Microservices - Group 3

Pedro Carrega, nº49480 Vasco Ferreira, nº49470
Ye Yang, nº 49521

Contents

1 From Monolith to Microservices: A Classification of Refactoring Approaches

1.1 Introduction

This paper focuses on comparing refactoring approaches of pre-existent monolithic architecture services to a microservices based architecture in order to make the most out of the scalability and efficiency this type of architecture offers.

As seen throughout the course, microservices bring many advantages over traditional monolithic architectures through non functional characteristics such as elasticity, scalability, redundancy, among others. Monolithic applications become larger and more complex overtime, having the entirety of its source code distributed on a single system. This makes it hard for developers to debug their code and thus making it hard to maintain.

However, the re implementation of the system with a microservices architectures is no easy feat, as one has to take into account how to partition the source code not only logically, but also regarding exterior factors in the real world as we will further address. The correct deployment of a system with such an architecture will depend on the correct refractoring and decomposition of the system.

1.2 Refractoring

When refractoring an application, we must not only consider code level refractoring, but also architectural refractoring. This is mainly due to most organizations building a system according to its own communication structure, thus both the system's architecture and the organization are interdependent making the refractoring process not only code based.

We must however, also take into account the granularity of the services in which the monolithic application will be decomposed into, so as not to overload some services and underload others due to improper task distribution. Two ways to split the system into smaller parts are underlined in the paper:

- **Implementation Technology** - separation of the monolithic system into smaller subsystems based on the level of computation (ex.: Heavy services written in C into one subsystem and IO-heavy services written in NodeJS in another subsystem).
- **Geography** - separating the system based on the physical location of each development team, whether it be due to commercial, cultural, legal or technical reasons.

To gather the data regarding the refractoring techniques in microservices, the authors queried papers with specific keywords on three of the largest libraries of scientific papers in computer science: IEEE, ACM Digital Library and Google Scholar.

1.3 Results

With the data obtained from the three libraries, only 10 were selected, as most others focused on the requirements of a specific scenario, without much theoretical basis, not being able to generalize their approaches for other systems.

The various approaches from the 10 selected can be categorized in 4 main decomposition strategies which take into account external factors, not only based on source code, as input in order to determine the granularity of each service offered. The strategies are the following:

- **Static Code Analysis aided** - primarily based on studying the source code of the application and logically decomposing it into smaller subsystems.

- **Meta-Data aided** - based on the architecture of the system, aided by UML diagrams and use cases. More abstract than Static Code Analysis.

- **Workload-Data aided** - based on the computational power required for the application in order to determine its decomposition (ex.: communication, performance).

- **Dynamic Microservice Composition** - in which we define a microservice runtime environment best suited for each set of services. This environment is subject to change with each composition re-calculation iteration in order to find the best fit for each service.

In Figure 1 we can see a decision graph made by the authors so as to simplify the choosing of the best option for specific type of application to be refractored. The decomposition approaches are best viewed directly from the paper which can be accessed through the following link: https://arxiv.org/ftp/arxiv/papers/1807/1807.10059.pdf

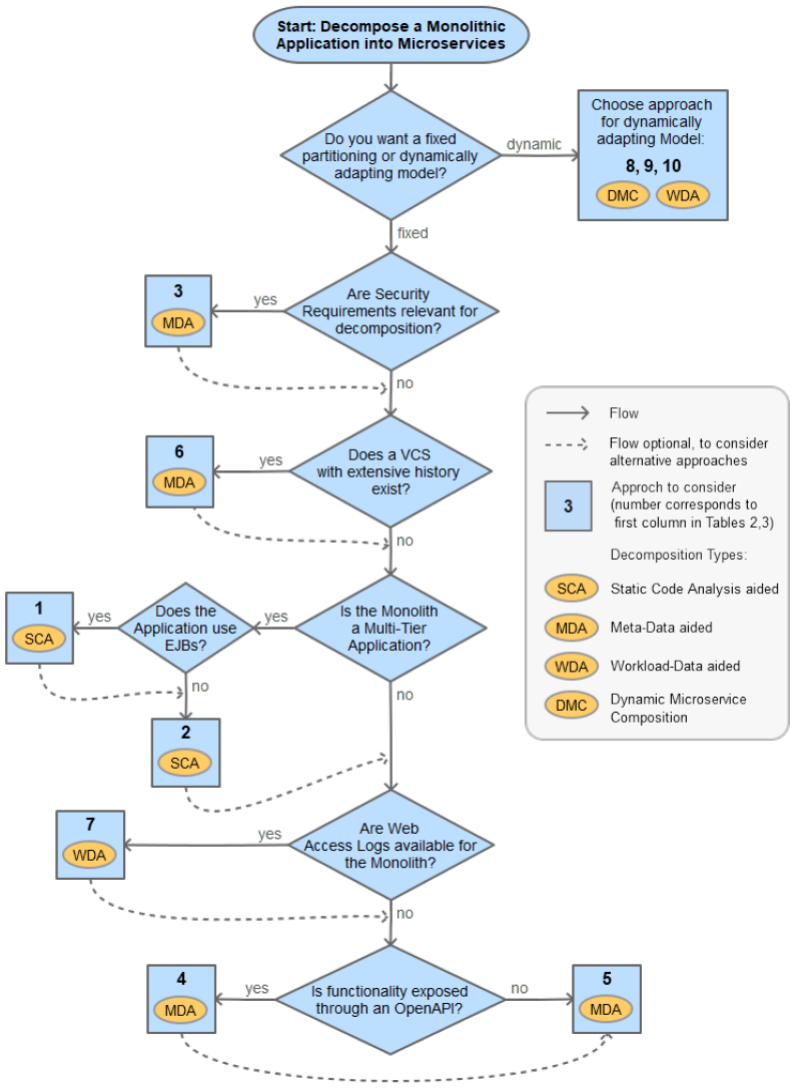

Figure 1: Decision Guide

1.4 Conclusion

In the presented paper, the amount of proposed strategies for application refractoring and decomposition are limited and although more generic than the papers that were filtered out, limitations to universal adaptability are still present. Nevertheless it is a solid baseline which can be used to refractor basic level applications, small to medium sized. Enterprise sized applications would have to apply a more sophisticated method that best caters to their specific scenario.

2 Research on optimization of course selection system based on micro service and dynamic resource extension

2.1 Introduction

In the academic world the course selection systems have been using remote technologies for some time, that way it is possible for students to enroll in the courses when they want and where they want. This is a better approach to the institution since course selection would require a huge amount of staff and would take more time for all students to enroll. Has expected, all this remote course selection has its problems, in most of the educational institutions where it is implemented, there is the problem of servers crashing during the process due to the huge amount of students concurrently accessing the services. The institution could use vertical scalability, by upgrading their servers so that it could support a bigger amount of concurrent requests. But this approach as it will be shown could be unfeasible since it will have an associated monetary cost.

To tackle this reoccurring problem three students from the Guangzhou

University though to optimize the architecture of the system instead of maintaining the same but improving the servers. Their idea was to deploy a micro-services architecture, separating all the business logic by micro-services, where there are the following benefits:

- Easier to develop and maintain
- Local modifications are easier to deploy
- Easier to scale the necessary services

To follow this approach, it was needed to re-design the system architecture.

2.2 System Architecture

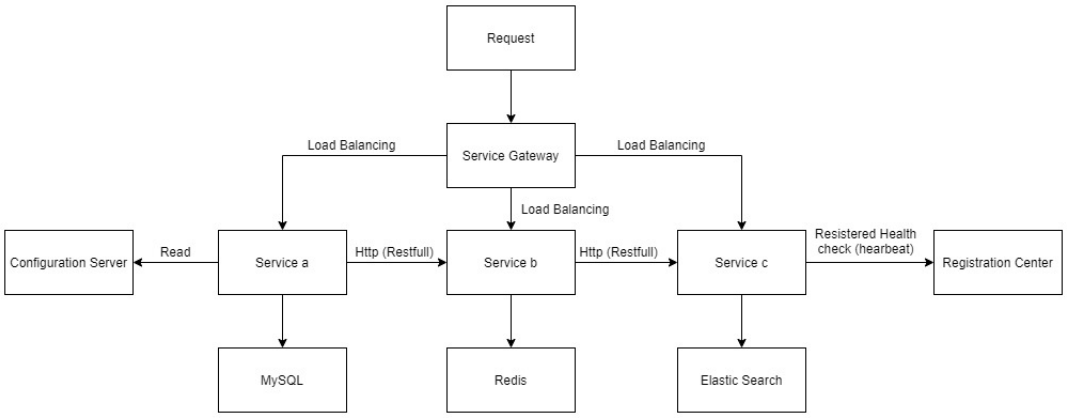

Figure 2: System Architecture

Registration Center: This component is used to standardize the information of the various micro-services, provide query API for querying available instances and the manage API for service registration, cancellation, discovery and service inspection (heartbeat mechanism).

Service Gateway: It is used provide a better access point to the client, abstracting all the underlying structure of the application.
Configuration Server: Since some services are dependent on another the configuration server is needed for a centralized management configuration.
Redis Cluster: Some necessary information is pre-stored in Redis. After the class selection the data in memory is store in the database by timing tasks, to achieve Redis pressure resistant property.
Distributed Tracking: By combining the elastic search with course log analysis system and micro-services can solve problems in time to avoid larger problems.

2.3 Micro Services

A service implementation mainly includes business logic, data storage, adapter, and service interface. The database adapter is useful to adapt different services to different databases, so that it facilitates the migration of services in different database environments. Database servers provide data storage support for micro-services to ensure the atomicity and independence of services.

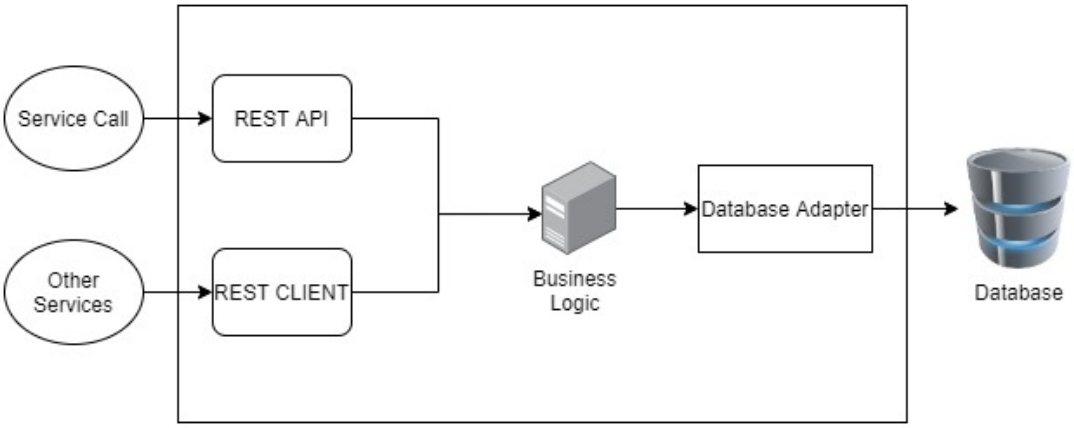

Figure 3: Micro Service Implementation

2.4 Results and Conclusion

After all the implementation the students accomplished a huge improvement in each service, due to the benefits of micro-services that were mentioned before. With the monolith, the institution servers would struggle with only 3000 to 4000 concurrent users. After the implementation the students were faced with the following results:

Number of concurrent users	**Transaction success rate**	**Average transaction response time**
4000	100%	1.314 seconds
10000	100%	2.52 seconds
15000	100%	7.011 seconds

Table 1: Results with micro-services

With the following results we can determinate that the application was a success, that the problem was not the hardware it self, but the

architecture that defined the system. By changing it to micro-services all the transactions were successful which is the main goal, but also the average transaction times were very low which provides a better user experience.

The goal is not to say that micro-services is always the way to go, but when applied in the correct scenario, applications can benefit enormously due to its properties, like on demand scalability of services, adjusting the number of nodes running certain services that are experiencing a high load while some services are not having any load at all.

To understand more of the evaluation done in the course selection system, visit the following link to read the paper: https://dl.acm.org/doi/abs/10.1145/3335484.3335546