ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/353013708

Microservice based architecture for the genetic algorithm

Conference Paper · April 2018

CITATIONS
0

READS
64

1 author:

Goran Petkovski
Ss. Cyril and Methodius University in Skopje
6 PUBLICATIONS **2** CITATIONS

SEE PROFILE

All content following this page was uploaded by Goran Petkovski on 06 July 2021.

The user has requested enhancement of the downloaded file.

Microservice based architecture for the genetic algorithm

Evgenija Stevanoska, Kristijan Spirovski, Goran Petkovski,Boro Jakimovski, Goran Velinov
Faculty of Computer Science and Engineering
University Sts Cyril and Methodius
Skopje, Macedonia
{evgenija.stevanoska, kristijan.spirovski, goran.petkovski}@students.finki.ukim.mk, {boro.jakimovski, goran.velinov}@finki.ukim.mk

***Abstract*—Microservice architecture is becoming more popular and more frequently used, mainly because of its numerous advantages over monolith approach. Namely, the developed systems nowadays need more agile distribution of the processing power, and due to their size, a way to deploy, maintain and test individual components separately. Each algorithm/software composed of individual and independently executable parts that do not share many parameters are good candidates for a solution based on a microservice architecture. This paper presents a microservice approach when building an architecture for the genetic algorithm. We identified eight independent parts of the genetic algorithm, and each one is represented as a microservice. This design leads to a solution that has low coupling and high cohesion. The advantages of this approach include distributing the computations on more physical locations, and furthermore, scaling only the parts of the system which require more performance (which means need large processing power or are frequently executed).**

***Keywords*—microservice, genetic algorithm, Spring Boot, Spring Cloud**

I. Introduction

The large number of NP-hard problems that can't be solved in polynomial time imposes the need to use an alternate approach for finding near optimal solutions to the optimization problems in efficient time. Popular choices for that task are meta-heuristic optimization algorithms. One of the oldest and still very popular meta-heuristic algorithm is the genetic algorithm (GA), which models the evolution of the population by applying mutation and crossover to the individuals in the population. The nature of the genetic algorithm offers easy identification of the independent parts and their parallel and distributed execution. These characteristics make the genetic algorithm suitable for a microservice based implementation.

A microservice is an independent service which is responsible for one functionality and it collaborates with other microservices using a strongly defined interface, often using messages with a predefined structure. Microservice architecture is an approach for developing applications using small, independent services. Each microservice is executed as a small, independent process. Nowadays, the microservice architecture becomes more popular, and the advantages over the monolith approach are obvious.

Microservice based implementation of the genetic algorithm allows individual components to be executed on more physical locations. Furthermore, not all components have the same processing needs. This approach allows us to identify the most computationally extensive components and assign them more processing power, which leads to easy scalability, where each component has only the resources needed for optimal execution. The components of the algorithm can be independently executed. The nature of the microservice architecture allows parallel execution of the independent parts.

This paper presents a microservice based implementation of the genetic algorithm. The goal is to show that GA is suitable for a microservice approach and to give a prototype of the implementation (which is defined by the choice of independent components). We implemented eight microservices that communicate through messages (REST API). The microservices are implemented in Spring Framework using Spring Boot and Spring Cloud libraries, and they are deployed to Pivotal.

The paper is organized as follows: Section II gives an explanation and a definition to microservoces and microservice architecture, and it also lists the advantages over the monolith approach. Section III contains a description and pseudo-code of the genetic algorithm. The proposed implementation is given in Section IV, which contains information about the micriservices, and also a list of the used technologies. Finally, the last section gives the conclusion.

II. Microservices

The term microservices was first introduced in 2011 at an architecture workshop, as a way of describing the common idea of the members of the workshop for a new software architecture. Since then, they are implemented in many popular and commonly used software solutions. For example, Netflix uses microservice architecture known as Grained SOA [1].

Microservices are small processes that can be independently deployed, scaled and tested. Each microservice has only one functionality and responsibility, which means it can be easily updated and understood by a programmer [2]. Functionalities that are not strongly connected are modeled using different microservices, thus, the microservice architecture supports the high cohesion-loose coupling paradigm.

Microservice architecture is an application whose components (modules) are implemented using microservices. The

microservices communicate using predefined interface (often using messaging concept). The microservice approach has numerous advantages: [3]

- Each microservice implements a limited number of functionalities, leading to a small codebase, and thus low probability of a bug in the code. Furthermore, if a bug exists, its scope is limited to the microservice, which leads to an easy identification of the part of the code causing the bug. The fact that microservices are independent means that they can be easily tested, and their scope easily understandable even if they are isolated from the system, leading to a higher code reusability. Usually, each microservice is developed and deployed by one team, and the team is responsible for maintaining it.
- Each microservice can be easily replaced while other services work normally. If an application based on a microservice architecture needs to be updated, the process can be performed by gradually replacing each microservice (or furthermore, uploading the new version alongside with the old one), and then updating all the microservices that are dependent of the replaced microservice.
- Consequence of the previous point is the fact that the replacement of one microservice doesn't cause down-time to the whole system. A reboot is needed only on the microservices of the replaced module.
- Microservice architecture scaling doesn't mean doubling all components' instances. Namely, it offers the opportunity to the developers to monitor the load to each microservice, and to scale only the microservices that need higher processing power.
- The only limitation that microservices have is the technology used for communication. Other than that, the developers are free to choose the optimal resources for a microservice, which includes the framework, the implementation language etc.

The microservice approach has few disadvantages, caused mainly by the message-based communication between the microservices. They include:

- Sending messages through the network is slower than in-memory calls. To achieve comparable performances with monolith architecture, the number of calls through the network should be limited.
- Special attention should be paid on security of the communication. The messages are usually send in json or xml format, which can be easily intercepted. To maximize the security, the communication should be encrypted.

These facts are highlighting the advantages of the microservice architecture over a monolith one, making the former one more popular and commonly used. Namely, monolith systems are often enormous, which makes them hard to maintain. Finding a bug in such system takes longer because of the inability to easily identify the part of the code causing the bug. Attention should be paid on dependencies between different libraries, because adding or updating a library to a newer version could cause inconsistency in the system. A change in one part of the system causes reboot on the whole application, which means greater downtime. Furthermore, the scalability of a monolith application is limited to making more instances of the whole application, and balancing the load between them. Obviously, this approach is not efficient when the traffic is increased for one part of the application. Also, monolith architecture uses one language for the whole application, which eliminates the opportunity to choose the most convenient language and framework for each functionality.

Compared to the standard service oriented architectures, Microservices show a large number of common parts. Namely, they are both relying on as the main component, but they vary in terms of service characteristics (for example, service size and functionalities each service models).

The advantages of microservice architecture over the SOA include:

- Microservices can operate and be deployed independently (unlike SOA), which makes it easier to deploy a new version of the application and scale only the necessary parts.
- they are better at fault tolerance. If a microservice fails, it only affects the part where the microservice is used, all other micriservices function independently. In SOA, Enterprise Service Bus (ESB) becomes a single point of failure.
- In SOA the data is shared and accessible from all services, making the services tightly coupled.
- Containers can be easily used with microservices (and not with SOA).

These facts just confirm that microservice architecture is gaining its rightful popularity, and is a good and commonly used architectural choice in newly developed systems.

III. Genetic Algorithm

GA is a common heuristic optimization method based on the principle of natural evolution. In nature, the individuals evolve following the principles of natural selection and survival of the fittest. Theoretical basis of the genetic algorithm are introduced by Holland [4]. GA follows this principles of evolution in order to find a near optimal solution to an optimization problem.

A. *Description of the Algorithm*

Genetic algorithms encodes a potential solution to a specific problem on a simple chromosome-like data structure and applies recombination operators to these structures so as to preserve critical information. An implementation of a GA begins with an initial population of (typically random) chromosomes. Each chromosome is evaluated using a fitness function that is specific to the problem being solved. Then, based on the fitness values, the GA allocates reproductive opportunities in a way that the ones that represent a better solution are given more chances to reproduce than the chromosomes that represent poorer solutions. On the chosen subset of chromosomes (to be part of the next generation) following operators for simulating the evolution process through generations are used in order to create the next generation:

- **Selection**. Every GA uses a selection mechanism to decide which individuals will comprise the mating pool which will form the basis of the next generation and will be used to generate new offspring. When dealing with a problem of minimization, the individuals with lower fitness values will have a better chance to be selected for the mating pool [5].
- **Crossover**. Crossover is the process of randomly selecting two parent chromosomes from the mating pool, exchanging genetic material between them and producing new offspring chromosomes. This process tends to increase the quality of the populations and force convergence.
- **Mutation**. This operator represents a small change on the genetic material of a chromosome. It is performed by changing one or more components of a chromosome. Mutation is used to improve the diversity of the chromosomes in the population. It lowers the probability of the algorithm being trapped in local optima, hence it plays an important role in any GA.

B. Parameters of the Genetic Algorithm

As previously mentioned, the genetic algorithm takes few input parameters which have a direct impact of the GA's behavior and the quality of the found solutions.

The population size and the number of generations. These two parameters influence the trade-off between the quality of the found solution and the execution time. Increasing the number of chromosomes in the population is proportionally increasing the size of the search space that the algorithm covers, and thus it increases the probability of finding a better solution.

Crossover probability. This parameter controls the portion of the population which is directly inherited from the last generation, (and thus the portion of the newly formed individuals). Usually, 20% of the individuals go to the next generation unchanged, and the remaining 80% of the population are filled by applying the crossover operators on the best individuals from the previous generation.

Mutation probability. As previously mentioned, mutation is a mechanism for avoiding local optima traps. With certain probability, each newly created individual is subject to mutation. Usually, this parameter is in the scale of 10%.

Dimension of the search space. This parameter depends only on the concrete application of the algorithm (the optimization problem that it tries to solve) and the defined encoding of the solutions as vectors.

C. Characteristics of the Genetic Algorithm

The genetic algorithm is easy to implement and to understand. The nature of the algorithm allows simple parallelism and distributed execution. The best found solution is always part of the last generation, and no additional memory for storing is needed. The number of values exchanged between the microservices are relatively small. Mutation and crossover allow the algorithm to escape a local optima and to find a good solution. The algorithm includes a probability model, so it is recommended to execute the algorithm more times from different initial populations. One of the main disadvantages of the algorithm is the execution time.

These characteristics are the reason why genetic algorithm is one of the most commonly used meta-heuristic optimization algorithms today, 25 years after its introduction. Especially the easy distribution and parallelization make the algorithm suitable for a microservice implementation, which is the main motivation for this paper.

IV. Implementation

In the proposed implementation, GA is divided into seven independent components, each implemented as a different microservice. Also, another microservice is added, which is visible to the user, and its goal is to catch the calls to the genetic algorithm and sends a request to the corresponding services. The complete architecture is given on Fig. 1. Each microservice is represented with a rectangle. The arrows are showing service calls (the service which is on the side of the arrow is called by the service on the other end). The green parts contain information about the input parameters and the output value of each microservice. The next paragraphs contain a detailed description of each microservice.

GA Service. This is the main service visible to the user. As an input it takes a parameter object, and is responsible for calling and coordinating of the other services. At the end of the optimization process it returns the best solution to the user.

Initialization. This microservice takes a parameter object as an input (which contains the parameters described above), and based on the values of the size of the population and the dimension of the chromosomes randomly initializes the first generation. The result of this microservice is an initial population, which is a set of chromosomes randomly placed in the search space. Furthermore, this service stores the parameter object which is used by all other microservices. This approach achieves greater speed by sending fewer parameters between service calls. Namely, this service implements REST API which is used to return the parameter object or to change some value of the parameters.

Evolve. The service is responsible for evolution of the generations. As input it takes the initial population, evolves it through more iterations, and returns a list which contains the best solution of each generation.

Select best. This microservice implements the selection mechanism of the best chromosomes in the population. As input it takes the current generation and outputs a subset of chromosomes which will be part of the next generation. This set forms the basis for the next generation, namely, the rest of the next population will be formed by applying crossover and mutation on the individuals contained in the chosen subset. In the literature many selection operators are introduced, which are suitable for different applications, by making a trade-off between the execution time and the probability of returning exactly the best k solutions. For example, tournament selection [6] (with complexity $O(k)$) makes k iterations, and in each

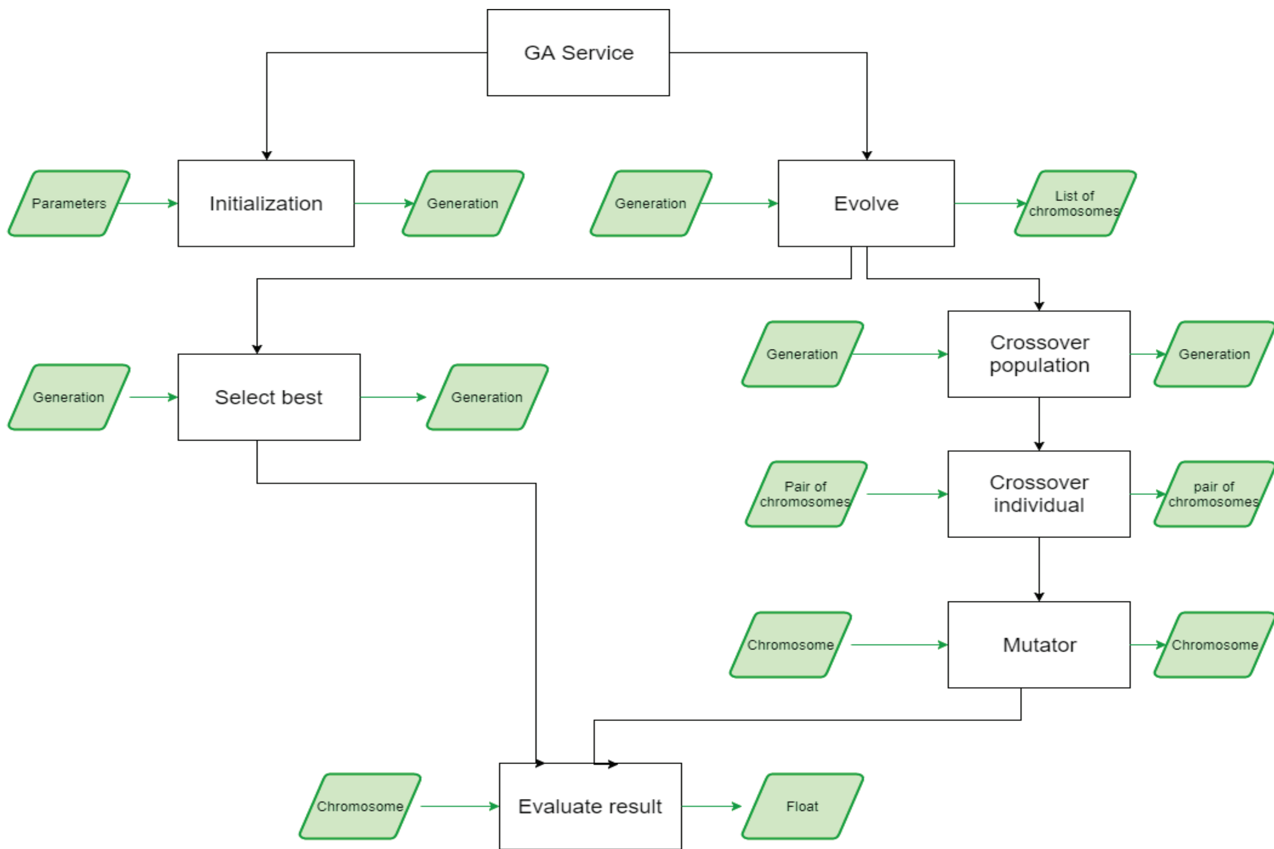

Fig. 1. Implemented microservices and their communication. The green parts present data flow (Input and output parameters of each service)

iteration randomly chooses two chromosomes in constant time, evaluates them and chooses the better one with greater probability to be part of the next generation. On the other end is the selection method which operates with $O(nlog(n))$ complexity, where n is the population size. It sorts all the individuals in the populations, and chooses the best k to continue to the next generation.

Evaluate result. This microservice is responsible for evaluating the quality of each solution, and it is specific for each optimization problem. It takes chromosome as an input, and returns a float value representing the quality of the solution.

Crossover population. After it is decided which individual will continue to the next generation, this microservice is responsible for completing the next generation by performing a crossover on chromosomes of the chosen subset. Namely, cp times it chooses a pair of individuals, and puts their offspring in the next generation. Its input is a generation, it evolves it and returns the new generation.

Crossover individual. The input of this service is a pair of chromosomes, and it implements a crossover technique, which is used to generate offspring to the input chromosomes. The return value is the generated offspring. There are many possible off-the-shelf choices for crossover operator, suitable for different applications. One of the commonly used are Cycle Crossover (CX), Order Crossover (OX) and Partially Matched Crossover (PMX) [7] [8].

Mutator. This service performs a mutation changes on one chromosome. The mutation process usually makes small changes to one or more components of the chromosome. Its input parameter is the original chromosome, and the modified one is returned after the mutation is performed. The most commonly used mutation operators are Insertion mutation, Simple Inversion Mutator and Swap mutator. [7] [8].

A. *Used Technologies*

All previously described microservices are implemented using Spring framework (Spring cloud part [9]), which contains a large number of libraries used in building a distributed system. This platform offers a simple way to deploy the application to the Cloud. Spring Cloud has powerful libraries which can be used in discovering microservices, managing the configuration, distributed messaging, load balancing etc. The mentioned libraries are developed and used by industry leaders, such as Netflix and other companies whose solutions are implemented using the microservice architecture. The libraries used in the proposed implementation are:

Eureka [10] is a part of Spring Cloud which offers discovering of the microservices, in order to balance the workload between them and to successfully manage the application

when some of the microservices are down. Eureka can be seen as a service registry.

To use this library, a service which works as a monitoring service for all other microservices needs to be created. Namely, when a new service is started (introduced to the system), it tries to register to the main Eureka service. The main task of the monitoring service is to check if each registered microservice is active and to show this information to administrator. Each registered microservice is called an Eureka client. Each Eureka client is registered with its host and port, as well as additional meta-data. The meta-data can contain service name, which can be used for robust code on the client side and easier communication between the microservices. Additionaly, when a service is re-deployed, and the service is given a new IP address, the only change happens in the Eureka service. The other services still can call the new service by its unique name, and their code base remains unchanged.

RabbitMQ library is used for secure and reliable communication between the microservices. Namely, RabbitMQ [11] is a message broker, which means that it is a module which translates the messages from the senders format to the format used by the receiver. Furthermore, the messages are exchanged in the following way: RabbitMQ builds a message queue, and then each service which needs to send a message connects to the queue and uploads the message. The message is saved until the recipient service is connected to the queue and receives the message. Further processing of the message is left to the service that received it. RabbitMQ implements many message protocols, but the most popular and commonly used is Advanced Message Queuing Protocol (AMQP).

Zuul [12] is a service used for dynamically routing, monitoring and improving security of the microservices. This service allows the inner ports of the services to be hidden, and it is used as a "front door" (known as "gate keeper") for all the client requests for other services. This means that its role is to accept the incoming requests and to redirect them to the right services. With this controlling strategy of the requests to the inner microservices it is possible to have further understanding about the health of the implemented system, as well as its protection from various malicious attacks. Zuul offers dynamically reading, compiling and executing of series of filters which are used to control the HTTP requests and responses. Zuul allows the client not to know the specific ports, but to focus on the provided API, and relies on Zuul to make the call to the corresponding microservice.

Pivotal Cloud Foundry [13] is used as a platform for deploying of the microservices. This platform is chosen because it offers many implemented solutions for microservice management, for example on demand scaling, dailover/resilience, road balancing, monitoring etc.

Fig. 2 illustrates the communication between the used technologies. Namely, the client requests first are coming to the Zuul service, because the ports and addresses of the individual microservices are not known to the user. This service distributes the request to the corresponding microservices and returns the results to the user. To achieve that, it relies on Eureka, which has a completely registry of the services. The communication between the services (including the Eureka service) is performed using RabbitMQ and its implemented message queue.

Note. All microservices communicate using the Rest Template. This allows simple request building and parsing of the answers. In the proposed implementation all the messages are in json format.

The performances of a microservice approach on the genetic algorithm implementation depends heavily on the computational performance of the evaluation function, and other specific components of GA. Namely, the network latency is a big problem, and can overpower the save in execution time by the distributed execution. That means, in order to obtain an improvement with a microservice approach, we need to test on a very complex evaluation function. Unfortunately, we were not able to do this, due to the high cost of the used platform for large scaling. Our services were deployed on the free version, with 1GB of RAM and single core CPU. Of course, for some applications, this approach will have poorer performances, but the point is that the presented approach has different advantages, for example flexibility and easy scaling up and down.

V. Conclusion

This paper shows that Genetic Algorithm is suitable for a microservice based implementation, because it consists of many independent components with different processing power needs. Advantages of this approach are numerous: for each component, information about its workload is available, leading to easily scaling of the components that need more processing power. The main disadvantage is the message based communication between the microservices, which is significantly slower compared to the in-memory calls. Furthermore, each new implementation of the crossover and mutation operators needs to be implemented as a new microservice and deployed, which is more complex and less generic that simply adding a new function in the monolith architecture.

As further work is left to develop the prototype into a more complex and fully implemented application, which offers many possibilities for different operators used by the genetic algorithm. In addition, the scaling of the components of the proposed implementation (at the moment) is done by hand, namely the administrator observes which components have high workload, and by hand initializes more processing units with this components. A way should be found to automate this process. Additionally, experiments should be introduced to examine the needed processing power, in order to achieve performance gain by the presented approach.

References

[1] A. Wang and S. Tonse, "Announcing Ribbon: Tying the Netflix Mid-Tier Services Together", Medium, 2018. [Online]. Available: http://techblog.netflix.com/2013/01/announcing-ribbon-tying-netflix-mid.html. [Accessed: 25- Mar- 2018].
[2] J.Thnes, "Microservices." IEEE Software vol 32 no. 1 pp. 116-116, 2015
[3] N. Dragoni, "Microservices: yesterday, today, and tomorrow." Present and Ulterior Software Engineering. Springer, Cham, pp.195-216. 2017.

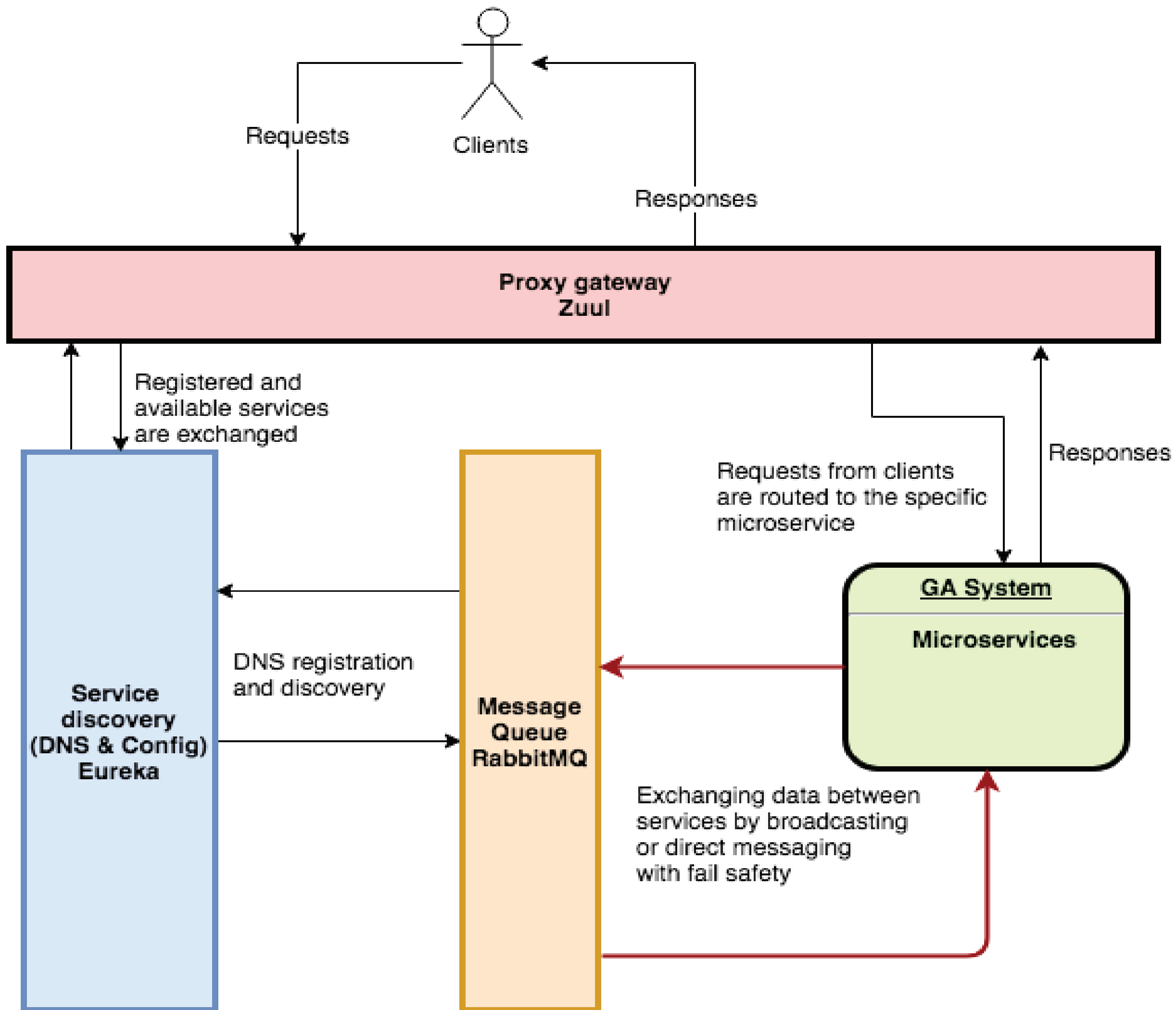

Fig. 2. Overview of the interactions between the used technologies.

[4] J. Holland, "Genetic algorithms." Scientific american vol 267 no. 1 pp. 66-73, 1992
[5] L, Kwang, and M. A. El-Sharkawi, "Modern heuristic optimization techniques: theory and applications to power systems". John Wiley and Sons vol 39, 2008.
[6] B. Miller and D. E. Goldberg. "Genetic algorithms, tournament selection, and the effects of noise." Complex systems vol 9 no. 3 pp. 193-212 1995
[7] S.N. Sivanandam and S. N. Deepa. "Genetic algorithm optimization problems." Introduction to Genetic Algorithms. Springer Berlin Heidelberg pp. 165-209, 2008.
[8] L.Y. Kwang, and M. A. El-Sharkawi, "Modern heuristic optimization techniques: theory and applications to power systems." vol. 39. John Wiley and Sons, 2008.
[9] "Spring Cloud", Projects.spring.io, 2018. [Online]. Available: http://projects.spring.io/spring-cloud/. [Accessed: 25- Mar- 2018].
[10] "Introduction to Spring Cloud Netflix - Eureka — Baeldung", Baeldung, 2018. [Online]. Available: http://www.baeldung.com/spring-cloud-netflix-eureka. [Accessed: 25- Mar- 2018].
[11] "RabbitMQ - Messaging that just works", Rabbitmq.com, 2018. [Online]. Available: https://www.rabbitmq.com/. [Accessed: 25- Mar- 2018].
[12] "Netflix/zuul", GitHub, 2018. [Online]. Available: https://github.com/Netflix/zuul. [Accessed: 25- Mar- 2018].
[13] "Pivotal Cloud Foundry (PCF)", Pivotal.io, 2018. [Online]. Available: https://pivotal.io/platform. [Accessed: 25- Mar- 2018].

View publication stats