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, the key to retrieve substantial information is communication. Software that enables messaging features, such as calling and video transmission as in Skype, sending emails with Google Mail, chatting via WhatsApp or Facebook and many more, ease the exchange of information and are therefore crucial for productive working in groups of many people. They are used for managing, discussing, preparing, executing an organizing parts of a project. On this basis we dedicated us in Milestone 5 to implement a feature that enables multiple clients to communicate via chatrooms.

This report begins with an introduction part, where different commercial 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 are 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 globally unique chatID, assigned individually for each 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

One of the main criteria when developing a key-value database (DB), is how to handle large amounts of data while making the system available for multiple clients. DB systems, such as Amazon-Web Services, Microsoft Azure SQL or Oracle Database offer the combination of scalability and cost effectiveness.

Another important feature is elasticy. Elasticity is measured whether a system can be dynamically scaled-up by adding more nodes or down by removing them. The key-value DB BigTable and PNUTS are great examples[3].

There are also more key-value DB mentionable.

*Authors contributed equally to this work.

Amazon DynamoDB is a non-relational database system which provides single-digit millisecond latency[1], considering the system is used world wide. Data records are stored with a primary or composite key and several attributes depending on client's request[8]. 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[2] DB 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 [6]. 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[8]. The system focuses on high level of availability by scaling millions of read and write requests per second[6].

Though, our main inspiration was the DB Redis. It rose in popularity after its creation in 2009 and got deployed by many big companies such as Instagram [9] and Twitter [12], which require enormous DBs and fast responses to the millions of users they serve. One of its multiple features is the Pub/Sub system [11], added in March 2010 [10].

With it, clients were able to subscribe to a set of keys, namely a topic, and receive an update whenever their associated value got overwritten. In this model clients keep waiting for the server to send them an update. During this period, subscribers are limited to only subscription related operations, whereas the publishers are allowed to execute other commands normally.

In our model there is no distinction between the publisher and its subscribers. Everyone is equal. Also instead of a topic there is now a shared place where everyone is supposed to write on and receive from. This way every participant is publisher and subscriber at the same time. Apart from that being "subscribed" to it doesn't bring any limitations regarding the access to the rest of the DB.

Thus, the way we modelled it in the end, was having a many-to-many relationship of all subscribed clients, effectively, implementing a group chat functionality.

2 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.

2.1 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.

2.2 Persistence

2.3 Caching

write through policy

2.4 CAP Theorem

According to the CAP Theorem, distributed databases have three substantial properties to consider: consistency, availability and partition tolerance[4]. Eric Brewer, the man behind the CAP theorem, stated that a distributed database can fulfil at most two of three properties[5]:

- 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[7].
- 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.

Twelve years after from proposing the CAP theorem, Brewer mentioned that software engineers do not have to strictly abide to the 2 of 3 principle, it is rather a spectrum than a binary choice [4]. 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[4] 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[5].

2.5 ACID

ACID is a traditional design approach[4] 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. 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.

2.6 BASE

BASE, another design approach defined by Brewer, offers looser requirements than ACID[4]. 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[8]. 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 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 2.5, 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 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.

3 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.

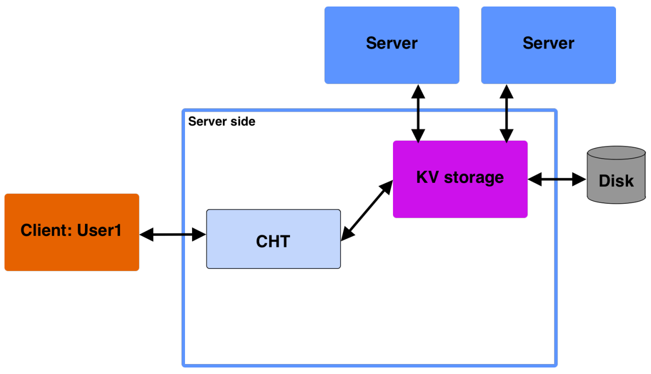


Figure 1: KV server architecture

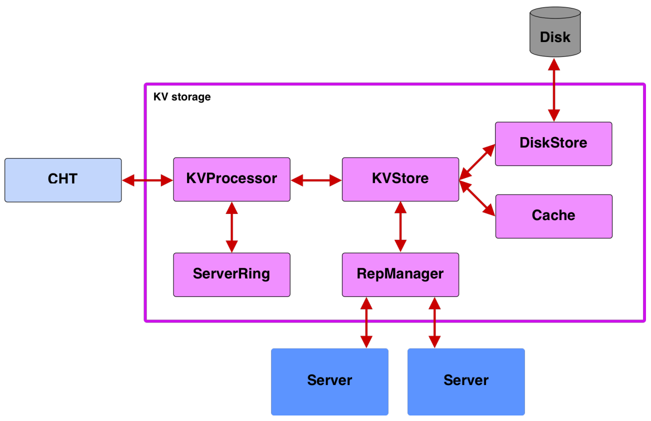


Figure 2: Client side architecture

–> Implementation moved here

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.

3.1 Client Side Implementation

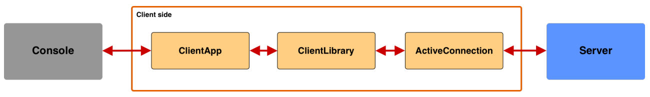


Figure 3: Client side architecture

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.

3.2 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.

server side: The client interacts directly only with the ConnectionHandleThread (CHT) created by the server to assist that specific client. At the start of the connection, the CHT contacts the ECS in order to assign the client a unique username. After that the CHT interacts exclusively with KV storage and the chat system, depending on the command issued by the client. The KV storage system provides the basic capabilities of a key-value store and persists data on disk. When more than two servers are online and replication is active, each server contains a replica of two other servers. Whenever a put operation is executed, it has to get forwarded to the server.

kv storage: The KVCommandProcessor (KVCP) is responsible to receive commands from the CHT and return the result. Before a value is stored or retrieved, the KVCP utilizes the ServerRing to check whether the server is responsible for the provided key or not. In the latter case, a simple error message is returned back to the client, which would then connect to the responsible server. In the former case, the request is sent over to the KVStore. If it was a GET command issued, the KVStore first tries to retrieve the value from the cache. In case the value has not been found, the DiskStore checks if it is stored on disk. With a PUT request, the KVStore sends the key-value pair to both the cache and DiskStore and also to the RepManager. This allows it to get sent over to the two replica servers. The RepManager is also charged with updating the replica on the server whenever one of the coordinators processes a PUT command.

4 GROUP CHAT

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 group chat 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.

4.1 Basic Functionalities

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.

4.1.1 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.

4.1.2 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.

4.1.3 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.

4.1.4 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.

4.1.5 Saved Messages. The messages are saved into a .txt file under the directory of the respective server. We provided this feautere, in case a network partition occurs in the system and messages get lost.

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.

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 group chat 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 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 check the members of the chatroom at any time.
(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.

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 2.4, our system focuses on being available.

4.2 Database Access

The ChatBot

Reason why we need him During the development we reached one significant problem. While being connected to a chatroom a client is making use of its only socket. That means he cannot access the database (DB) at the same time.

There are different solutions for that.

1. More sockets The easiest way would be increasing the number of sockets a client uses by one. This way the client has one socket for the database operations (DOPs), whereas the second one is responsible for the connection to the chatroom.

We decided against it, because that would double the amount of TCP connections per client in the network. In our use case when a client is connected to a chatroom he isn't heavily working with the DB and focuses more on communication. Only a few and small DOPs are necessary while chatting.

2. Quick Dis- and reconnection The client could quickly leave the chatroom and do the DOPs. After that, he quickly rejoins the chatroom.

We decided against it, because then there might be a chance to miss messages. In our use case seeing every message while connected is a crucial aspect.

3. ChatBot In this approach every chatroom of the server gets a ChatBot object assigned to it that executes the DOPs written in the chat messages. Then before writing the chat messages the DOPs will be replaced by the corresponding response of the DB.

We prefer this solution, because in our use case we expect chatrooms to receive mostly normal messages and not DOPs.

Functionality There is always one ChatBot object per created chatroom. The ChatBot acts mostly similar to a client by connecting to one of the servers of the DB. Since every server is part of the DB it will firstly connect to the server that created it.

When a normal message and/or DOPs is received, the chatroom object filters the command and calls the ChatBot to execute it. The ChatBot will call every put, get and delete request likewise to client. The chatroom object will then replace the DOP text with the reply of the DB. Only after this translation the chatroom object will send the now modified message to every participant in the chatroom.

Bottleneck of the chatroom Thinking about the implementation the ChatBot is obviously a bottleneck. Imagine there is a full chat room with 30 users. If they now start doing their own DOPs (for instance updating their weekly report) all operations are executed by one ChatBot. The workload of 30 sockets are pressed into one. This leads to a significant efficiency loss.

As already mentioned, it isn't that big of an issue because that won't happen in our use case. Heavy modification and updates of the DB must be done outside of chat rooms, to make full use of the clients individual socket.

chat system: All chat related requests get transferred by the CHT to the ChatManager. This has a list of Chatroom objects. When a client tries to connect to a chatroom, the ChatManager checks the list whether a chatroom with the provided chatID already exists or not. In the former case, if the chatroom is public, the client is granted access or, in case of a private chatroom, requested to enter a password. In the event of no chatroom found with the given chatID, the client is allowed to create the chatroom by specifying its type and providing a password, if a private chatroom is chosen. In order to increase security, the password gets hashed on the client side and only the hash of the password is stored at the server side and used for validation.

5 PERFORMANCE ANALYSIS

After we were done with the implementation of MS5, we wanted to test our system to 1) analyze the response times of normal read and write operations, and 2) check the impact of the chatting functionality on the response times.

320 key-value pairs are used as input for the tests, which were derived from the Enron email set. The way the test works is that clients execute `PUT`s on all keys, then they try obtaining them back

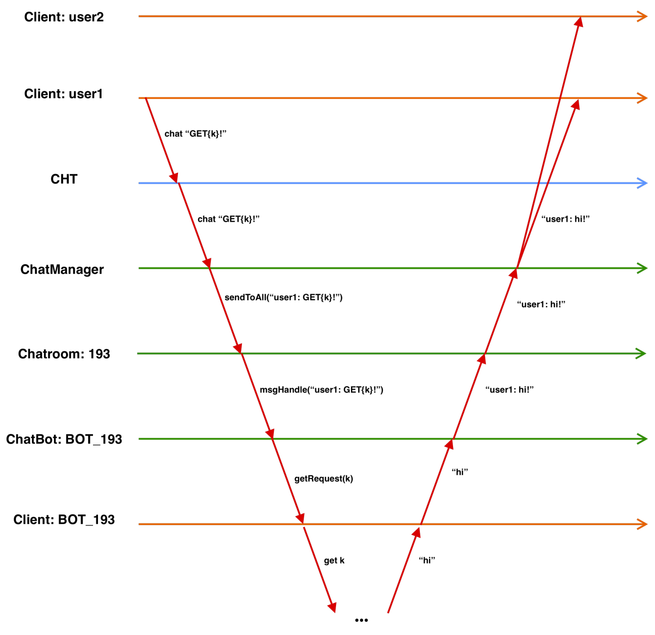


Figure 4: Client side architecture

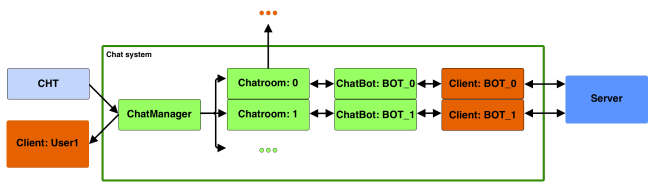


Figure 5: Client side architecture

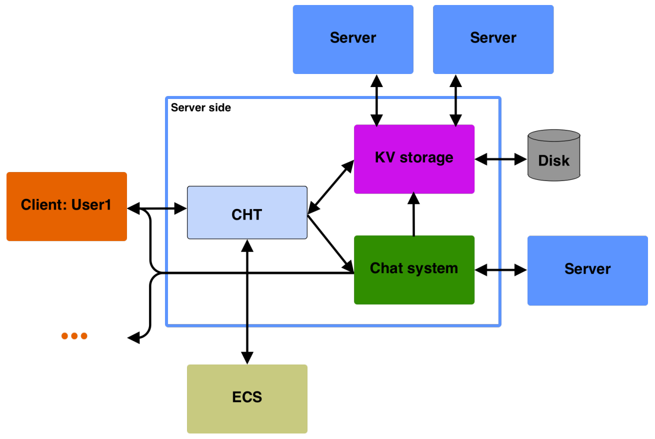
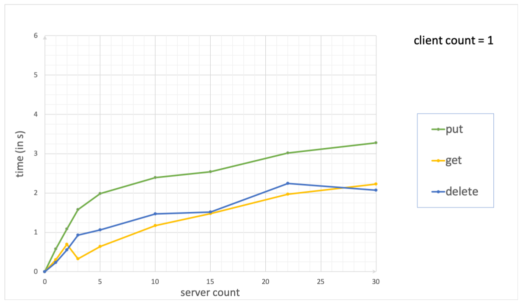
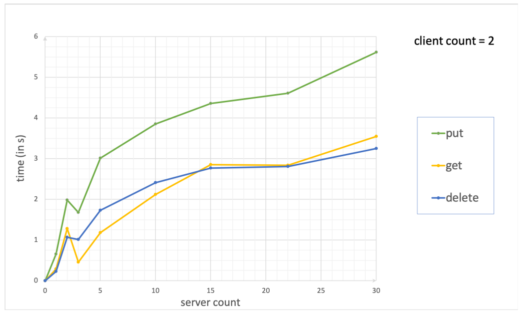


Figure 6: Client side architecture



(a)



(b)

Figure 7: Key-value servers performance

with GETs and in the end, all keys get deleted. Each of the three different operations got timed seperatly, so that their efficiency could be compared. The cache replacement strategy in place was Least Recently Used (LRU), which is the default strategy, and the cache size was set to 16, which represents 5% of the storage size.

Before a new set of commands got started, the list of keys got shuffled, which causes both the degree of cache utilization during read operations and the chance of reconnecting to the responsible server to be based on luck. In order to avoid skewed results, each data point is calculated by taking the average of five test runs.

In figure 7 we vary the amount of key-value servers and measure the time 320 operations took. At first glance, it is clear that adding an extra client in 7b leads to lower response times, since double the amount of requests need to get processed by the servers. Otherwise, in both situations PUT operations take the most time, because each time the disk has to be accessed and a file has to be written. Even though deletetion manipulates data on disk as well, files just have to be located and removed from the directory, with no need to open them. GETs usually took the least amount of time, since disk access is not always necessary. A significant drop of read response time can be noticed at the three server mark due to replication just getting started. At that point, no reconnection is required for the read commands, owing to the fact that all three servers are now responsible for read operations on all keys. The chance of a reconnection being needed is equal to $1 - \frac{3}{N}$ with N being the amount of key-value servers running. As apparent by the formula, the chance decreases as the server count increases, leading to the rise seen later on in the graphs.

After we have evaluated the response time for the normal database operations, we try executing these same commands but inside a chatroom. For our testing purposes, we had two clients in the same chatroom perform those commands. The difference here is that, unlike in 7b where each client directly contacts the key-value servers, only one common bot is utilized by both users. After each

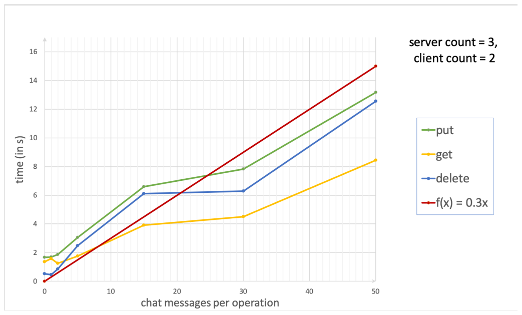


Figure 8: Key-value servers performance in chat mode

command, the user would send a certain number of chat messages as depicted on the X-axis, in order to simulate a general use case scenario of our system. Graph 8 depicts the increase in response time in relation to the chat messages sent. When few messages are sent, the extra time required compared to the normal database operations seems unnecessary. However, the response times do not grow at the same ratio as the chat activity, but rather roughly follow the path of the function $0.3x$. That means that for each additional chat message per operation per client (equal to overall $320 * 2 = 640$ extra messages), the system is slower by just about 300 milliseconds. In one extra second, more than 2100 messages can get exchanged.

6 SUMMARY

To conclude our paper, with the group chat extension in our system we provided clients a chat system which supports low latency and high availability. We got inspired by Redis Pub/Sub strategy and adjusted it in our system by bringing the clients on the same level instead of having publishers and subscribers. As we mentioned in our motivation **??** subscribers are waiting on an update of the subscribed key and during this process clients do not use any other functionalities in the system. To prevent the idle time which client just waits a response from the server, we enhanced the system maintaining many-to-many relationship in contrast Redis' one-to-many relationship.

During the development process, initially we considered the demands of the client, for instance having globally unique usernames, accessing the database while chatting and having private and public chatrooms. Subsequently, with the implementation of the chat system we fulfilled the potential demands and extent it with features such as "ACTIVE" or "WSP" commands as explained in the section Newly added commands for the group chat **??**.

Consequently, after the performance tests we recommend our system for light-weighted communication. The reason for that is as explained in the Server side implementation 3.2 in greater detail, it might occur that the chatrooms are not equally distributed in the system and thereby just one server might have to carry the load of having all the chatrooms. And this leads us one of our so to say weaknesses in the system which is the limitations. We limited the number of chatrooms to 15 for one server and the number of users using one chatroom to 30. The numbers could be arranged depending on the use case of the system. If the user of our group chat prefers to work on a large-scaled messaging platform and can give up on having low latency, the limitations for the number of chatrooms and the clients could be increased.

One of our strengths is allowing the access to the database while clients are chatting.

REFERENCES

[1] [n.d.]. *"What Is a Key-Value Database?"*. "https://aws.amazon.com/tr/nosql/key-value/"
[2] Daniel Abadi. 2012. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *Computer* 45, 2 (2012), 37–42.
[3] 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.
[4] Eric Brewer. 2012. CAP twelve years later: How the "Rules" have changed. *Computer* 45, 2 (2012), 23–29.
[5] 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
[6] 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.
[7] 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.
[8] 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.
[9] 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
[10] Salvatore Sanfilippo. 2010. *Redis.* http://oldblog.antirez.com/post/redis-weekly-update-3-publish-submit.html
[11] Redis Development Team. 2020. *Pub/Sub Specification.* https://redis.io/topics/pubsub
[12] Yao Yu. 2014. *Scaling Redis at Twitter.* Youtube. https://www.youtube.com/watch?v=rP9EKvWt0zo