MCIT
Modeling, control & information technologies

Development and investigation of adaptive micro-service architecture for messaging software systems

https://doi.org/10.31713/MCIT.2021.13

Gamzayev Rustam
V.N. Karazin Kharkiv National University
Kharkiv, Ukraine
Rustam.Gamzayev@karazin.ua

Shkoda Bohdan
V.N. Karazin Kharkiv National University
Kharkiv, Ukraine
xa11867771@student.karazin.ua

***Abstract* — Messaging Software systems (MSS) are one of the most popular tools used by huge amount of people. They could be used for personal communication and for business purposes. Building an own MSS system requires analysis of the quality attributes and considering adaptation to the changing environment. In this paper an overview of existing MSS architecture was done. Data model was developed to support historical and real time data storage and processing. An own approach to build Adaptive Microservice MSS based on the messaging middleware and NoSQL database was proposed.**

***Keywords* — microserivce; messaging; system; architecture; software.**

I. Introduction

Development of the Messaging Software system (MSS) is not trivial and requires considering complex approaches in the modern scalable architecture design. With the invention of the World Wide Web, humanity has taken a big step in information technology and in social relationship. Using it, each person can look for information that meets his needs, to communicate, using smart devices and IoT. After an amount of devices connected to internet have increased, and collaboration relationships of different software systems become more complicated it was necessary to modify approaches and methods to develop software systems.

There are presented a lot of different approaches that are used in building of architectures and design. Early development started with single big applications that were easy to maintain at the beginning but after a time when it was becoming bigger and bigger it was obvious that once it would be impossible to manage this big chunk. In a nutshell, it was a monolithic architecture.

Building monolithic applications can be very expensive and hard to support. A big application that contains the whole logic is also hard to scale. For example, if this type of an application can be deployed in 10 and more minutes, it is hard to imagine how fast and safe can the application be restored in the case of unexpected issues [1]. Though it can be really fast to develop at the beginning as you don't think about most of the issues but it is not a good solution for the future if we talking about the development of messaging systems that process more and more data from time to time.

When developing MSS we need to consider the following parameters [2]:

- reusable;
- lightweight;
- easy to scale;
- independent.

One of possible approaches for software development that will meet such quality attributes is a microservice architecture. Microservice architecture is an architectural style which structures a system as small, loosely coupled services. Simply, we can imagine this architecture as a decomposed monolith. Each of those services have a single responsibility for specific domain models. It allows us to deliver mostly each part independently. Moreover, it gives an opportunity to build scalable high load systems which are the main aspects of messaging applications. But we need to consider and reimagine much more that we could omit earlier. After some time this architecture became one of the most popular. Though it is used in the web sphere mostly it is hard to find an application that does not use this approach nowadays. There are a lot of studies but the one that is done in [1] describes microservice architecture usage in cloud native systems as they are full of distributed computing. From it we can see that after some time microservice architecture was started to be used in different processing models and it will be used for a very long time for sure.

The purpose of the work is to analyze, study high-load architecture, existing solutions, practices, approaches and development using a highly loaded architecture with micro-service solutions to maintain and ensure the security of a large amount of data, not having a large amount of hardware resources.

II. Related work

Though microservices mostly appeared in 2014, since that time there were published an enormous

number of papers and this number extremely goes up even now.

In [3] was conducted a research regarding architectural models and technologies and focused on common problems, pitfalls and main algorithms what should be followed in order to design and develop a system that using a microservice architecture.

Comparing to the monolith architecture, security layer is harder to implement in the microservice architecture. Having multiple services in the network gives us multiple APIs and multiple attack points which can be accessed in case of not very secure system. In [4] was conducted a research regarding microservices security. Attacks, processes and other things were described which can make any microservice system vulnerable. Also, a lot of approaches were presented on how to organize the process of securing microservice architecture.

Working with data can be also challenging. Having a big model that is divided to each microservice and its data source creates new ways on how to process, obtain and aggregate data. In [5] a research described the ways of managing data in microservice architecture. Moreover, the architecture generates a new ways of data communications using HTTP or asynchronous methods of communication (by using RabbitMQ or Kafka).

III. Analysis and investigation of existing messaging systems development methods

Messaging software systems become very popular in the last 10–15 years. One of those was ICQ. First ever created prototypes didn't have such a variety of functions comparing to the modern solutions but still it was very helpful.

Every day something new is developed. When smart devices become an integral part of our community, a society faced new problems which should be solved. The marketplace is growing very fast and provides such solutions from time to time. It's hard to imagine a messaging application without audio or video streaming functionality nowadays though in the past we had a possibility to exchange with messages containing different types of information. For now, there are a lot of prototypes that are made for specific environment. It can be a business area's solutions which were developed for big companies or it can be open solutions that connect people all around the world.

In this paper some of the most popular solutions were analyzed to get the idea of how the messaging service works under the hood to make a generic approach and design on how to build MSS. Those were Slack, Avito Messenger, Viber, WhatsUp, Discord.

Slack is an enterprise messenger written using a PHP programming language and a MySQL database. The algorithm of this messenger can be described by the following steps:

1) A request is made to the server (authentication), which returns a token for interaction. The session starts for the user;

2) MySQL databases are searched. There are a lot of them and they all contain information on chats and users.

3) A server looks for the user's shard and when it is found a user will be able to interact with an application.

For a data transfer, Slack uses an analogue of a websocket protocol that is developed by a Slack team.

The entire database of the application is divided into many databases containing the same structure and different data (sharding) – the principle of database design, in which logically independent rows of the database table are stored separately.

Because MySQL is used, the team was faced the situation that when it was necessary to scale the system, it became very complicated and expensive due to sharding architecture. When a new node was added, many other nodes stopped working properly, which caused the system's performance to drop and had to be refactored. Due to the closely related structure, the team had to spend a lot of time reworking the architecture [6].

Avito Messenger is a messenger for users of the website Avito – an online service for placing ads about goods, real estate, work vacancies etc. and as well as services from individuals and companies. Messenger also uses sharding, but with a different NoSQL database – MongoDB. A total of 8 replica sets are used, which are independent, as well as shards by user ID. This messenger allows to have non-group chats between only two users. To store messages, 2 shards are used - for the recipient and for the sender. The algorithm of sending the message is next: firstly, the message is sent to the service API component, which immediately writes it to the sender's shard. The message then goes to the service-db-store component and is stored for the recipient [7].

Next investigated messengers were Viber and WhatsUp. Those are primarily mobile messengers but can be also accessed from desktop clients. Viber and WhatsUp messengers use local data storage. That is, if the message was delivered to the recipient, it is not stored on the server, but stored only on the recipient's device. In the case of WhatsUp, if a message is not delivered to the recipient, it is stored on the server for 30 days. Then, if it was not sent, it will be deleted from the server and will not be received in the future. Viber does not store messages on the server at all [8, 9].

And the last very popular messenger that was investigated was Discord. Discord is a free messenger with support for voice over IP (VoIP), video conferencing. Mostly, it is developed for gaming purposes such as streaming, recording etc. Initially, a MongoDB database was used in an architecture. Everything in Discord was specially stored in a single replica network, and messages were stored in a collection with a single composite index on channel_id and created_at. Over time, the limit of 100 million messages in the database was surpassed, and then problems began to arise: long delays, big chunks of non-managed data – etc. It was decided to move to CassandraDB because:

1) the read / write ratio of the database is approximately 50/50.

2) linear scalability (transferring data to another shard after reaching the limit was not an option).

3) the ability to generate readings by criteria, such as the last 30 days [10].

After the deep analysis of those prototypes it was decided that the approach that was used in Discord met all our requirements and it was used in the next part – a design of the messaging system.

IV. Design of a messaging system

Messengers process a lot of messages in real time so a correct and fast communication protocol is a must. There are a lot of different protocols such as HTTP that are used in the modern web. HTTP is an application layer protocol for data transmitting that uses requests. The basis of this protocol is a client-server technology, where the server is a provider that expects requests from clients, and clients are who send requests to the server [11]. But having a lot of HTTP requests sent to a server can make an overload and slow down the processing if we try to send messages using this protocol. There should be used a protocol that allows us to process data asynchronously and without a need of waiting for the response from the server if we talk about messaging sending. Something that is similar to websocket can be used as it allows to initiate a bidirectional connection between a client and a server to process data. Though there are a lot of modern solutions for real time messages processing, it was decided to use a STOMP streaming protocol that is built on top of websocket for the messaging sending and an HTTP protocol for simple requests such as user registration/login, information editing etc.

STOMP is a real-time data transfer protocol that is very similar to HTTP and runs on top of TCP and websocket protocols using commands for real-time connection and communication [12].

As those two protocols by default are not secure, TLS will be used in order to not expose any data to the real world and non-authorized users. An complex encryption mechanism was not included as the main purpose was to research design and performance sides

Because in a microservice architecture we can have many independent services the main task is also to set up communication between these services, as data or business logic can be broken down between different services. Typically, synchronous communication is used through the HTTP protocol. This means that we will wait for the result of the request to another micro-service until we receive a response. If a service takes some time to process a request, it can block the entire system or individual services, which affects performance.

In this case, very often use asynchronous principles that allow you to give a specific task for processing to the queue, and when the result of the task is ready, a user will receive it. It was decided to use both technologies to support asynchronous communication – RabbitMQ and Apache Kafka.

RabbitMQ was only used for the data protocol because the STOMP protocol's queues are slower than RabbitMQ queues. It increased a message processing's performance a lot.

Apache Kafka was sed to interact between consumer / producer services. These two types are responsible for the basic logic of chat, for example sending messages, joining chats. Once the message has been sent from the user interface and read from the RabbitMQ queue, it is in the first stage - the processing stage. The service checks whether the message can be sent to the chat (whether the user can send the message to the chat, whether he is connected to this chat etc.) and checks the structure of the message. If everything that user entered is correct, the message is written to the database tables and sent to the Kafka queue. From this queue, the message will be taken and sent to the main chat by the consumer service.

Because the data in a storage appears quickly, retrieving this data from tables for example from SQL databases can become very slow. As said earlier, it was decided to use a similar approach that is used by a Discord team – using a Cassandra database.

Apache Cassandra is a distributed database management system related to NoSQL types and is designed to create highly scalable and reliable storages of huge data sets. Because messengers require a huge storage to save messages into, it is a must to have a right way in order not to slow down a system as a number of messages increases. With Cassandra, it is possible to create a separate structure for storing messages of each chat with the subsequent sectioning of this data by time sections. This means that separate sections will be created for each chat, in which messages for a specific period and a chat will be stored. This specific time period is determined by the year and month.

Additionally, for the application security it is essential to have authentication and authorization processes. Authentication and authorization processes are used to prevent unauthorized access. Authentication is the process of recognizing an individual. Authorization is a process that allows you to give permission to the client to perform certain actions. Because developing an application security layer is very complex from the beginning and time consuming, it was decided to use already developed solution Keycloak to organize these two processes.

Keycloak is a single sign-on server with the ability to manage the status of a connected client. With its help, it is also possible to organize the process of user registration. It also allows you to configure special user groups and grant them rights to use certain system functionality.

And the last important part through which a user interacts is a web interface. Also, the web application must monitor states of chats. A Redux library allows to manage states of web applications. And the main single application page's structure was developed using the React.js library. A collection of ready-made graphic components that were used were provided in the Ant Design library.

If we combine everything together, we shall have a similar architecture that is displayed on the figure 1.

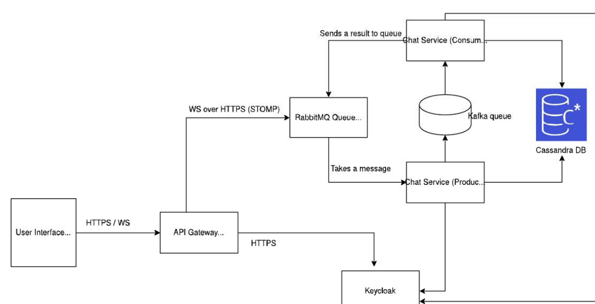

Figure 1. Messaging system architecture

Therefore, based on the done research, it was shown that developing MSS is very hard and needs a deep understanding. Using microservice architecture it was shown that additional actions were needed to settle drawbacks and make a system highly available. In order to work with any amount of data a data storage model was introduced and selected based on the done research of the modern messaging solutions. Various communication and security methods were used to achieve the maximally optimized system.

REFERENCES

[1] Pachghare, Vinod. (2016). Microservices Architecture for Cloud Computing. *Journal of Information Technology and Sciences*. 2. 13.

[2] Bushong V., Abdelfattah A. S., Maruf A. A., Das D., Lehman A., Jaroszewski E., Coffey M., Cerny T., Frajtak K., Tisnovsky P., Bures M. On Microservice Analysis and Architecture Evolution: A Systematic Mapping Study. Applied Sciences. 2021, 11(17):7856.

[3] Gamzaev R. O. Architectural models and technologies of microservices in order to increase the adaptability of variability in the development of lines of software products [sciences. pr.]: materials of the XV international scientific-practical Internet conference (m. Kiev, 11 chervnya 2021 r.). Kiev, 2021. P. 224–233.

[4] Mateus-Coelho, Nuno. (2020). Security in Microservices Architectures.

[5] Damyanov, Ivo. (2019). Data Aggregation in Microservice Architecture. International Journal of Online and Biomedical Engineering (iJOE). 15. 81. 10.3991/ijoe.v15i12.11095.

[6] Keith Adams. How Slack Works, Youtube, URL: https://youtu.be/WE9c9AZe-DY

[7] Alexandr Emelin. Avito Messanger Architecture // Youtube, February 26, 2020. URL: https://youtu.be/4tIS58sQ7Mc

[8] Bringing modern storage to Viber's users: Google, July 1, 2020. URL: https://android-developers.googleblog.com/2020/07/bringing-modern-storage-to-vibers-users.html

[9] Privacy Notice: WhatsUp, 2021. URL: https://www.whatsapp.com/legal/updates/privacy-policy/?lang=en

[10] Stanislav Vishnevskiy. How Discord Stores Billions of Messages: [Електронний ресурс] Discord Blog, Jav 14, 2017. URL: https://blog.discord.com/how-discord-stores-billions-of-messages-7fa6ec7ee4c7

[11] MDN contributors, "An overview of HTTP", URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview

[12] Jeff Mesnil, "STOMP Over WebSocket", URL: http://jmesnil.net/stomp-websocket/doc/