Praktikum: Cloud Data Bases
Final Report

Aly Kamel*
aly.kamel@tum.de
Technical University of Munich

Iremnur Kidil*
ge34hoz@tum.de
Technical University of Munich

Ricardo Kraft*
ga84vax@mytum.de
Technical University of Munich

ABSTRACT

In today's world, communication is key between people to retrieve information which is substantial. Almost every software system includes a messaging platform nowadays in order to ease their clients' lives by offering a chat service that enables sharing essential information. On this basis we extended our program with a group chat feature, where multiple clients can communicate with each other via chatrooms.

This report begins with an introduction part, where different databases are examined. Depending on our observations of how the commonly used database systems work, our motivation of developing the group chat extension is elaborated.

Moving on, the background containing CAP Theorem, ACID and BASE design approaches are clarified for the sake of a better understanding of the developed distributed and replicated database system.

Furthermore, our extension is examined focusing mainly on how the system works from the client's point of view. The features of the group chat and the implementation is studied in greater detail. In the implementation section a brief description of the replicated and distributed storage service from Milestone 4 is represented. The replication strategy is an important feature of database systems in general because it guarantees eventual consistency and basic availability which is crucial for distributed database systems.

The main idea with the group chat extension is that multiple clients can join a chatroom with a chatID, assigned individually for every different room, and exchange messages there. Moreover, clients can perform read and write operations with the help of a chatbot while being in the chatroom.

Eventually we conclude our paper with a performance analysis by comparing results from our performance measurement tests and touch upon the advantages of providing a group chat.

1 INTRODUCTION

To develop a key-value database, one of the main qualifications to consider is, how to handle large amount of data while making the system available for multiple clients. Currently, there are lots of large-scaled cloud database systems, such as Amazon Web Services, Microsoft Azure SQL or Oracle Database, that have a system consisting of properties: scalability, cost effectiveness and latest technology availability.

Another property that key-value stores such as BigTable and PNUTS maintain in their system is elasticity. Elasticity is measured whether a system can be scaled-up dynamically by adding more nodes or can be scaled-down by removing nodes[2]. The feature is provided in our final program by clients being able to add any number of servers to the system or shutting them down.

In the following, some of the popular key-value databases are described for the purpose of discussing the design approaches and compare them with our system.

Amazon DynamoDB is a non-relational database system, which provides low latency, to be more specific single-digit millisecond latency[1], considering the system is used all over the world. Data records are stored with a primary or composite key and several attributes depending on client's request[7]. The only constraint is that these records, including attribute names and attribute values, cannot exceed 400 KB[1] which is rather a large space for storing the provided data.

Apache Cassandra is another key-value based high scaled[**?**] database system, which is known for managing some of the world's largest datasets on clusters with the help of thousands of nodes distributed amongst multiple data centres [5]. For the professionals Cassandra is the first choice since it offers more flexibility regarding fault handling and managing wide range data, unlike BigTable and Dynamo[7]. The system focuses on high level of availability by scaling millions of read and write requests per second[5].

Another example, and one of our biggest inspirations, is Redis. This key-value store rose in popularity after its creation in 2009 and got deployed by many big companies such as Instagram [8] and Twitter [11], which require gigantic databases and fast responses to the millions of users they serve. One of its multiple features is the Pub/Sub system [10], added in March 2010 [9]. Using that, clients can choose specific keys they are interested in and receive an update whenever their values gets changed. The way this is modeled is that the Redis clients keep waiting on their sockets for the server to send them an update. During this period, while publishers are allowed to execute other commands normally, subscribers are limited to only subscription related operations. In a way, it allows really basic communication between the different clients.

In our implementation, we wanted to extend on this idea, but make it so that clients are not blocked during a subscription session. The main use case of our system, mainly accessing and manipulating key-value pairs, should still be accessible, even while a client is subscribed to channels. Also, we did not want to distinguish publishers from subscribers. All clients who are interested in a key should be able to exchange information with each other.

And not only should subscribers be able to store and obtain information from the database, but also send back information to the publishers. So the way we modelled it in the end, was having a many-to-many relationship to all other subscribers on the same key, effectively, implementing a group chat functionality. Subscribers on the same key represent users in the same chatroom.

*Authors contributed equally to this work.

By observing the provided features of above-mentioned databases, one can say that our program is a very small scaled version of them. It provides the core functionalities such as providing elasticity, responding the client with consistent values as fast as possible, being available within a certain amount of time, and maintaining resilience towards partitions in the network.

In the following section our motivation of creating a group chat is explained beginning with a brief description of how our system is built up during the Cloud Data Bases Praktikum. Nevertheless our report focuses mainly on the detailed analysis of our extension and evaluation of the performance tests.

2 MOTIVATION

Cloud Data Bases Praktikum started off with a simple implementation of a client with the aim of a better understanding of the client/server architecture and socket programming. Nonetheless, the system is developed with implementing a one-to-many relationship being one server and multiple clients. The program is improved with a basic key-value storage service extension for clients to perform read and write operations with the help of mentioned storage server.

In the following milestones, the architecture is enhanced to a many-to-many relationship consisting of multiple servers and multiple clients. To manage the servers an External Configuration Service is created, hence the servers which are placed on an hashring could communicate with each other. Following that replication strategy is added to the system in order to maintain availability, consistency and partition tolerance.

In the last step, namely Milestone 5, we extended our program with a group chat feature. In the decision process, our initial goal was to come up with an idea that the clients can make the most use of.

We started with examining the features of various database systems to get inspired. Thereafter, we opted to have a chat system, where multiple clients can enter a chatroom and exchange messages. All messages send in the same room will be sent to all active members of that chatroom. The essence of providing chatrooms for clients is for them to retrieve information in a short amount of time by being able to ask their questions other members of the same chatroom.

In our implementation, we wanted to extend on Redis' Pub/Sub idea but make it so that clients are not blocked during a subscription session. We wanted to maintain the main use case of our system, mainly accessing and manipulating key-value pairs, even while a client is subscribed to channels. Also, we did not want to distinguish publishers from subscribers. All clients who are interested in a key should be able to exchange information with each other.

And not only should subscribers be able to store and obtain information from the database, but also send back information to the publishers. Thus, the way we modelled it in the end, was having a many-to-many relationship to all other subscribers on the same key, effectively, implementing a group chat functionality. Subscribers on the same key represent users in the same chatroom.

Going into more detail, we decided to have two different chatroom options, being public and private rooms. In a public room, clients can exchange more general information, such as TODO. In contrast, private rooms are more suitable for sharing sensitive information, such as credit card passwords. Since a private chatroom can only be accessed via a password, clients are ensured to keep their messages safe.

Since clients can perform read and write operations while chatting, we wanted to have a chatbot to handle such requests. All messages, including the commands put and get, are directed to the chatbot which is responsible to access to the database and perform the requests.

3 BACKGROUND

In this section, we elaborate on some of the basic concepts that our key-value system depends on. First, we discuss the CAP theorem and analyse where our system lies on that spectrum. Then, we mention two design paradigms, namely BASE and ACID, and explain the differences between them.

3.1 CAP Theorem

According to the CAP Theorem, distributed databases have three substantial properties to consider: consistency, availability and partition tolerance[3]. Eric Brewer, the man behind the CAP theorem, stated that a distributed database can fulfil at most two of three properties[4]:

- Consistency:
  Clients are provided with fresh data, meaning the most up-to-date version of the data that got stored after the last write operation.
- Availability:
  Servers respond to the request of clients at all cases. Every non-failing node in the system must be able to serve the client in a reasonable amount of time[6].
- Partition tolerance:
  Whenever a server crashes, the rest of the system will still stay functional after the crash, without any information being lost. This is mainly secured by replicating data across multiple servers.

12 years later, Brewer mentions that designers do not have to abide strictly to the 2 of 3 principle, it is rather a spectrum than binary[3]. In other words, a distributed database system can favour high level of consistency and partition tolerance by having low level of availability. Thus, the initial theorem is improved by not having to sacrifice availability completely in this case.

There are two design approaches for distributed database systems, namely ACID and BASE. According to Brewer, these two design approaches may be referred as opposites of each other[3] because of their priorities and use cases. ACID approach is used most of the times for relational database systems (SQL) and focuses on consistency to maintain reliability. On the contrary BASE is more suitable for the non-relational database systems (NoSQL) concentrating on providing the client high level of availability[4].

3.2 ACID

ACID is a traditional design approach[3] when it comes to large-scaled distributed systems. The main goal of ACID is that despite the system having partitions, the client should always be provided

with consistent values. It is an acronym which stands for the four following properties:

- Atomicity:
  A set of transactions succeed all at once or fail all together. In other words, it is an all or nothing strategy.
- Consistency:
  The system never contains any stale data. When a client wants to read from any server, the returned value must be the value from the last write commit by any client. In other words, all clients should always get the same result whenever they try to fetch the same information.
- Isolation:
  All transactions happen isolated from each other and do not affect each other. Thus, if one transaction fails, all others are undisturbed.
- Durability:
  Once a client is informed that a transaction has been successfully committed, even if a crash occurs, the transaction would still stay committed. This is ensured by persisting data permanently on disk.

3.3 BASE

BASE, another design approach defined by Brewer, offers looser requirements than ACID[3]. Non-relational database systems concentrating on high availability make use of the BASE approach, which sacrifices strong consistency in favour of being always accessible. Examples include huge storage services such as Amazon's Dynamo, Facebook's Cassandra or Google's BigTable, where millions of active users always expect the service to be available. Nevertheless, there must be trade-offs. By not guaranteeing the consistency at all time, the system cost is reduced, and clients are happier, but a client might not get an immediate response to his/her request.

With Milestone 4, replication is added to the system with the intention of increasing the availability by distributing the data records to the two replica servers. Redundancy comes along with the replication strategy, however when a node crashes or fails, read operations can be processed via replicated servers.

- Basic Availability:
  Clients are guaranteed to get a quick response from the server without getting blocked, however the returned value may be stale, stated in other words, inconsistent with the latest version.
- Soft State:
  Since stale data is permitted, servers never truly know whether they are currently up-to-date or if they still contain invalid data.
- Eventual Consistency:
  If there are no updates in the system for a long time, then all servers will gradually become consistent. Since the time is not specified, servers are always in a soft state, as explained above.

Our system conforms to the first property of BASE by being generally available, because servers respond to clients requests even if a latency occurs. The group chat extension, later to be discussed thoroughly, preserves the same requirements.

Eventual consistency is achieved within the system due to requirements of Milestone 4. If there are no updates for a long time all the replicas will become eventually consistent by updating key ranges of the coordinator and replica nodes placed on an hashring. In order to maintain strong consistency as described in 3.2, a client performing a PUT operation would have to also wait until the value gets sent over to both of the servers replicas, which would result in slower PUTs. Since our system is not designed to handle important real-time transactions, such as in a banking system, eventual consistency satisfies our database needs while allowing faster response times and higher availability.

Since the implementation of Milestone 3, it is possible to monitor key-value stores continuously. The ECS pings each key-value server every 700 ms to be informed about the servers' availability. If a server is does not respond to the ping within a given period, it is considered shutdown. With replication active, single failing server nodes can be tolerated, thus partition tolerance is provided as well. Since each piece of data exists on three different servers, the only way information can get permanently lost is if all three responsible servers crash simultaneously with no time in between to transfer data to a running server.

One of the downsides of our system is that the chat system runs just one of the requested servers by the client. Thus, partition tolerance may not be covered in case of that serving crashing. As in the most database systems there are trade-offs between the properties: availability, consistency and partition tolerance.

4 KEY-VALUE SERVER

During the first four milestones, we were tasked with developing a distributed key-value storage system, which includes many features such as replication and caching. In this section, we focus on the implementation behind some of those properties.

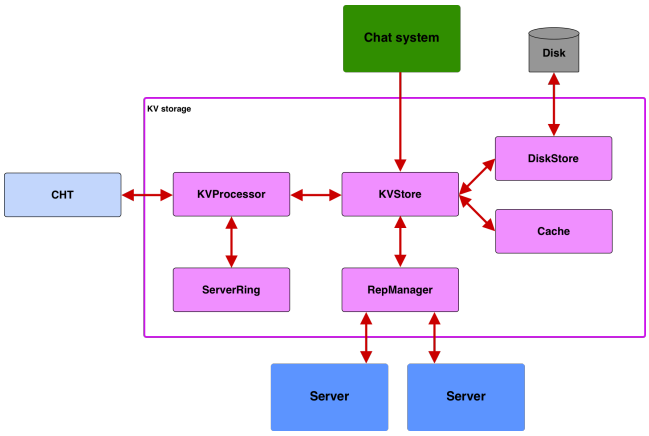

Figure 1: Client side architecture

4.1 Key-value store

Key-value stores are perhaps the simplest type of a NoSQL database. Values, which in our case are limited to strings, can be easily stored and retrieved just by providing the key they were assigned with.

4.2 Persistence

4.3 Caching

5 GROUP CHAT

The developed system is based on a client/server architecture, which uses a many-to-many relationship. The program enables multiple clients to communicate with multiple servers. Besides replicated and distributed storage service, with the created extension clients are now able to exchange messages with each other.
In the following sections the functionalities of the groupchat is described by first representing newly added commands to the client library and then explaining the execution of the workflow.
The main goal here is to clarify the usage of the system for a client, who does not have the full knowledge of how a distributed database system works.

5.1 New Commands for the Group Chat

In addition to already implemented commands, such as "put" or "get" with the purpose of accessing the database or commands like "logLevel" for changing the level of the logs dynamically, seven more commands are implemented in order to realise the groupchat extension.
Following commands are possible during a chat session:

(1) PUT <key> <value>:
    Stores the given value and allows future access to it through the provided key.
(2) PUT <key>:
    Deletes the value assigned to the given key.
(3) GET<key>:
    Inserts the value assigned to the given key into the message.
(4) WSP <user1>,..,<userN> <msg>:
    Sends a whisper to the users provided by the client. It is possible for a chatroom to have up to 30 clients in it. The logic behind whispering feature is a client should be able to send messages during a chat session, only to the people who he/she wants to share it with.
(5) QUIT:
    Leaves the chat session.
(6) ACTIVE:
    Returns a list of all users in the chatroom. A client does not have to always keep up with the notifications about who joined the chat or left it, that is why the command "ACTIVE" eases for a client to see online members at that moment.
(7) HELP:
    Displays the help text.

Put and delete operations are fulfilled with the help of a chatbot. Most of the software systems include a chatbot in their system to work as a costumer service. Our intention by implementing a chatbot is to decrease the workload of the chatroom.

5.2 Chatbot

One of the extensions followed by the groupchat is a chatbot which is designed to help chatrooms to access the distributed database. The chatbot shares a lot of similarities with the client application. When a client wants to perform read or write operations while chatting, requests are directed to the chatbot.

5.3 Execution of the Workflow

To start off, in the same sense as Milestone 4, initially the External Configuration Service (ECS), where the storage servers are monitored and controlled, is executed. Following that, depending on client's decision, a number of servers are created. A client must connect to one of the servers by typing its IP address and port number in order to use the database system.

5.3.1 Unique Username. After successfully connecting to a key-value server, the client has to either enter a username or use the command QUIT to have a username randomly assigned to him. Usernames are implemented as globally unique identifiers for the clients, which prevents different clients from having the same username in different chatrooms. That way users are guaranteed to know who they are communicating with as long as their partner is connected to the system.

In order to avoid unnecessary extra connections, the ECS stores a list of users. Whenever a client tries to set its username, the request gets sent through the server to the ECS, which then checks if the username has been already taken by a client connected to any of the servers online. The end user either receives a welcome message with the username displayed if the operation was succesful, or an error message otherwise.

5.3.2 Chat Command. Moreover, to use the chatroom client needs to type "chat" on the console and choose a chatID for the chatroom and type it right next to the command "chat". A chatroom is created with the given id, subsequently client have two options: either enter a private room with a password feature or a public room.

5.3.3 Private/Public Chatrooms. If the client is the first person to create a private chatroom, then the right to give a password to the room belongs to the same person.

5.3.4 Password. Another client who is connected to the same server and wants to chat in the same chatroom can only access to the private room with the selected password by the client, who created the private chatroom. The password is hashed in order to provide safety for the client. If client wants to have a public chatroom, then a password is not needed.

To prevent heavy workload for a server, a chatroom has the maximum capacity of 30 people. The chatroom offers a communication platform for all the clients sharing the same chatroom. Every message sent by the clients have timestamps in order to keep on track with the flow of the messages for other clients. One message can contain maximum 200 characters.
When a client joins a chatroom, all the messages which sent until that time, will be visible to the latest joined client and all the members will be informed about who joined or left the chatroom.

5.3.5 Saved Messages. The messages are saved into a .txt file under the directory of the respective server.

Whenever a client wants to leave the chatroom, QUIT can be typed in order to use the functionalities of replicated and distributed database service from Milestone 4. To enable the chatroom service client only needs to enter "chat" command with the desired chatID.

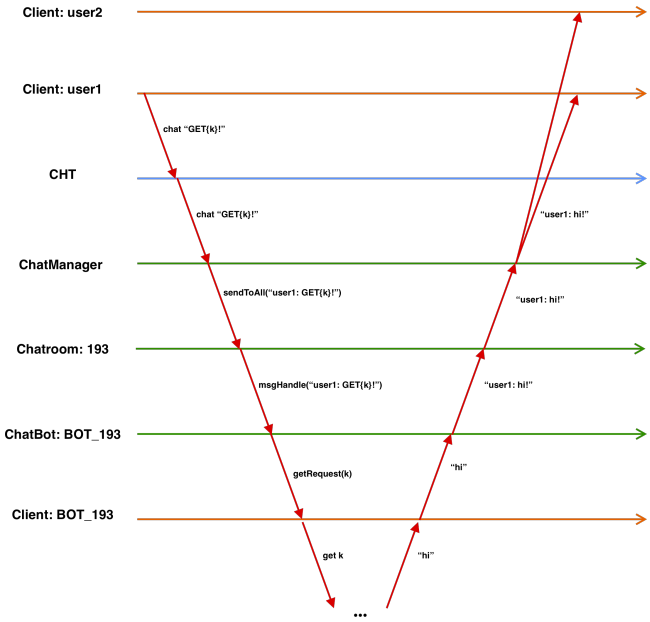

Figure 2: Client side architecture

6 IMPLEMENTATION

In this section the implementation of the chatroom is examined in depth. Firstly, the extension for the client side and then for the server side are described respectively. Lastly the chat system is explained in greater detail, since it encapsulates our work for Milestone 5.

6.1 Client Side Implementation

Figure 3: Client side architecture

The client side consists of the three following main components as seen in figure 3:

(1) *ClientApp* represents the client interface which allows input through the console. From there the client is able to issue commands to: connect to and disconnect from the system, interact with the key-value store and chat. The input is then checked and parsed before getting sent to the ClientLibrary. The result of the user command is displayed on the console through the ClientApp.
(2) *ClientLibrary* serves as a bridge between the client and the server.
(3) *ActiveConnection* abstracts the TCP socket connection to the server which allows the client to connect to the server socket and exchange data without worrying about the underlying structure.

6.2 Server Side Implementation

Each server owns a list containing the active chatrooms that it is responsible for. In order for a client to join one of those chatrooms is that it would first need to connect to that server, similar to the way storing key-value pair functions. The decision, to make chatrooms accessible only from one server has both its advantages and disadvantages. For one, it reduces the complexity of the system because otherwise servers would need a way to exchange updates regarding the active chatrooms and chat users whenever a user joins or leaves.

A problem with our implementation is that if the chatIDs are not equally distributed across all servers, which may occur due to the unpredictable nature of the hashing function, a single server could then be in charge of most chatrooms. This would cause that a server to be overloaded with requests and would lead to greater response times and, in the worst-case scenario, would result in a bottleneck for the whole system. In order to combat this issue, we limit the number of chatrooms belonging to one server to 15 and the number of users in a single chatroom to 30. This means a server is responsible for up to 450 chat users. These limits could also be easily changed depending on the intended use case of the system.

The biggest advantage of our decision is that it heavily reduces network traffic. Since all chatroom users are connected to the same server, that server can easily forward messages between them. Otherwise additional socket connections would have been required which would have both increased the load on the network and the overall complexity of the system.

Our idea for the chatting functionality was for it to be as lightweight as possible with clients entering and leaving chatrooms regularly.

6.3 Chat System

The chat system consists of three main components as namely:

(1) *ChatManager*, who is in charge of providing all chat functionalities to the client, including connecting to a chatroom and sending messages.
(2) *ChatRoom*, which is identified by a chatID. A Chatroom object contains a map of all connected users and their sockets, in order to be able to forward messages.
(3) *ChatBot*, which is responsible to execute PUT and GET commands during a chat session. The chatbot acts exactly like a client, meaning it first has to connect to the responsible server and then send its request.
By utilizing a chatbot, chat requests containing a key-value operation are slowed down, since only one "client" is performing those operations. However, if each chat user is tasked with executing the commands himself instead, then whenever a key is located at a server different from the server responsible for the chatroom, the client would need to reconnect twice. Not only that, but also all chat messages sent while the user was disconnected would either get lost or will be displayed after the user has reconnected, which would make the user temporarily unavailable. As mentioned in section 3.1, our system focuses on being available.

7 PERFORMANCE ANALYSIS

TODO

8 SUMMARY

To conclude our paper, we

REFERENCES

[1] [n.d.]. *"What Is a Key-Value Database?"*. "https://aws.amazon.com/tr/nosql/key-value/"

[2] Divyakant Agrawal, Amr El Abbadi, Sudipto Das, and Aaron J Elmore. 2011. Database scalability, elasticity, and autonomy in the cloud. In *International Conference on Database Systems for Advanced Applications*. Springer, 2–15.

[3] Eric Brewer. 2012. CAP twelve years later: How the "Rules" have changed. *Computer* 45, 2 (2012), 23–29.

[4] Eric A. Brewer. 2000. Towards Robust Distributed Systems (Abstract). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing* (Portland, Oregon, USA) *(PODC '00)*. Association for Computing Machinery, New York, NY, USA, 7. https://doi.org/10.1145/343477.343502

[5] Artem Chebotko, Andrey Kashlev, and Shiyong Lu. 2015. A big data modeling methodology for Apache Cassandra. In *2015 IEEE International Congress on Big Data*. IEEE, 238–245.

[6] Seth Gilbert and Nancy Lynch. 2002. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *Acm Sigact News* 33, 2 (2002), 51–59.

[7] Sultana Kalid, Ali Syed, Azeem Mohammad, and Malka N Halgamuge. 2017. Big-data NoSQL databases: A comparison and analysis of "Big-Table","DynamoDB", and "Cassandra". In *2017 IEEE 2nd International Conference on Big Data Analysis (ICBDA)(*. IEEE, 89–93.

[8] Mike Krieger. 2011. *Storing hundreds of millions of simple key-value pairs in Redis*. https://instagram-engineering.com/storing-hundreds-of-millions-of-simple-key-value-pairs-in-redis-1091ae80f74c

[9] Salvatore Sanfilippo. 2010. *Redis*. http://oldblog.antirez.com/post/redis-weekly-update-3-publish-submit.html

[10] Redis Development Team. 2020. *Pub/Sub Specification*. https://redis.io/topics/pubsub

[11] Yao Yu. 2014. *Scaling Redis at Twitter*. Youtube. https://www.youtube.com/watch?v=rP9EKvWt0zo