UDC 004.75

HANDLING WITH A MICROSERVICE FAILURE AND ADOPTING RETRIES

O. Dmytrenko[1, a], M. Skulysh[1, b]

[1] *National Technical University of Ukraine «Igor Sikorsky Kyiv Polytechnic Institute», Institute of Telecommunication Systems*

Abstract

This article deals with the problem of controlling the load on microservices. Various options for preventing overloads, their advantages and disadvantages are analyzed. The method for flexible control of re-sending requests to microservices is proposed. The requirements for QoS were taken into account and focused on adaptive queue management.

Keywords: distributed networks, retry mechanism, microservice communication, circuit breaker, microservice failure

Introduction

Nowadays, distributed systems have become very widespread in modern IT products. The distribution can take place in both parts of a product — in the program part or in the database part. When separate functionalities are located in standalone pieces, this is called a microservice or a lambda, a serverless technology. When it comes within a DB distribution, typically, NoSQL is the technology at the back end. Even if it is an SQL DB, it should comply with the CAP theorem. In this article, we highlight existing problems with the distributed systems and propose some ways to handle them.

1. Problem

In theory, if the nodes a system consists of are functional, then the main problem is tuning them appropriately and making sure that the system functions in accordance with requirements. However, in practice, a system needs to be prepared for failures, as some nodes have to be replaced over time, outages might take place, and the network could suffer certain disturbances. In addition, the nodes can be overloaded and either respond with long delays or not respond at all [1]. Figure 1 presents an image showing the two microservices and the connection from one to another. Let us consider some approaches to handle these situations and the ways for the system to recover.

2. Existing Methods of Handling Service Unavailability and Ways to Improve Them

In this paragraph we show the existing methods to work with the issue of microservice overload with the requests and propose their enhancements, providing the results and statistics collected from the real data.

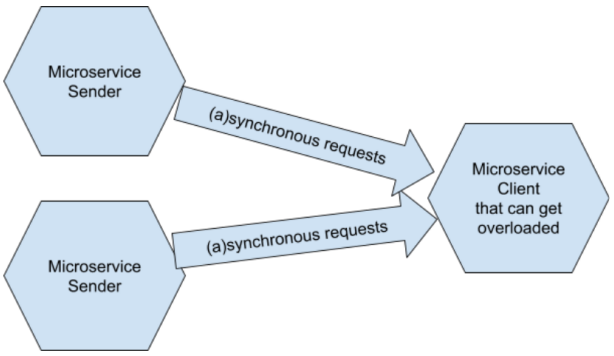

Fig. 1. Microservices and connection between them

2.1. Circuit Breaker

A leading idea that can be implemented in the distributed systems with different parts dependent on each other is a *circuit breaker*. The idea is to determine the specific moment in time when something goes wrong, and a part of the system gets overloaded, which might lead to unavailability of the whole system [2]. In this case, the node should be temporarily stopped from being used, so that it can recover and return to the normal state afterward.

2.2. Fixed Rate Retries

Another way to overcome unavailability of certain system parts is to send retries at a *fixed rate*. A standard mechanism is to call the service with certain periodicity over time. This strategy is best suited to the case with a limited number of services that perform the retries, and if the system can recover in a short period of time.

2.3. Exponential Retry Algorithm Adopted to the Business Use Case

An *enhanced retry algorithm* suggests performing the retry calls in an *exponential* manner [3]. According to an exponential backoff formula we used, periods of time are successively increased by a factor of 3, starting with an initial delay of 1 minute. Several first retries

[a] olexandra.dmytrenko@gmail.com
[b] mskulysh@gmail.com

should be done in a short period of time (during the first several minutes), but as the addressee continues to ignore the requests, the timeout should be prolonged exponentially. For example, the sequence of delays can be approximately as follows: 1 m, 3 m, 9 m, 27 m, etc. The process can be described by the formula below:

$$D_{i+1} = k \cdot D_i \ ,$$

where D stands for delay, k is a coefficient, and i is the number of a retry. In the example above, $k = 3$ and $D_1 = 1$ minute.

Speaking about the exact numbers and received data, we had a maximum of 20 concurrent requests, but only 4 could be processed at a time. As far as maximal load could happen only several times during the working day, the requests that were not accepted at once succeeded in the next 2-4 retries. Once we experienced a substantial network break that lasted for around 36 hours. Our retries were set to last up to 24 hours. so a massive amount of work was lost. Further we came up with an enhancement about which we'll speak later.

2.4. Exponential Backoff Algorithm with Rate Limiting and Its Usage

In practice, exponential backoff works much better and does not overload the client. Yet, if the algorithm does not succeed, at some point the delay will reach 36 hours, which is too long, and the system could be renewed long before that. In this case, *rate limiting* can be applied [4]. The time for exponential retries can be bounded from above, say, by 24 hours, or 7 retries. After that, the rate limit can switch back to the standard one, equal to 12 hours. Depending on the probability of system recovery, making checks every 6 hours after 6 retries also sounds rational. Here, the number six is derived from the workday of employees who may have noticed and reported the service outage and the time it will take them to fix the issue before their working day ends or the new working day starts.

2.5. Flexible Retries Based on the Response

Another enhancement of this algorithm that can be proposed is to take into consideration the *type of an error*, returned on the request. For example, if it is known that the server is off (e.g. error 404), the outage could take place which might not last for long [2]. Sending requests won't influence the receiving side, so they could be sent relatively often. When the reason is respondee's overload, the first retries should be sent in a while, in 10 minutes or more.

2.6. Batching for the Retries and Their Results

As long as the system performs normally all the time, new requests continue to be sent and are left without responses. Making retries to each request might cause overhead not only to the receiving node, but also to the sending one. Making retries for each request and collecting them requires a separate resource. It would not be safe to store them in-memory, especially if there can be many of them. It is much more reliable from a long term perspective to store those in the DB or a queue. An optimisation that could be made is to send *requests in batches*, i.e. collect all the request bodies in one batch and send it to a client once [5].

This implementation substantially reduced the number of requests. The time it took to process a request out of 100 subrequests was several times longer, but took less than 6 seconds compared to 1-2 seconds of processing a singular request. Request timeout is set to 10 seconds, so collecting unsent requests into batches increased the performance a lot and made the logs look much cleaner. The downside of such an approach is that not every kind of request can be organized in batches and additional functionality should be implemented on the client side to be able to process batch requests.

2.7. Need for the Retries, System Consistency and Following Investigations

When dealing with retry mechanisms, a question of data actuality arises. If the retry succeeds only in a day or two, the data might have lost its actuality by that time. Specific solutions depend on the type of business that uses the project. If this is an online ticketing service, the successful operation should be marked before the buyer should take any actions connected with the event. Waiting one day in the absolute majority of cases is acceptable. When the retry is about delivering a lost movie package that has no sense without the previous and the following piece, there is no sense in it.

From personal experience, we find that when it is too complicated to have the system in the consistent state all the time, it is acceptable that the system parts synchronize periodically, e.g. once per day during night time. Such a solution may not only be acceptable for the business, it can also be advertised by the managers as an advantage of the system, e.g. cell phone operators created contract subscriptions where the state of account is renewed at night and give possibility to the clients to use the phone without limits till the end of the day even having big debt.

Our following investigations we want to perform for the chain of microservices where one calls the other, the request which fails is sent by the most deep microservice and the response for the failing request is needed at the start of the chain.

3. Preventing Microservice Unavailability

When replacing the monolith architecture of the system with a microservice one, product owners need to be aware of the advantages of this decision and have substantial reasons to perform the replacement, because in general such a system will be more complicated and harder to develop. However, one of the main reasons for such a design is scaling certain parts of the application that may be happening in a cloud environment or in the owned server rooms. In order to create another instance of the microservice, there should be understand-

ing, when it's already the time, so that the resources would be wisely used.

The characteristics of the hardware, e.g. amount of memory, CPU frequency, and the number of processors might be different for different instances of the same microservice. This means that every instance has a different limit of a request number. Ideal situation is to prevent the service unavailability instead of making some retry logic, maybe even creating a separate retry microservice not to affect all the top microservices that participated in the request creation, and inventing effective ways to inform the user later about the result of their operation.

3.1. Real Time Adoption to the Changing System Characteristics and Requests Number

In order to count the *congestion window size*, or the maximum amount of requests that can be sent to a system at a time, one needs to constantly send more and more requests measuring the latency of the system, i.e. the response time. When the latency starts to grow, it is time to lower the request rate lest the system fails [6].

In order to be able to control the frequency of the requests, a queue is needed that will buffer them. If the microservice works at the maximum rate and the requests queue is growing, its new instance could be created. The new requests should be directed to the new microservice.

Conclusion

Having a microservice architecture or messaging between services can become a significant issue unless it is wisely organized. A mechanism like a circuit breaker can help in preventing the service failure or give the possibility to the service to renovate. Retry mechanism will take care to deliver the required information where it has to belong. At the same time, it is better to prevent the problem in the first place. Constant counting of the congestion window in the services whose hardware characteristics can change depending on the load will guarantee the service constant availability and provide understanding if there is need to extend the service resources.

References

1. Hora: Architecture-aware online failure prediction / Pitakrat T., Okanovic D., van Hoorn A., and Grunske L. // Journal of Systems and Software. — 2018. — Vol. 137. — P. 669–685.
2. Shunmugasundaram S. Patterns for the distributed systems in cloud - part 2. — 2020. — Access mode: https://medium.com/@vagees/patterns-for-the-distributed-systems-in-cloud-part-2-3176a4f67b2.
3. Scaling Exponential Backoff: Constant Throughput, Polylogarithmic Channel-Access Attempts, and Robustness / Bender Michael A., Fineman Jeremy T., Gilbert Seth, and Young Maxwell // J. ACM. — 2018. — Vol. 66, no. 1. — P. 1–33.
4. Sun X., Dai L. Backoff Design for IEEE 802.11 DCF Networks: Fundamental Tradeoff and Design Criterion // IEEE/ACM Transactions on Networking. — 2015. — Vol. 23, no. 1. — P. 300–316.
5. Li J., Bao Z., Li Z. Modeling Demand Response Capability by Internet Data Centers Processing Batch Computing Jobs // IEEE Transactions on Smart Grid. — 2015. — Vol. 6, no. 2. — P. 737–747.
6. Landau E., Thurston W., Bozarth T. Performance under load. — 2018. — Access mode: https://netflixtechblog.medium.com/performance-under-load-3e6fa9a60581.