Web-scale Data Management - Final Project Report

Pragmatic Projects: Redis - Group 1

Bhoomika Agarwal, Martijn Janssen, Srinath Jayaraman Nagamani, Naqib Zarin
5018463, 4361709, 5049903, 4384474
Delft University of Technology
Delft, Netherlands
{b.agarwal, m.g.a.janssen, s.jayaramannagamani, n.zarin}@student.tudelft.nl

1. INTRODUCTION

In this project, we implement a micro-services architecture[1] using Redis as a back-end and compare it to the Postgres back-end implementation. Our goal is to see the benefit in terms of ease of use, performance, scalability and to implement ways to ensure consistency using SAGAs. We implement an online web-store where a user can perform typical webshop actions, like adding credit to the account, adding items to an order, followed by an order checkout. The aim for the entire system is to ensure that database consistency is preserved, after an order has failed, the system has to be returned to a correct state.

2. TECHNOLOGY CHOICES

2.1 To Go or not to Go?

Go is an open source programming language that makes it easy to build simple, reliable, and efficient software[2]. It is a programming language that is in continuous development to eliminate the challenges of the past and to keep up with the needs of the future. We decided to use it for our project because:

- **Speed**: As Go has a simple structure and syntax, it was easy for us to ramp-up and learn, given the time-constraints. Owing to its simplicity, it is easy to maintain, and development is faster. As it is a compiled language, it provides faster feedback, error-checking, easier deployment and code optimizations.
- **Highly concurrent and scalable**: Go was designed with scalability in mind. Since Go has it's own scheduler, which is not tied to the system scheduler, it allows us to run multiple processes simultaneously in an effective way. A result of this is that every incoming connection can be handled in it's own thread, a goroutine. To aim for effective inter-process communication, channels allow for fast communication between these goroutines.

2.2 Why Redis?

Redis is an open source (BSD licensed), in-memory key-value data store, used as a database, cache and message broker[3]. For our use-case, we chose Redis because of the following properties[4][5]:

- In-Memory
- Fast and performant if we store the data in RAM
- Low CPU usage
- Most actions are $O(1)$
- Message broker functionality ensures easy concurrent message passing
- Supports rich data structures

2.3 Shipping

Docker[6] is a containerizing tool that is "Empowering App Development for Developers". We use it for this project as it provides an easy way to package our application as a container that can run on different platforms, while also making deployment and configuration of our application easier.

2.4 Adding Kubernetes to the mix

Kubernetes is an open-source system to automate deployment, scaling, and management of containerized applications[7]. We use Kubernetes as it allows easy integration with Docker. In addition, it allows easy container orchestration, scaling and load-balancing in a uniform and convenient manner. In this project we have opted to use the GKE environment to allow for easy management.

2.5 The saga behind SAGAs

When given a choice to implement Two-phase commit(2PC) using SAGAs[8] or the Open XA standard[9], we decided to write a common implementation for both Postgres and Redis. As Redis does not have support for the Open XA standard for 2PC, we chose SAGAs. The SAGA flow is in our opinion easy to combine with Redis, while additionally keeping system complexity low. No need to add another central entity in the system, or learn one more technology.

3. ARCHITECTURE

3.1 Microservice architecture

As mentioned in section 2, we have one shared architecture for both Postgres and Redis. An overview of the top-level architecture can be found in Figure 1. In a nutshell, we have a client that makes calls that will be directed via an API gateway to the correct microservice. Each microservice has only one responsibility, and doesn't need to take into account all the other services. Our system architecture reflects the main advantages of a microservice architecture: we minimize the costs of change and increase the operational efficiency.

We enforce the 'single responsibility principle' by ensuring that:

- different responsibilities are divided over different services
- inter-service communication is only allowed through API calls or SAGA events
- each instance-kind only has access to the database of its kind

This modular approach prevents an increase in technical debt. For example, if we want to add new functionality (split the payment bill with other users), developers will likely not place code into other modules (e.g., stock service) to which they don't belong to by design.

There are two ways of coordination in sagas: Choreography and Orchestration[10]. Using choreography, each method can publish an event (or multiple) that triggers an action in another service, or multiple services. In an orchestration, there is one orchestrator that tells the participants what local transactions to execute. This approach introduces a single point of failure and can create a distributed monolith, something we want to prevent. Therefore, our implementation contains a mix of direct service to service communication, with Choreography SAGA events where required.

3.2 Workflow

Let's illustrate the workflow of our application by demonstrating how we handle the checkout functionality (see Figure 2). When the checkout is triggered in the order service, we fire a SAGA event. The event is sent to the payment service to initiate the payment. The payment service checks the payment status of this order to ensure that it has not been paid yet. It then makes an HTTP-call to the user service, telling it to subtract the cost of the order from the user credit. The user service checks whether there is sufficient balance and replies accordingly. If that action succeeds, the payment service sends a message to the stock service to perform stock subtraction for the items in the order. If the order items are successfully subtracted (no stock below 0), the service will send a success message to the order service.

This 'happy flow' shows that we perform inter-service calls without violating microservice principles in our architecture. We see that each service has its own state and other services can only trigger an action on data that is not theirs with an event or an HTTP-call.

3.3 Structure

The project structure can be found in Figure 3. Each microservice lives in its own folder, with all the code required for that microservice. All shared code is placed in folders like cmd, server and util. The cmd folder contains the commmands and flags for running locally. The server folder sets up the server and handles the routing, connecting to databases and the message broker. Finally, the util folder has helper methods for channel messaging, data objects and different types of HTTP response writing methods.

4. SCALING

Auto-scaling (or automatic scaling), is a computational technique widely used in cloud computing, where the amount of resources - usually measured in terms of the number of active servers - is varied based on the amount of traffic a server farm is experiencing. It is related to (and built on) the concept of load balancing[11].

In our project, we have implemented auto-scaling using the autoscaler from "Google Kubernetes Engine", or GKE. The configuration we used is provided in the "autoscale.yml" file. A small sample of this file is given below.

A YAML file for deploying a component in a Kubernetes cluster is essentially a configuration file that creates the required resource to statisfy the specification. For example, we are telling Kubernetes (GKE in our case), to mainain a set of **"x"** replicas of Pods.

Looking at a small sample of the auto-scale deployment file below, we see multiple commands specifying the autoscaling configuration used. These definitions consist of a key (for example "kind") and a value (for example "HorizontalPodAutoscaler").

```
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: payments
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: payments
  minReplicas: 1
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 50
```

- The "apiVersion" key specifies the API used with which the HPA (Horizontal Pod Autoscaler) objects are created. According to GKE documentation: "autoscaling/v1 is the default, and allows you to auto-scale based only on CPU utilization. To auto-scale based on other metrics, using apiVersion: autoscaling/v2beta2 is recommended." [12]
- The "kind" key creates a Horizontal Pod Auto-scaler that maintains between 1 and "x" replicas of the Pods we create. Value of "x" is specified later on in the same yml file.
- The "metadata" key is used here to specify the name. This is for the payment service, so we name this deployment of the autoscaler "payments".
- "spec" key specifies the configuration of this deployment file. There are further values mapped to this key:
  - apiVersion defined here is "apps/v1" - "apps is the most common API group in Kubernetes, with many core objects being drawn from it and v1. It includes functionality related to running applications on Kubernetes, like Deployments, RollingUpdates, and ReplicaSets." [13]

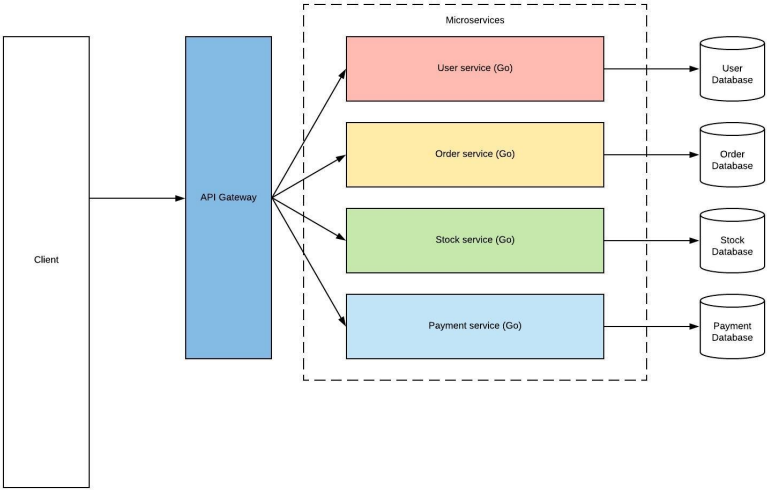


Figure 1: Microservices architecture.

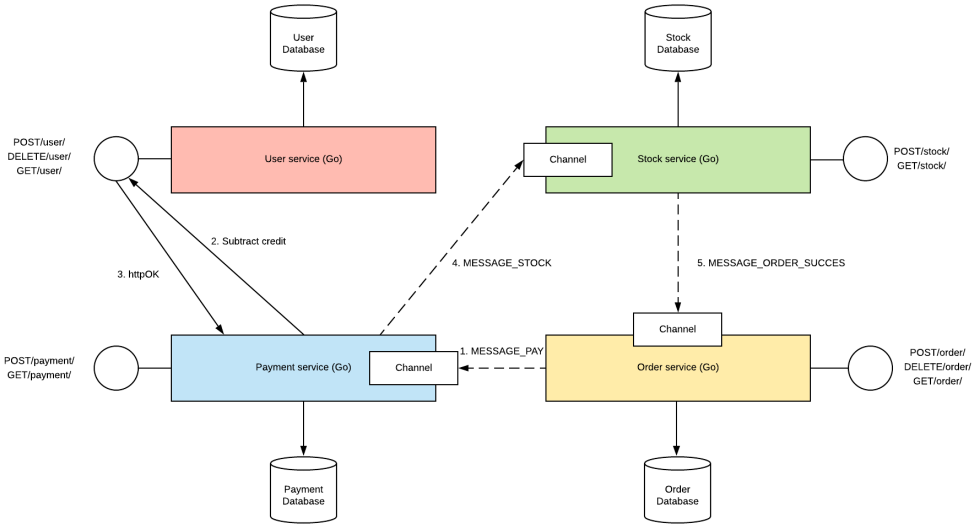


Figure 2: Workflow after checkout is triggered (no failures).

- "metrics" defines the custom metrics we are using to deploy this GKE cluster:
  - The "type" of resource we're defining here is "Resource", which for our use case is CPU resource of type "Utilization".

The reason why we are only scaling CPU resources and not memory is because local testing revealed that the memory utilization was not the limiting factor. It appears that our implementation is significantly more CPU heavy than RAM. The CPU load is also a good indication for when we need to have more resources, since the workload is CPU dominant. If at a later point the RAM does become a bottleneck, we can easily add scaling on that metric. From the evaluation we have done so far, the database is not a limiting factor, so the scalability question only comes into the picture when we're talking about the micro-services themselves. If an instance has, for example, 1 vCPU allocated to it, then it does not really use more than 80MB of RAM, as shown in Figure 4. Memory available for pods does currently not have an upper limit, and if there is significant CPU usage then we have a clear definition of when to scale.

5. CONSISTENCY

Our system architecture uses a *Database per service* pattern with each service having it's own isolated database. However, some business logic spans across more than one service, necessitating a mechanism to maintain consistency across services. We implemented SAGAs[8] - a sequence of local transactions - for the order checkout flow to ensure consistency and transaction support. We use a choreography-based SAGA implementation where "each local transaction publishes domain events that trigger local transactions in other services"[10]. The Redis Publisher Subscriber Messaging Paradigm[14] is used as a communication channel.

Figure 5 shows the SAGA implementation triggered by the order checkout flow. When the client sends a request to checkout, the order service publishes a message to the pay-

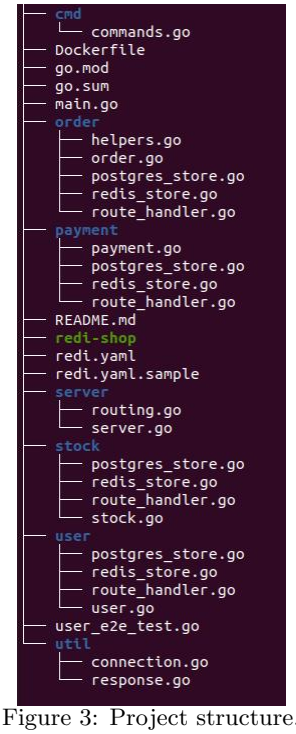
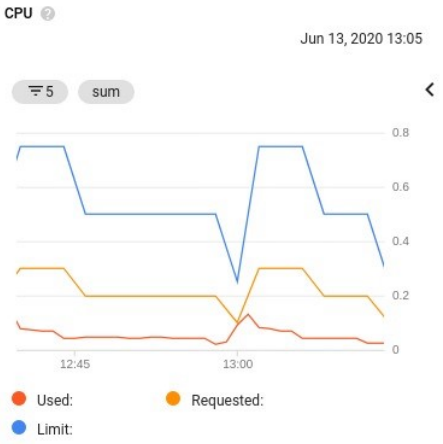


Figure 3: Project structure.

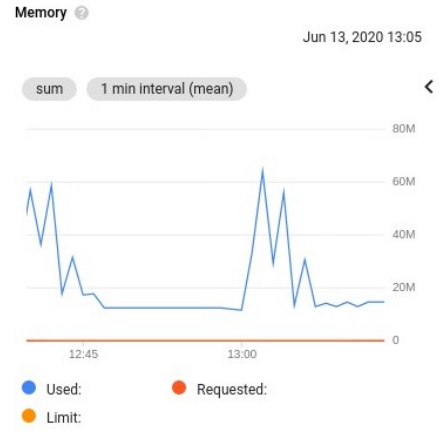
ment service via the payment channel to request payment. A tracker is added from the start of the entire flow to identify each checkout action throughout the system. This outgoing event triggers the SAGA flow for the payment service as shown in Figure 6.

When the payment is successful, the payment service sends a SAGA event to the stock service to subtract the stock as shown in Figure 7. When the stock subtraction succeeds for each individual item in the order, the order checkout flow is completed by sending a success SAGA event to the order service via the order channel. The order service then updates the response status as a success and completes the checkout flow. Since there is no way to identify which incoming event is tied to which request, the tracker is used on the event to associate the event with the request that came in.

There are two failure checkpoints in this flow - failure from the payment service and failure from the stock service. In the first case, the payment fails when the user does not have enough credit (bad request) or the user service is unavailable (internal server error). In case the payment fails, a message with the corresponding error is published to the order service via the order channel. The order service listens for these failure messages and returns the corresponding failure status to the user. If the stock subtraction fails due to the stock for an item being insufficient, a message is sent to the payment service via the payment channel to revert the payment. When the stock for a specific item in the order is insufficient, the stock subtraction for all the previous items in that order is reverted. At the same time, a message is sent to the order service with an error status. It is to be noted here that the failure flows are trivial in most cases as they are caused by insufficient stock or insufficient user credit. As there are no upper limits on either of these, they can be reversed by simply adding back subtracted stock or



(a) CPU utilization



(b) Memory utilization
Figure 4: Resource utilization

user credit. Adding to these values isn't going to give any conflicts or have any constraints.

6. BENCHMARKS

In order to evaluate the performance of our system, some benchmarking is needed. This is important in order to define the quality of service we are able to provide. We use load testing to get insights in breaking points, throughput and latency.

6.1 Setup

For testing we used a local server which started a container for each micro-service. In total 4 containers are created for each microservice, 4 database instances, Redis or Postgres, and 1 Redis instance to be used as a message broker. With both backends, the amount of active users doing requests is gradually scaled up to compare performance. For load balancing between the microservices, Traefik was used. The server itself has a AMD GX-420GI processor, with 4 cores at 2.0Ghz.

6.2 Insights

Our research gives the following insights.

6.2.1 Scaling number of users

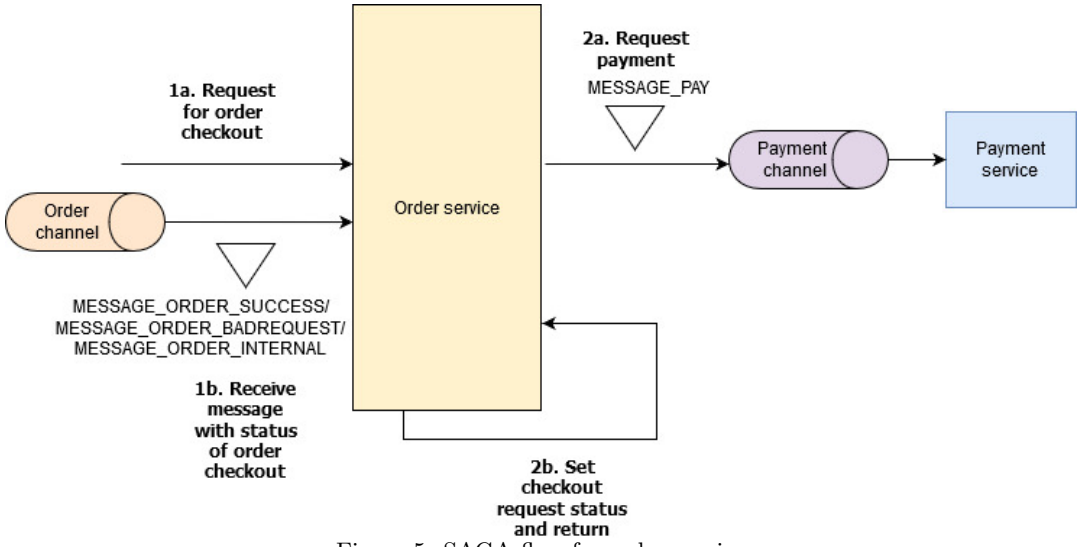


Figure 5: SAGA flow for order service

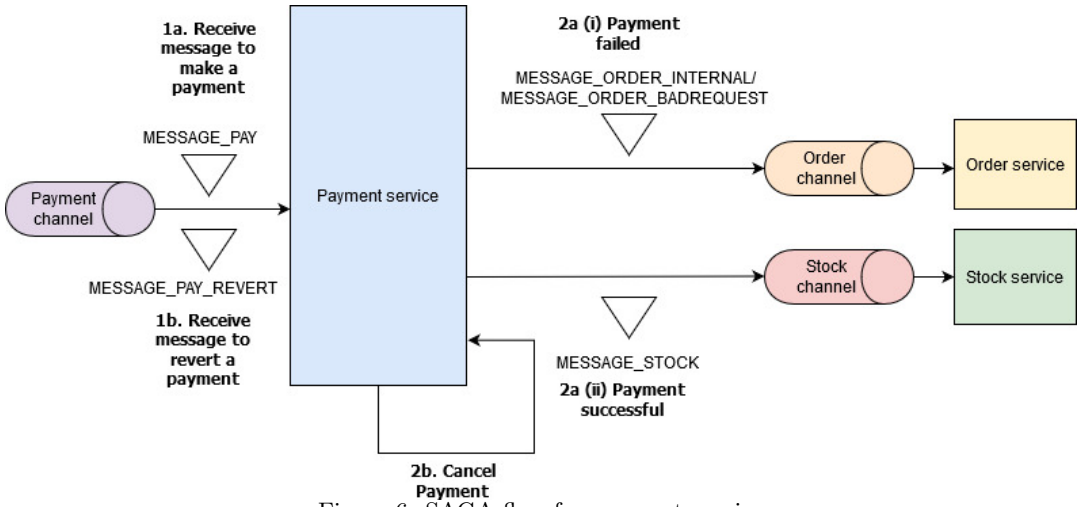


Figure 6: SAGA flow for payment service

One of our benchmark experiments was to test the limits of the database servers. We scaled until a breaking point was reached, indicated by an increase in latency. For Postgres, when scaling the amount of users past 2000, the system slowed down - an increase in latency and 100% CPU load was noticed. Postgres wasn't able to handle that amount of requests on the database and was stuck at 215 requests per second. On the other hand, for Redis, the requests were handled just fine without any problems even at 8000 users. The issue encountered here was that the resources of the testing system were used to a maximum with 90% load on all the cores and request processing on our side was the bottleneck. This implies the limit for Redis isn't even 8000 users, but much higher. The limit of 900 requests per second for the testing system was reached, but the server was able to handle more at 90% CPU usage.

6.2.2 *Consistency*

Our results show that in total 0.63% of calls in Redis result in failures whereas with Postgres 0.7% of the requests fail. Also, the failure rate (failures/s) for our Redis and Postgres implementation is 3.75 and 1.36 respectively. This indicates that the results from the systems closely resemble each other. Looking at wrong responses in Redis, 6% of the time an incorrect response was given from the checkout route, all resulting from stock subtraction. For Postgres it was the same, with 7% of the calls failing with an issue in stock subtraction. From these numbers, we assume that there is probably still a mistake in the implementation for the stock service subtraction, but we haven't found that issue yet. With this implementation, Redis seems to be as consistent as Redis.

6.2.3 *Latency*

Figure 8 shows the comparison between the response times for the Redis and Postgres implementations in milliseconds(ms), compared with the total number of requests per second. The average response time for Redis is 322ms, compared to the average of 250ms for Postgres. These times appear to be quite close, but upon closer inspection of the graph, it seems that the response times in the later requests have had quite a

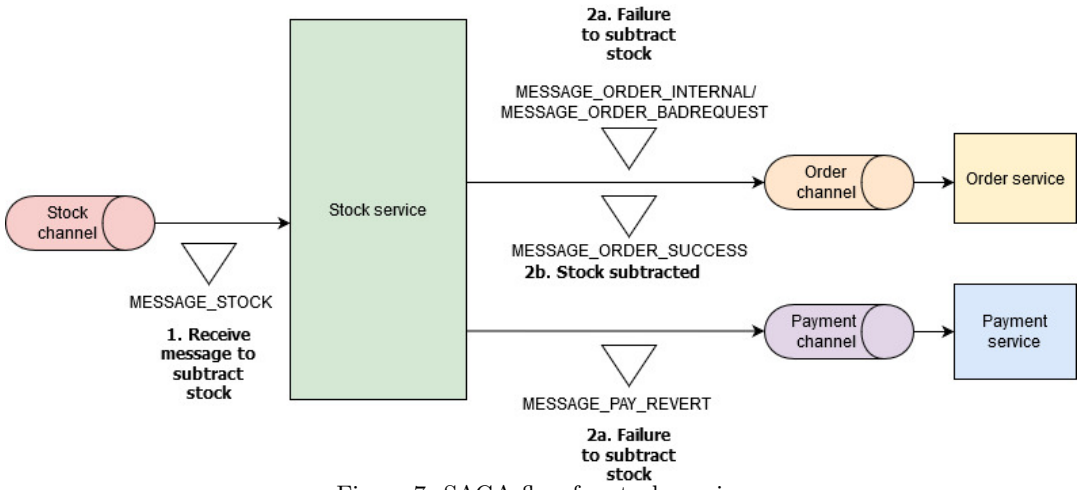


Figure 7: SAGA flow for stock service

negative influence for Redis, since there the median response time rose drastically. It it likely that this was influenced by the loadbalancer deployed, since Traefik was at 100% load at that point. In other experiments, the Redis implementation has shown to be quicker than Postgres.

7. CONCLUSION

Although there is no difference in logic or implementation between Postgres and Redis, the results show that Postgres cannot handle the load like Redis does. Redis also uses less resources compared to Postgres, which is a very nice property to have.

References

[1] Microservices. https://martinfowler.com/articles/microservices.html, accessed June, 2020.

[2] Go programming language. https://golang.org/, accessed June, 2020.

[3] Redis. https://redis.io/, accessed June, 2020.

[4] Anwar ul Haque, Tariq Mahmood, and Nassar Ikram. Performance comparison of state of art nosql technologies using apache spark. In Kohei Arai, Supriya Kapoor, and Rahul Bhatia, editors, *Intelligent Systems and Applications*, pages 563–576, Cham, 2019. Springer International Publishing.

[5] Comparing in memory databases: Redis vs. mongodb. https://scalegrid.io/blog/comparing-in-memory-databases-redis-vs-mongodb-percona-memory-engine/, accessed June, 2020.

[6] Docker: Empowering app development for developers. https://www.docker.com/, accessed June, 2020.

[7] Kubernetes: Production-grade container orchestration. https://kubernetes.io/, accessed June, 2020.

[8] Hector Garcia-Molina and Kenneth Salem. Sagas. *ACM SIGMOD Record*, 16(3):249–259, Jan 1987.

[9] *Distributed transaction processing: the xa specification.* X/Open, 1991.

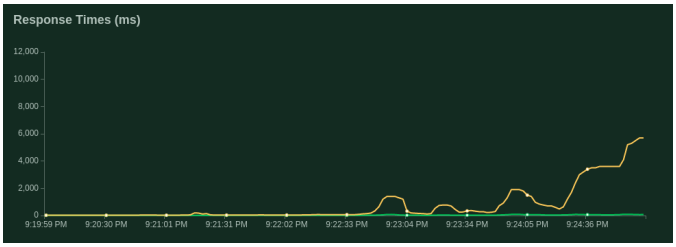
[10] Pattern: Saga. https://microservices.io/patterns/data/saga.html, accessed June, 2020.

[11] Autoscaling - from wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Autoscaling, accessed June, 2020.

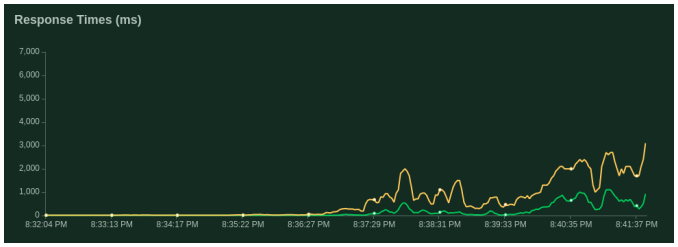
[12] Configuring a horizontal pod autoscaler. https://cloud.google.com/kubernetes-engine/docs/how-to/horizontal-pod-autoscaling#api-versions, accessed June, 2020.

[13] Which kubernetes apiversion should i use? https://matthewpalmer.net/kubernetes-app-developer/articles/kubernetes-apiversion-definition-guide.html, accessed June, 2020.

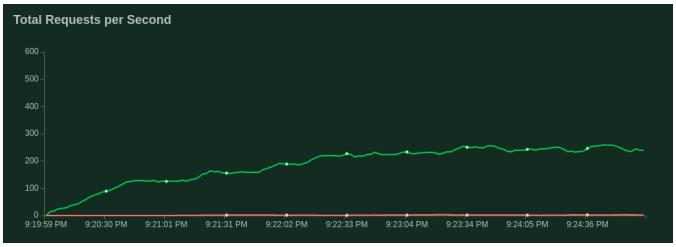
[14] Pub/sub. https://redis.io/topics/pubsub, accessed June, 2020.



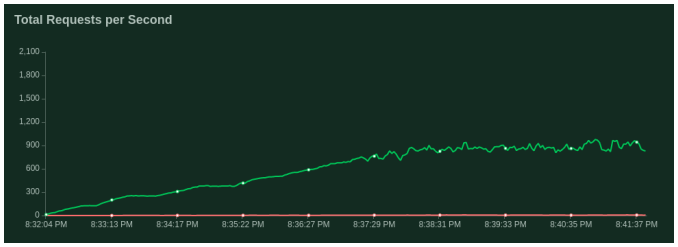
(a) Response time: Postgres



(b) Response time: Redis



(c) Requests per second: Postgres



(d) Requests per second: Redis

Figure 8: Comparison of Response times and Requests per second for both implementations