.

Adaptation Phase

To address the issue of saturation, we chose to implement a dynamic strategy to manage the traffic. We developed a “flow reduction service” for non-critical zones. This service reduces data volume by averaging the values of two incoming packages from the same device and sending just one of them to the GI. This means that the flow of packages from the non-critical zones is halved when this service is active while data from the critical zone will be processed as usual. Figure 2 below illustrates how the packages from non-critical zones are redirected through the flow reduction service when the adaptation is active. To ensure that this adaptation is applied when needed, we had to find a way to monitor the GI.

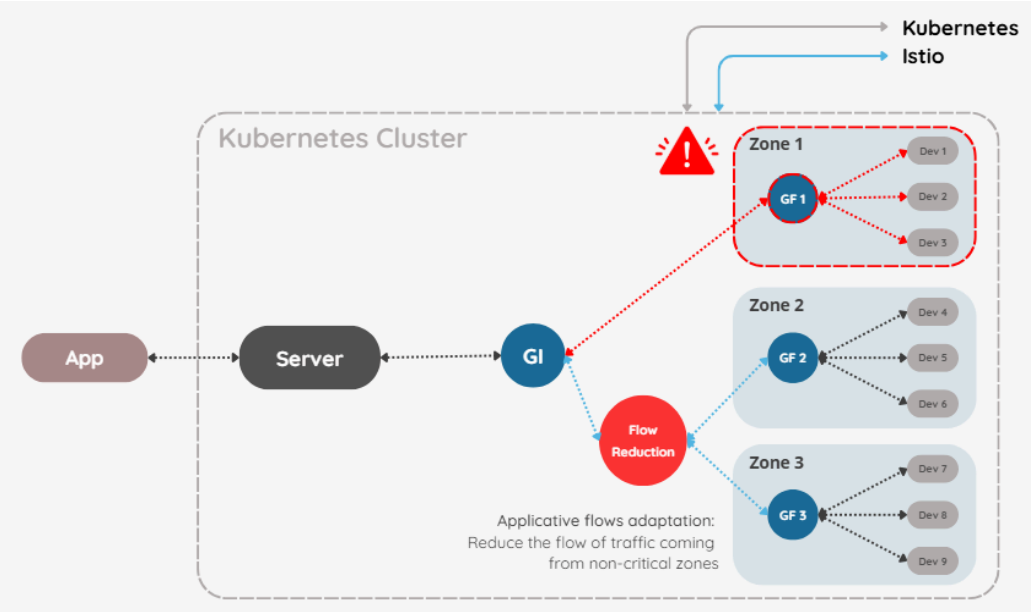


Figure 2: Packages from non-critical zones are directed through the flow reduction service while critical data from Zone 1 is passed directly to the GI

Monitoring Phase

We wanted to be able to continuously monitor the GI's state using native Kubernetes. To achieve this we used Kubernetes native metrics API. We focused on monitoring the CPU usage of the GI to determine if it was saturated. We also made sure to monitor the status of the flow reduction service, whether it was up or not, to know if it needed to be deployed or deleted.

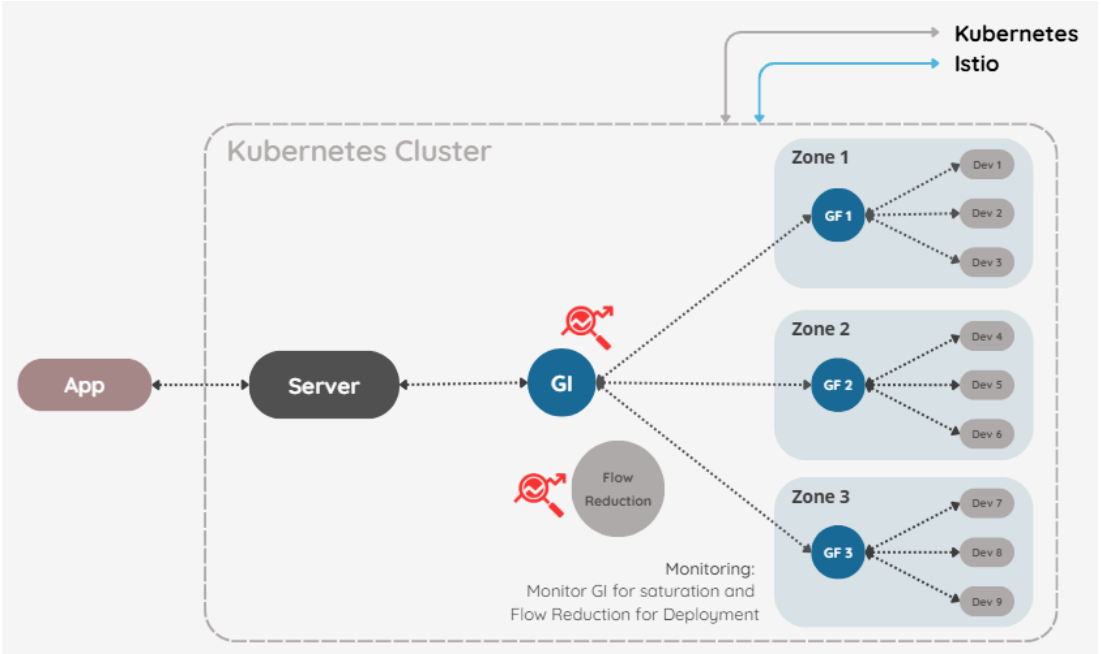


Figure 3: The intermediate gateway and the flow reduction service are being monitored

MAPE-K Loop

To enhance the system's autonomy, we implemented a simplification of the MAPE-K (Monitor, Analyze, Plan, Execute, Knowledge) loop in a python script, which was our general controller. While the MAPE-K loop traditionally includes a "Knowledge" component, our implementation did not require one considering the limited functionalities we aimed to implement, meaning we implemented a MAPE loop for our project. This MAPE loop autonomously monitors the system's health using the methods explained in the monitoring phase of this report and, using his knowledge, analyzes the results of the monitoring, plans which actions to take based on these results, executes these plans, and updates the system accordingly.

By incorporating the MAPE loop, the system can autonomously adjust to changing conditions without the need for manual intervention. The system is designed to imitate a level 3 autonomic computing state, meaning it can make decisions and take actions on its own based on the analysis of real-time data.

Diagrams to provide a clear visual representation of our system's objectives and functionalities are included in the appendix.

Implementation

Deploying The Initial System

The first step in this project was setting up the initial system in a Kubernetes cluster. We achieved this by creating Dockerfiles to build docker images for each component by using the provided .js applications and uploading these images to our Dockerhub.

We then created and applied deployment YAML files and ClusterIP service YAML files for each component of the system which, using the aforementioned docker images, deployed the components in the Kubernetes cluster and equipped each component with an internal IP that can be recognized within the cluster. Once the initial setup was complete, we tested that it was working as expected by viewing the logs of the server and verifying that the server was receiving packages from the devices from each zone. We could now move on to the next phase of the project in which we implemented the adaptation use case.

Implementing the Adaptation

The flow reduction service we developed was a key part of our solution. This service reduces the number of non-critical data packages by averaging the values of two incoming packages and sending just one. By cutting the number of non-critical packages in half, the system reduces the overall traffic load.

We created the Dockerfile for this service, built the docker image and deployed it in the cluster using Kubernetes. To make this service accessible within the cluster, we applied a ClusterIP to it. We also created an Istio virtual service which, when active, reroutes the packages from the non-critical zones towards the flow reduction service and onward to the intermediate gateway. This virtual service also ensures that the rerouting of packages remains transparent to the components of the system.

When the full adaptation is active, the flow reduction service is scaled up to 1 using Kubernetes and the rerouting virtual service is applied. When the implementation is disabled, the flow reduction service is scaled down to 0 and the rerouting virtual service is deleted, thus reverting the routing of packages from non-critical zones back to the initial routing directly towards the GI. When active, this approach effectively reduces the overall traffic from non-critical zones without compromising the data integrity of critical zones, ensuring that the GI is not overwhelmed. Kubernetes played a central role in deploying the flow reduction service, while Istio allowed us to reroute traffic based on real-time demands.

The key aspect of this solution is dynamic adaptation. Depending on the need, we can deploy or stop the flow reduction service. This makes it great to respond to real-time challenges. Additionally, the solution can be implemented with no downtime, ensuring the continuous operation of the applications. Since this solution is transparent, there is no need for additional configuration or changes to the existing applications, making it easy to integrate.

To verify that the adaptation had been implemented correctly, we looked at the flow of packages that reached the server after applying the adaptation. From this, we could verify that the packages coming from devices from the non-critical zones contained data that was calculated as the median between two packages from one device. We also noticed that the packages from these zones were coming in half as often in comparison to the packages from Zone 1. At this point, we had correctly implemented the adaptation manually. We then aimed to automate this adaptation.

General Controller

The core of the system is the General Controller, which oversees the entire process. It is responsible for managing different phases of the system, including monitoring, adaptation, and the MAPE loop. The MAPE loop ensures that the flow reduction service is deployed when necessary and that the system adapts dynamically and automatically to changing conditions.

We began by implementing the adaptation phase following the MAPE logic. Given a RFC providing information on the system's status, the planning phase determines the appropriate action - whether to reduce flow, reset it, or take no action. The execution function then carries out the chosen plan, enabling the adaptation phase to run automatically. We then focused on the monitoring phase.

Implementing the Monitoring

For the system to react appropriately to the case of GI saturation, continuous monitoring of the GI was necessary. In this phase, the General Controller continuously communicates with the Kubernetes API to retrieve real-time data, including the CPU usage of the GI and the status of the flow reduction service. If the CPU usage exceeds 80%, indicating saturation, the controller deploys the flow reduction service implemented in the adaptation phase, if not already up. If the system is not saturated, the controller ensures that resources are saved by terminating the service.

To determine whether the flow reduction service is running, we check if at least one pod is available for the service. To calculate the percentage of CPU usage for the intermediate gateway (GI), we first gather data on the CPU consumption across its pods and convert it into cores. We then compare this with the total CPU requested by the GI's containers. Since container CPU allocation is not always exact, we specify both a requested amount and an upper limit to ensure that the allocated CPU remains close to the requested value. This approach allows us to estimate CPU usage accurately.

If there is a change in the flow reduction service status or the CPU usage percentage, we increment an alert counter. This counter helps determine whether an action should be triggered in the MAPE loop, ensuring the system adapts efficiently to varying conditions.

User Interface

To facilitate system management, we developed a simple user interface using the Flask Python web framework [Figure 4]. This interface allows users to toggle the different phases of the system: the full MAPE loop, the monitoring phase, and the adaptation phase. It provides better visibility into the system's operation and allows for more effective management of the flow reduction process.

When monitoring, it displays whether the flow reduction service is deployed or not, the number of alerts and the percentage of CPU being used. Its functionalities are further detailed in the upcoming Demo section.

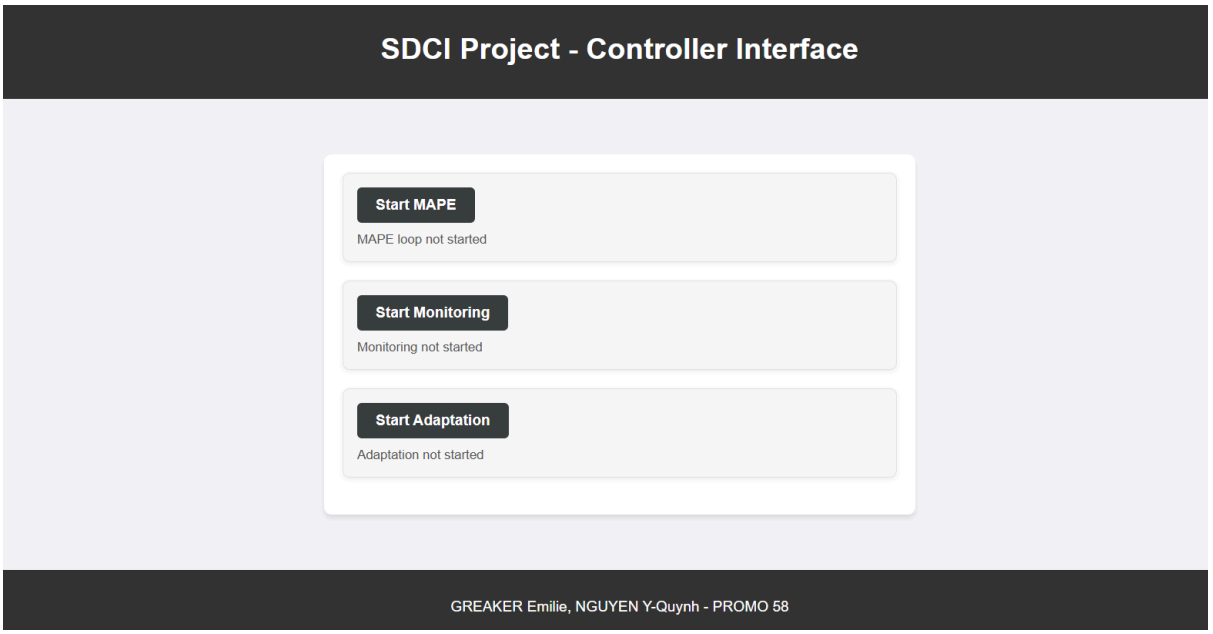


Figure 4: The Python Flask user interface

Demo

The following demo was carried out through the user interface previously mentioned.

If we start off by simply starting the monitoring by itself, we can see the information on the two components that we are monitoring; the flow reduction service, and whether or not it is up, and the CPU usage percentage of the GI [Figure 5]. Initially we can verify that the flow reduction service is down and the value for the CPU usage percentage will change and vary, usually in the interval 8-11 (~9.4 in Figure 5). We can also see that as time goes on, certain alerts will be raised. These alerts reflect changes in the CPU usage percentage of the GI. As the monitor itself does not execute any action to resolve these alerts, the alert number will keep incrementing over time.

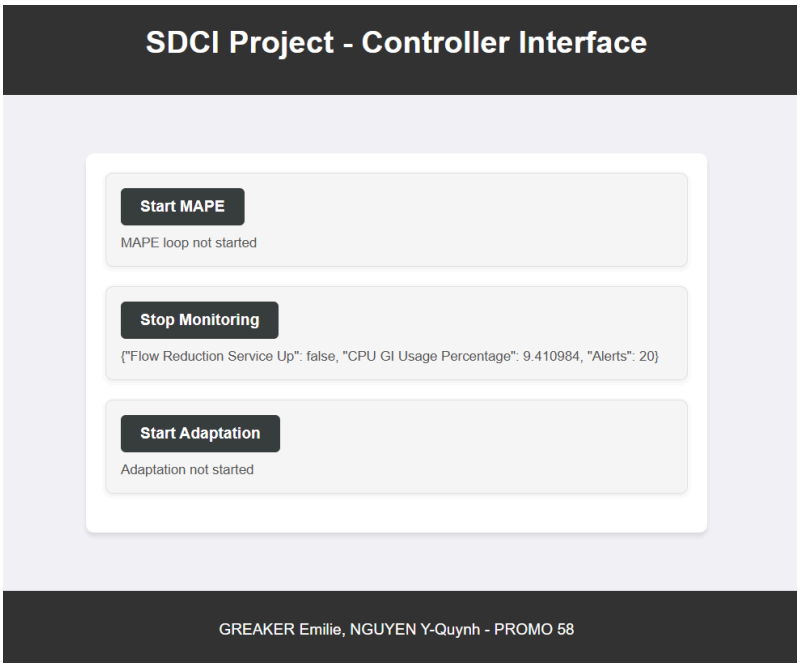


Figure 5: User interface when only the monitor is active

From now we manually start the adaptation of the system and pay attention to the monitored values [Figure 6]. First, we can see that the “flow reduction service up” boolean will be set to “true”, indicating that the system has indeed applied the flow reduction service. After this we can start paying attention to the values of the CPU usage percentage of the GI. From these values, we will notice after a bit of time that the values will gradually drop and vary between values in the rough interval 5-8 (~5.55 in Figure 6).

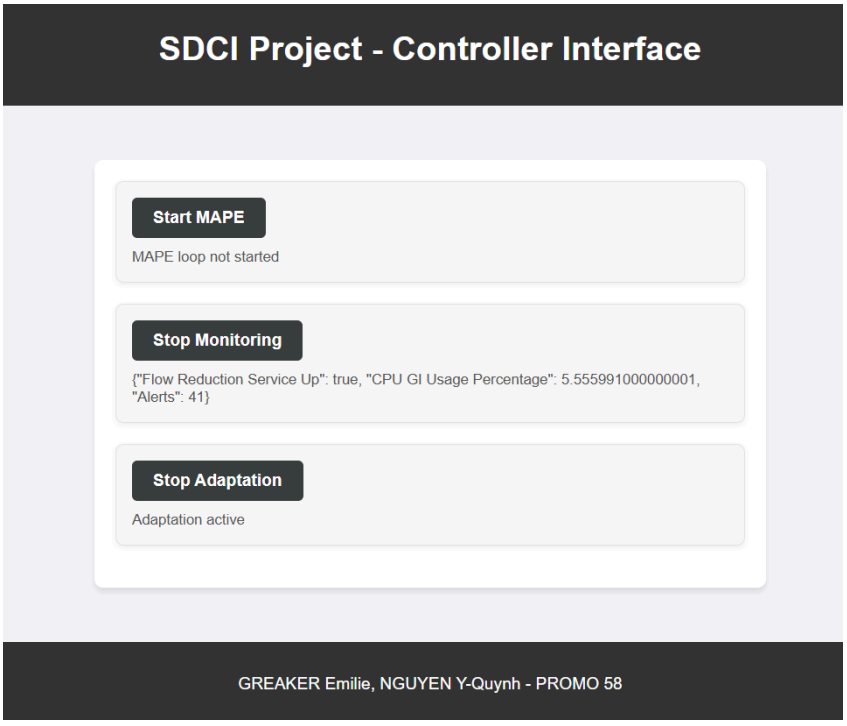


Figure 6: User interface when the monitor and the adaptation is active

Lastly, if we turn off the monitoring and the adaptation and start the full MAPE loop, we can see the power of the MAPE loop to automatically react to alerts and modify the system if needed [Figure 7]. As part of this, we can see that the monitor of the MAPE loop will raise alerts from time to time to indicate that the CPU usage percentage of the GI has changed, but that these alerts will be automatically handled and resolved by the full loop. As we are not really dealing with saturation of the GI, as this was only a hypothetical problem, the CPU usage percentage of the GI will not reach 80% usage and the MAPE loop will therefore not recommend the system to apply the flow reduction service. However, if we start the adaptation manually from the user interface, we can see that the MAPE loop will detect that the adaptation is on for a brief second before it automatically turns it back off as it will conclude that the adaptation is not needed at this time (not shown in the figure).

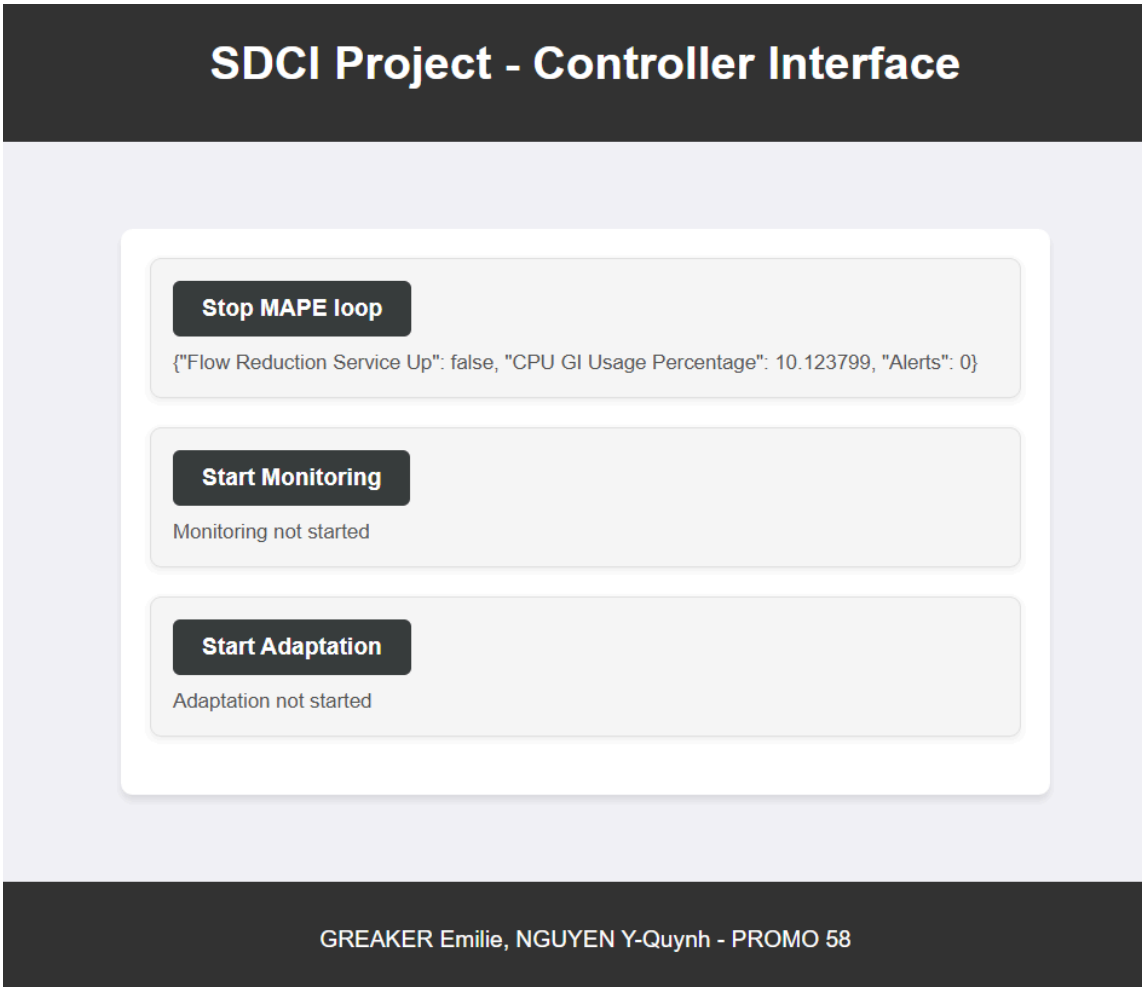


Figure 7: User interface when only the MAPE loop is active

Conclusion

In conclusion, our approach successfully addressed the challenge of managing data flows in a system with an intermediate gateway at risk of saturation. By using Kubernetes and Istio, we implemented a self-adapting system that prioritizes critical data while dynamically managing non-critical data. The General Controller and MAPE loop allow the system to operate autonomously, adjusting to real-time conditions without manual intervention.

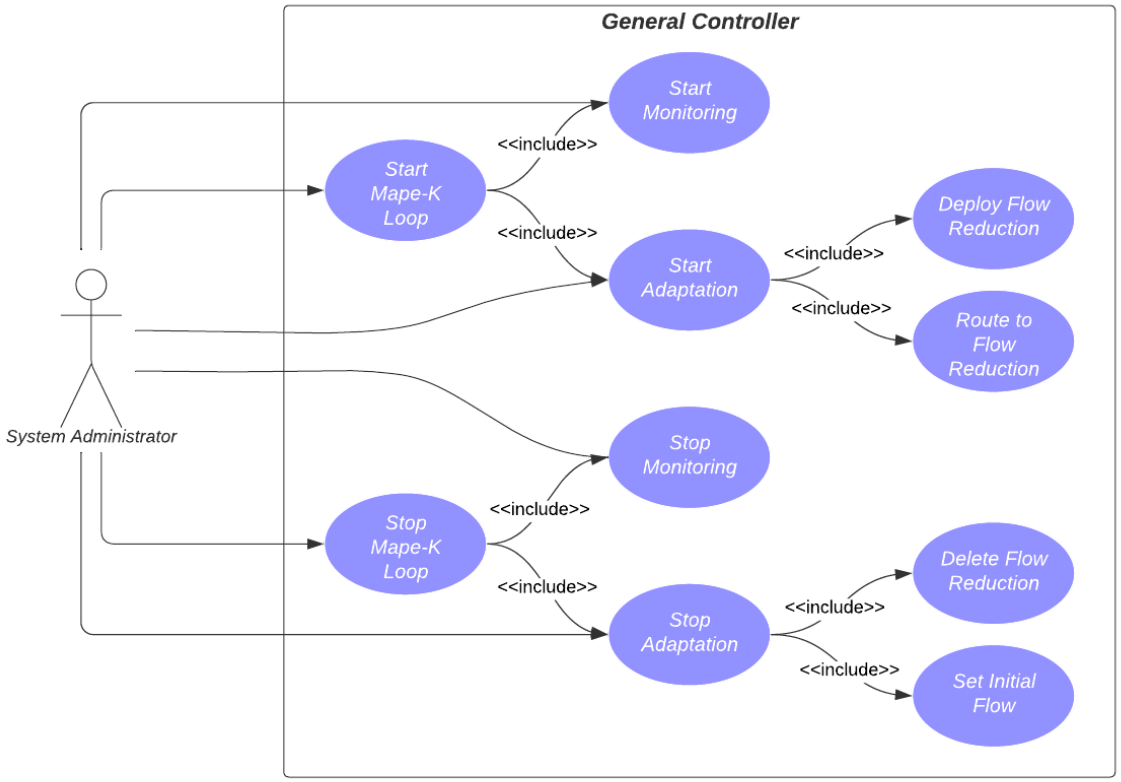
When analysing our project, there could have been several different ways to cater to the hypothetical problem of GI saturation and monitoring. For the monitoring, we could have utilized Istios Telemetry API to monitor more aspects of whether the system was saturated or not, such as request latency and memory consumption. These parameters could be extracted with the Istio Telemetry API in addition to the CPU usage per container, that we extracted using the Kubernetes metrics API. Another and perhaps easier way to handle saturation of the GI would have been to simply use Kubernetes native functionalities to horizontally scale the GI when needed, using Kubernetes autoscaling features.

However, Kubernetes in general has certain limitations when it comes to autonomous adaptation and complex decision making. As an example, Kubernetes does not offer rerouting or scaling based on custom logic. This is where the General Controller provides a more refined level of control. While we currently monitor standard system metrics, the General Controller allows for more precise and specialized adaptation if needed. Additionally, our approach optimizes resource usage, unlike Kubernetes' autoscaling, which can be more resource-intensive and costly.

All in all, our work demonstrates the potential of autonomic computing and dynamic service management, showcasing the power of Kubernetes and Istio in addressing complex architectural challenges. We hope this project serves as a valuable example of how autonomous systems can improve operational efficiency and ensure the integrity of critical data flows. For us personally, this project allowed us to get a better and more in depth understanding of Docker, Kubernetes and Istio.

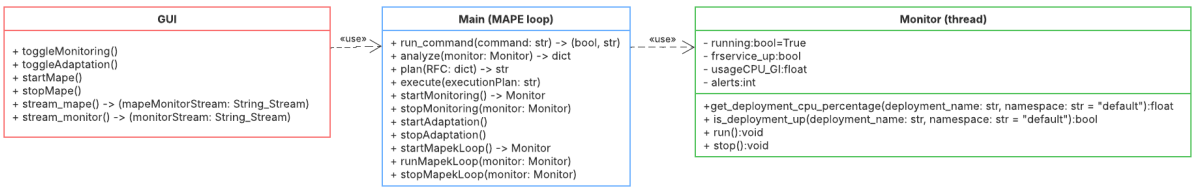
Appendices

Appendix A: Use Case Diagram of the General Controller



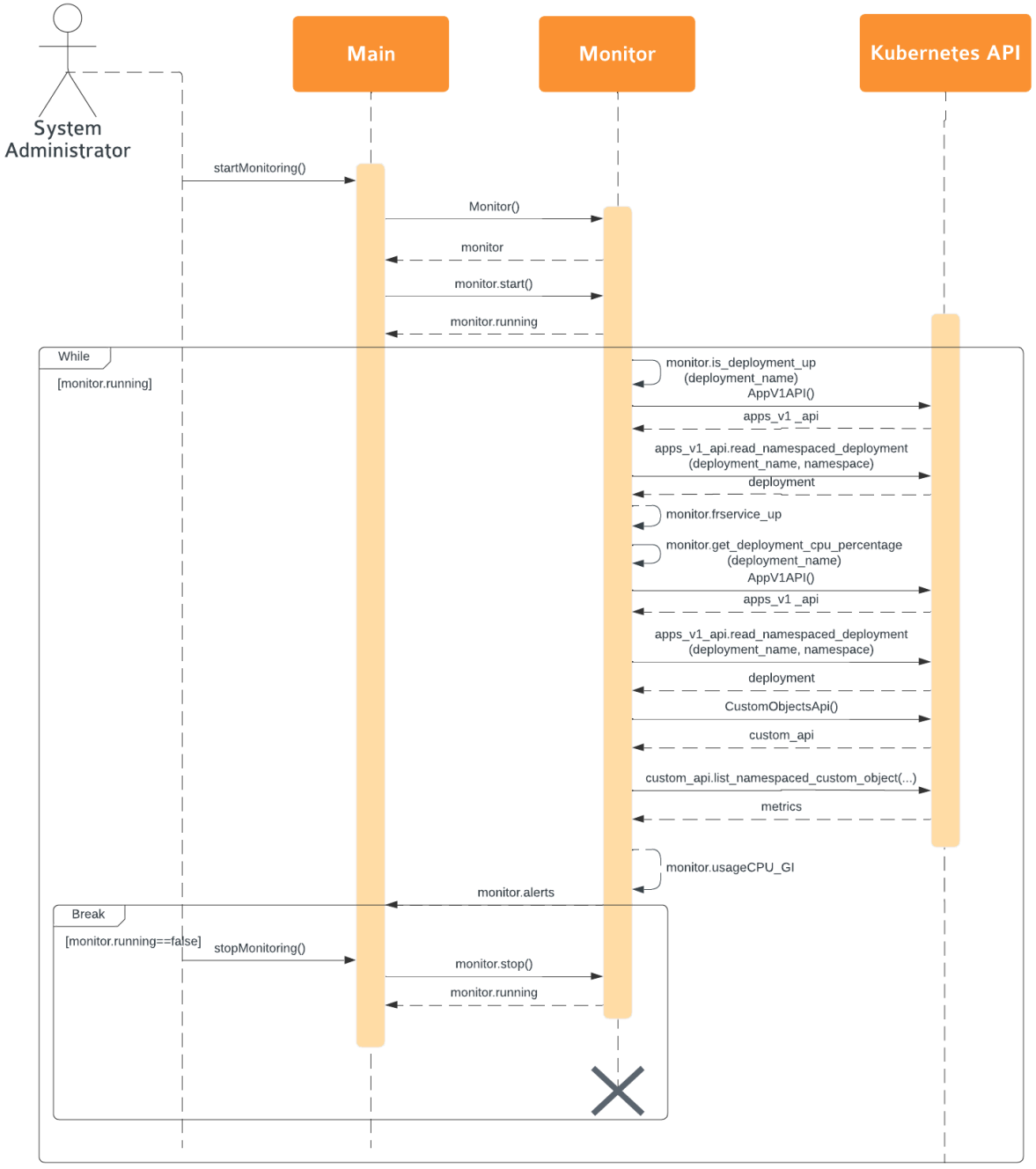
The use case diagram represents the general controller's functionalities. We can either toggle the monitoring and adaptation manually or the MAPE loop that will then toggle those processes automatically as needed.

Appendix B: Class Diagram of the General Controller



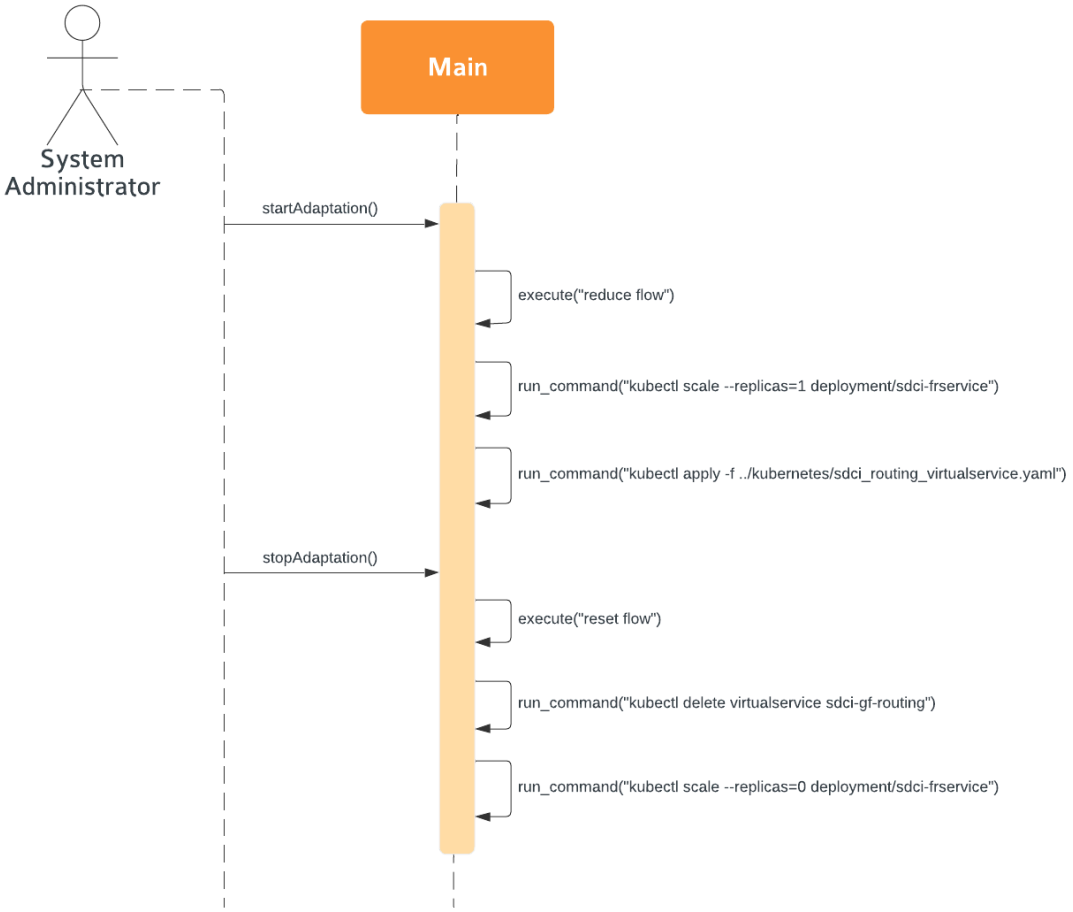
This diagram represents the classes of the general controller.

Appendix C: Sequence Diagram - Monitoring



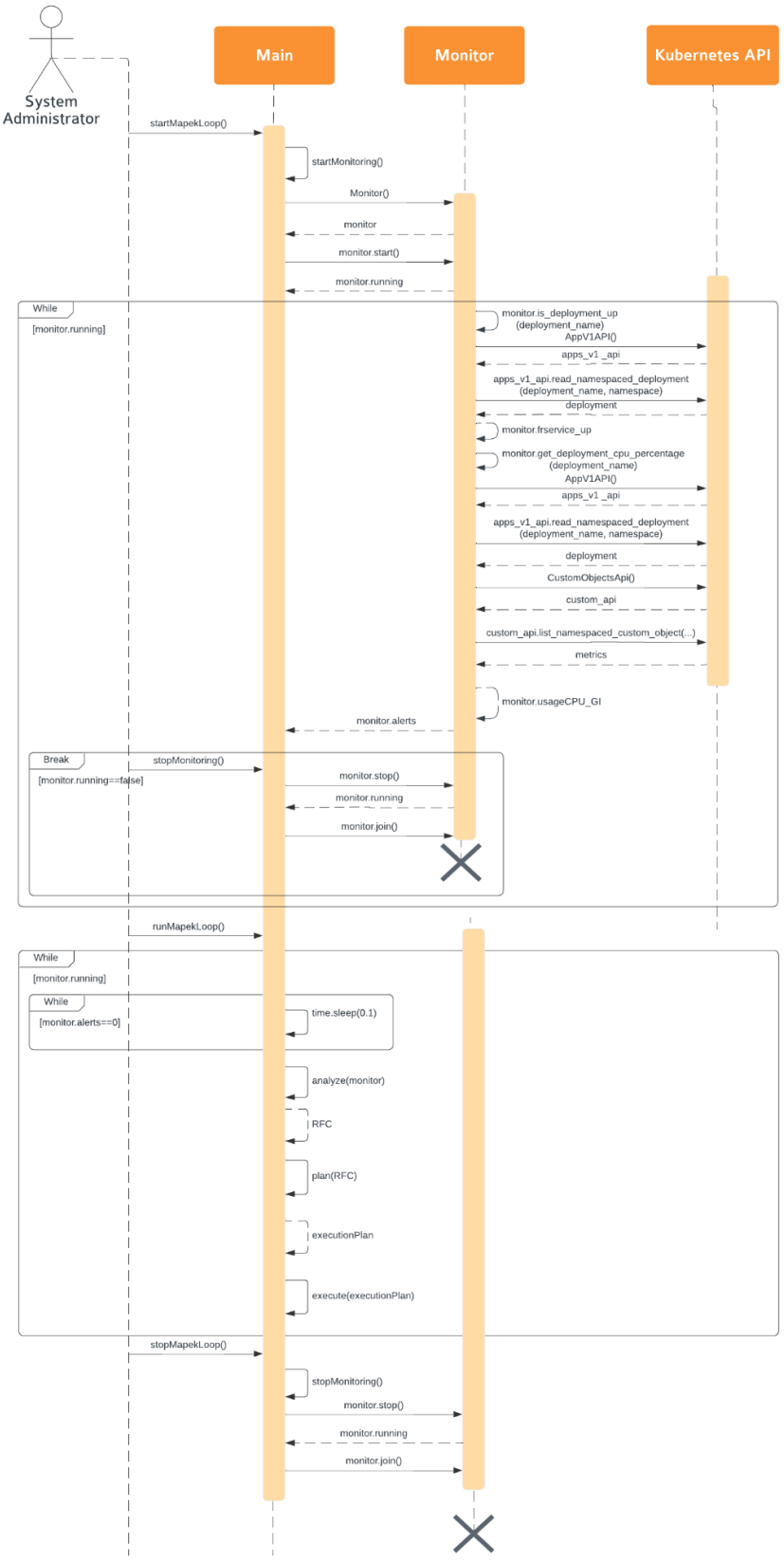
This sequence diagram represents the interactions between the different components when starting or stopping the monitoring.

Appendix D: Sequence Diagram - Adaptation



This sequence diagram represents the interactions between the different components when starting and stopping the adaptation.

Appendix E: Sequence Diagram - MAPE loop



This sequence diagram represents the interactions between the different components when starting and stopping the MAPE loop.