Lab Course - Cloud Data Bases

Group 1

Oleksii Kulikov
Technische Universität München
Department of Computer Science
Munich, Germany
alex.kulikov@tum.de

Andreas Amler
Technische Universität München
Department of Computer Science
Munich, Germany
amler.andreas91@gmail.com

Tim Bierth
Technische Universität München
Department of Computer Science
Munich, Germany
bierth@gmx.net

ABSTRACT

The following report focuses on the key aspects of our group's cloud database system. It includes the motivation to use a database system of the given format, related work aspects, our approach to design decisions throughout the project including the final extension, as well as an evaluation of the database performance.

We understand an individual design decision as an approach not precisely defined in the specification of any of the given milestones. This includes the implementation of data management, the approach to communication in general, the message format, the failure detection mechanism and the general idea of the extension. The performance of the final product is presented and evaluated in consideration of a specific use case.

In the end of the report, a conclusion or rather an incentive for further development is given.

KEYWORDS

Cloud databases, Data storage, Message formats, Failure Detection, Error management, Authentication, Subscription.

ACM Reference Format:
Oleksii Kulikov, Andreas Amler, and Tim Bierth. 2018. Lab Course - Cloud Data Bases: Group 1. In *Proceedings of February 2018 (Cloud Databases).* Munich, Germany.

1 INTRODUCTION

In the scope of the lab course Cloud Databases offered by the Technische Universität München our team received the task to implement a cloud data storage based on the key-value store concept.

Due to key-value stores being lightweight and fast they have been becoming increasingly popular in the recent time. While traditional relational databases can offer high functionality with a wide range of table operations available, they are more complex to implement and require significantly more resources.

In comparison, key-value stores do not require a predefined schema and hence the data does not necessarily need to be organised in a particular way. In a key-value store, values can be easily accessed by looking for the corresponding key. The values can hold different data formats, ranging from simple integers or strings to complex objects like video files for instance. Due to the vast simplicity and yet high flexibility of key-value stores, the specified database can be implemented in full accord with the specific application case.

Cloud Databases, Munich, Germany,
2018. ACM ISBN 123-4567-24-567/08/06…$15.00
https://doi.org/10.475/123_4

With regard to the given specification a general use case of the system can be formulated. We eye a company willing to deploy a database system with a high number of servers to manage vast amounts of data. Furthermore, the service offered should be well established for effortless scaling of data up and down as well as for different numbers of requests. The need for high scalability and flexibility hints at global players in the databases segment with a large customer base.

More information on state of the art usage of cloud based key-value stores is given in the following brief overview of related literature.

2 RELATED WORK

In the modern IT world small and usually local user applications are being continuously replaced by large-scale online services, which Dr. Anu Gupta et al. from the Panjab University, Chandigarh call ”one of the biggest changes in IT after the rise of World Wide Web" [1, p.1]. New terms like Data as a Service (DaaS) and Database as a Service (DBaaS) emerged. Such services are usually being offered in form of a Cloud.

While cloud databases are commonly used, they do not come without difficulties for the developer. Some of the key challenges to achieve are full availability, consistency, scalability, security, fault tolerance and privacy, to mention just a few value factors of a good distributed database [1, p.3]. Even though relational databases usually offer higher functionality than NoSQL data stores, the latter do offer a very solid basis for implementing all of the requirements above.

With our key-value store implementation we tried to point out that scalability, fault tolerance and security in specific are easily implementable in the scope of a university lab course.

We found the paper "A Modeling methodology for NoSQL Key-Value databases" [3, p.2] helpful in developing a more specific use case for our database system. It introduces key-value databases as a new approach to particular scenarios, where they outperform their relational counterparts. According to the paper, key-value stores are the model of choice for rapid prototyping, as not much attention needs to be paid to the way the data is stored and to the modus operandi of the database [3, p.2].

Later on, the deployment of key-value stores is discussed in the context of online multiplayer games, where the database needs to store all kinds of information about online matches and player performance [3, p.5].

Another article states that key-value databases perform particularly well when handling fluctuating traffic and great latency. This can be the case in online gaming, where extremely large amounts

of data need to be collected at peak gaming times [2]. The data gathered can for instance consist of subscriber entries in a streaming channel for gaming contents. Another gaming specific application case would be users entering a multiplayer game, who need to be temporarily stored for the duration of the gaming session. We consider these the cases that our database would excel at.

3 APPROACH TO DATA STORAGE

Our data storage is comprised of a cache that can hold a set amount of KV-tuples in main memory, in addition to a persistence system that is used to store data on disk.

3.1 Cache

In this section, we describe and analyze the data structures used to implement the cache of our database system. For our analysis, we consider the three operations *lookup*, *insert* and *replace* (i.e. replacing a KV-tuple in the cache according to the chosen displacement strategy). We implemented the strategies Least Frequently Used (LFU), First In First Out (FIFO) and Least Recently Used (LRU).

When FIFO is chosen as displacement strategy, the cache stores KV-tuples in a queue. This guarantees the ability to add and replace elements in a first-in-first-out manner in constant time. In order to provide efficient lookup, we use a hash map that indexes the queue. Consequently, when looking up a key, the map can be queried to retrieve a reference to the corresponding tuple in constant time.

To implement LRU, tuples are stored in a linked list. Whenever a tuple is referenced, it is moved to the front of the list, whereas tuples at the back of the list are first to be replaced. This guarantees that the tuple being replaced is the one that was least recently used. Analogously to FIFO, we use a hash map to provide fast lookup.

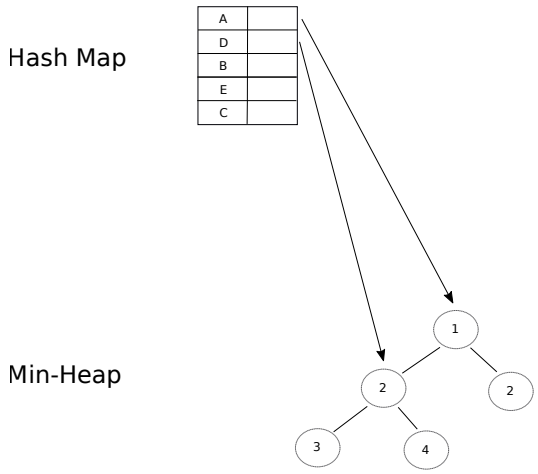

Figure 1: Illustration of LFU cache storage.

Finally, when LFU is chosen as displacement strategy, KV-tuples are stored in a binary min-heap. For this purpose, tuples are internally assigned a priority that initializes at the value 1 and is increased each time the tuple is referenced. Therefore, in order to find the least frequently used tuple, we can simply inspect the root of the binary tree that represents our heap. Analogously, when replacing the least frequently used tuple, the tuple stored in the root can simply be overwritten, while resetting the priority of the root to 1. Inserting new tuples into the LFU cache requires time asymptotic to $O(\log n)$, since a bubble-up operation has to be performed to propagate the new entry upwards within the tree to reach the correct position. In order to provide fast lookup we once more use a hash map indexing the binary heap. Figure 1 illustrates the structure of the LFU cache, including the hash map and pointers to exemplify the indexing. Finally, Table 1 summarizes the performance of our cache.

Table 1: Summary of cache performance by displacement strategy and operation

	lookup	insert	replace
FIFO	$O(1)$	$O(1)$	$O(1)$
LRU	$O(1)$	$O(1)$	$O(1)$
LFU	$O(1)$	$O(\log n)$	$O(1)$

3.2 Disk Storage

Our database system uses a heap file to store KV-tuples on disk, i.e. a file where new tuples are simply inserted at the back. This enables fast insertion speed, but would ordinarily lead to slow lookups, since the whole file has to be scanned. For this purpose, we additionally maintain an index file that stores a copy of all keys in the storage, with the distinction that keys are stored in an ordered manner. Additionally, keys in the index file are padded to a fixed length of 20 byte. These properties allow us to perform a binary search on the file, locating any key in time $O(\log n)$, with n being the total amount of keys in the storage. Every key additionally has an 8-bit integer appended to it, indicating the position in the storage file that stores the corresponding value. Inserting into the index file requires up to n keys to be shifted, since the ordering has to be maintained. However, this is comparatively cheap, since keys are relatively small compared to values, which can encompass up to 120KB. Therefore, shifting within the index file is preferable to performing the same operation in the storage file.

When deleting a key, it is physically removed from the index file, which once again requires shifting of up to $n - 1$ keys. The corresponding KV-tuple within the storage file, however, is not immediately deleted physically. Instead, we set a bit that marks the tuple as obsolete, causing it to be ignored by all lookup requests. Naturally, after a high amount of delete operations, the storage file will be bloated with unused tuples. For this purpose, we implemented the *vacuum* operation that scans the storage, physically removes all obsolete tuples, and consolidates space. Given a sufficiently large storage file, vacuum is an expensive procedure and should therefore be used sparingly. In our current implementation, it is used after data has been moved from the server to another one. This presents a fitting scenario, since a large amount of tuples has been removed, ensuring that there is space for vacuum to consolidate. Additionally, the server is write locked at this point, ensuring that the storage is not altered simultaneously.

4 MESSAGE AND READER FORMAT

There exist two different types of messages. One is called "KV-Message" and the other is called "Admin-Message". The KV-Message is used for client-server communication and whenever keys and values are involved. The AdminMessage is solely used for communication between the ECS and the kv-servers. The status byte of each message type operates on a predefined bit range. However, as the project progressed, the bit range of the KV-Message had to be extended and now additionally includes Bit 61 and Bit 62.

The implementation focuses on sending as little information as possible, while maintaining a usable message protocol.

4.1 KVMessage Functionality

There are three types of KVMessages:

Type 01: - status
This type applies to the following status flags:
SERVER_STOPPED
SERVER_WRITE_LOCK
AUTH_SUCCESS
AUTH_ERROR
SUB_SUCCESS
SUB_ERROR.

Type 02: - status | key_length | key | -
This type applies to the following status flags:
GET
GET_ERROR
DELETE
DELETE_SUCCESS
DELETE_ERROR
UNSUB.

Type 03: - status | key_length | key | value_length | value
This type applies to the following status flags:
GET_SUCCESS
PUT
PUT_SUCCESS
PUT_UPDATE
PUT_ERROR
FAILED
NOT_RESPONSIBLE
AUTH
SUB.

The status and the key_length consist of one byte only, as they are predefined. The key itself, although limited in size, may vary from 1 to 20 bytes. Value_length allows the length of a four byte integer, followed by the value.

The bit range of KVMessage is: 1 - 20, +61, +62.

4.2 AdminMessage Functionality

There are two types of AdminMessages:

Type 01: - status
This type applies to the following status flags:
START
STOP
SHUT_DOWN
LOCK_WRITE
PING
CRASH
RECEIVED_AND_EXECUTED
AN_ERROR_OCCURED.

Type 02: - status | length_b1 | length_b2 | length_b3 | length_b4 | payload
This type applies to the following status flags:
META_DATA
MOVE_DATA
SERVER_DOWN.

The status consists of one byte only, which may be followed by a four byte integer and the payload. They payload can be anything. This makes the AdminMessage applicable to a wider range of purposes, but limits the ability to check for validity. Therefore this message is used when the user is not involved, as human error can be excluded.

The bit range of AdminMessage is: 21 - 40, 41 - 60 (confirmation message merge).

Originally we distinguished between admin messages formulating commands to a server and messages formulating responses. However, as the structure of both formats was similar, the only relict of this temporary design decision is the distinguishment between a bit range responsible for commands and a bit range responsible for confirmation or rather feedback.

5 APPROACH TO PORTS

We have divided the flow of communication according to the communication partners involved. All port ranges are calculated as an offset from the first port in the server file, as can be seen in Figure 2. In practice, this has the advantage of being able to process different flows of communication with different network cards. This also enables us to delegate the processing of different flows of communication to third party providers.

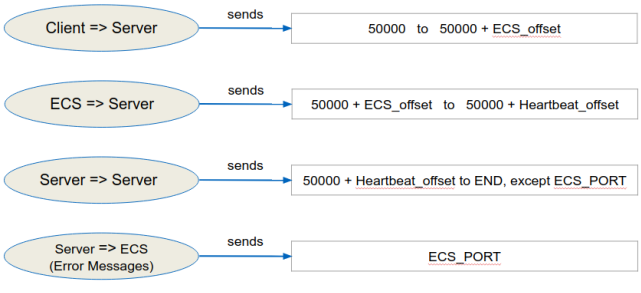

Figure 2: Ports usage.

6 FAILURE DETECTION

In the scenario of one or more servers failing to maintain the service, the failure first has to be detected and then repaired, so that

the initial infrastructure of the database is restored. As the initial detection has to happen on the server side and the restoration on the ECS side of the system, we view the two components as separated in the following.

6.1 Failure Detection on Server Side

The failure detection on server side consists of two major components: detection on replication and a concept called Heart Beating. We start with the former.

When a client sends a *put* request to one of the servers, the server is supposed to replicate the request to its successors in order to maintain database consistency. In the case that the received reply from the successor was correct, nothing else needs to be done. If, however, the connection to the successor can not be established, or no/incorrect reply is received, then an error message is being sent to the ECS to inform it about the failed successor. From here on, the ECS is responsible for handling the failure and nothing else needs to be done on the server side. The process is visualized in Figure 3.

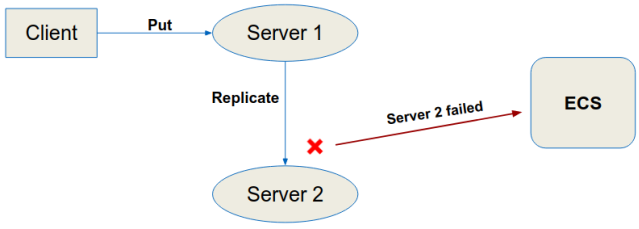

Figure 3: Detection on Replication.

Heart Beating, on the contrary, does not get executed on request, but periodically instead. Each of the servers in the cloud database has the task to regularly check whether its successors are still intact. For this purpose, every server periodically sends *ping* messages to both of its successors, as can be seen in Figure 4. We set the frequency to $f_0 = \frac{1}{5}\frac{1}{sec}$ in our implementation. In a real world application it would be sensible to adjust the pinging frequency according to the specific use case of the database.

After the *ping* message was sent the further proceedings are similar to the replication case. If the connection to one of the successors cannot be established, or the reply format is wrong, then an error message indicating the failure is sent to the ECS.

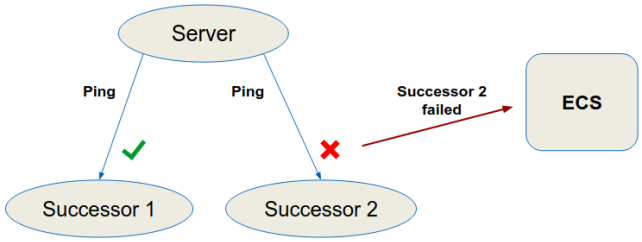

Figure 4: Heartbeating concept implementation.

Our database implementation operates with three major Java classes responsible for the concept of Heart Beating: Romeo.java, PingListener.java and Julia.java.

The interaction between the classes is the following: Romeo.java is a thread which periodically sends *ping* messages to the current server's successors and reacts to the received reply. PingListener.java is a thread responsible for a server socket that is open for server-to-server requests. Whenever the PingListener receives an incoming connection from another server, it delegates the connection to a new Runnable instance of Julia.java. Julia.java receives and processes the request. If the request happens to be a *ping* message, Julia sends back a reply informing the sender that the successor is intact. Figure 5 illustrates the communication between Romeo and Julia.

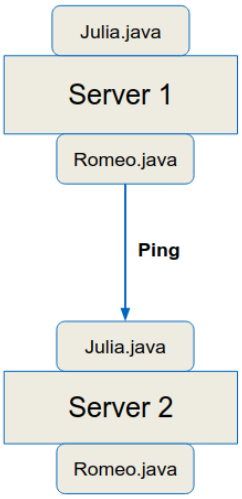

Figure 5: Ping sender and ping listener concept.

6.2 Error Processing on ECS Side

All servers are sending their error messages to the ECS whenever they detected a crashed server. These messages are processed on one port only, by collecting them in a buffer and then concurrently processing this buffer. If there is at least one entry in the error buffer (ErrorBuffer.java), it is taken out and it is being checked, if the crashed server was already registered as crashed. If it was registered, nothing needs to be done. If this information is new to the ECS, the proper reconciliation operation is initiated. Either way, at the end the original message and all identical message are deleted from the buffer. Figure 6 illustrates the described proceedings.

7 EXTENSION

In the scope of the fifth milestone we extended our cloud database by two additional components: Authentication and Subscription Service.

7.1 Authentication Service

A database where every client has full access to the entire functionality can be useful in a context of a closed system without any interaction with the outside. It is, however, utterly useless in a public environment if every client has unlimited access to its contents. In order to fulfill common security requirements, only registered

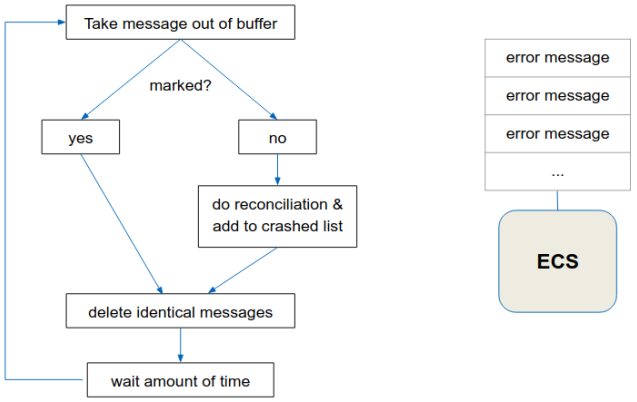

Figure 6: Illustration of error buffer usage in form of a timeline.

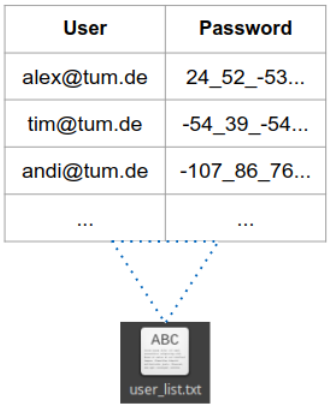

Figure 8: user_list.txt file sample.

and verified clients are allowed to use the database functionality. Hence, the concept of authentication needs to be implemented.

As it is inconvenient for the client to authenticate for each single request to the database (e.g. connect, put, get, subscribe, etc), our client application only requires to do so once, when first executing the *connect* command. The user enters their credentials in the form *connect <email address> <password>*. After successful validation on the server side, the authentication is valid for one entire session - until the client manually disconnects from the database.

The credentials validation is performed on the server which initially received the connection request, as shown in Figure 7. Each server has access to an internal configuration file containing a list of all registered users' email addresses and passwords. If the authentication data sent by the client corresponds to an entry in the user list, the server sends an *authentication successful* message to the client. After this, the client gains full access to the database functionality for one session.

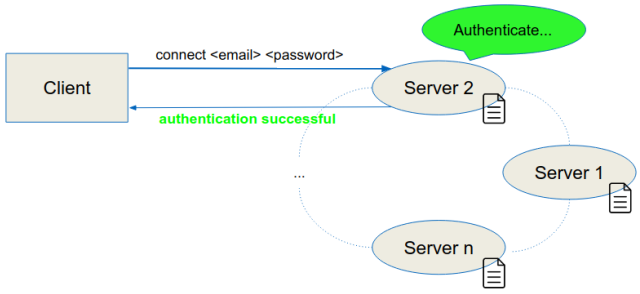

Figure 7: Authentication process.

To ensure full scale security of login data, the user list is available to servers only. Neither the client, nor any third party entity has access to the user data. Additionally, each user password is not being saved in cleartext, as can be seen in Figure 8, but instead in form of an integer interpretation of a hash value generated from the original password.

As a database administrator, there is no need to manually generate first the hash values and then the integer interpretations. Our team created a Java based tool for this purpose, which accepts a user password in cleartext as input, internally calculates its hash value, and delivers the corresponding integer interpretation as output. If needed in a specific use case, the tool can be integrated into the database to automatically register new users to the service. For a usage sample confer to Figure 9.

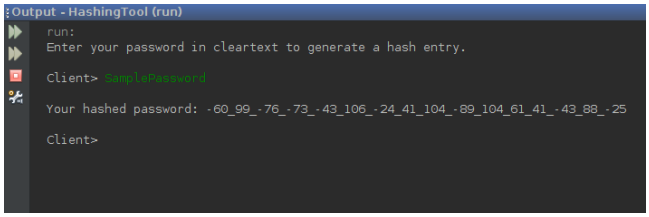

Figure 9: Sample hashing tool usage.

7.2 Subscription Service

The user is able to subscribe to a key with a subscription command. As the user has to login to use the services of the cloud data base, any server client interaction is preceeded by authentication. One may call this a session. If a session exists, the server involved knows about the e-mail of the user initiating an interaction. As the user executes a subscription and thus sends the server the key it wants to subscribe to, the server notes the subscription in a separate data structure. This data structure is basically a hash map, where each entry consists of an e-mail address and a key. The process is visualized in Figure 10.

With an unsubscription the user lets the server know, that he does not want to receive information about the denoted key anymore. It is then deleted from the hash map. A subscription is handled as a write operation. Therefore the server, which holds the key value tuple in its database is also responsible for over seeing subscriptions to this key. In case the topology of the server ring is changing, the data structure containing the subscriptions is passed on to the server, which now handles write requests to those keys.

If a client performs a write operation, the server checks, if the key is subscribed to and then sends an e-mail to every subscriber of this key, informing him of the update.

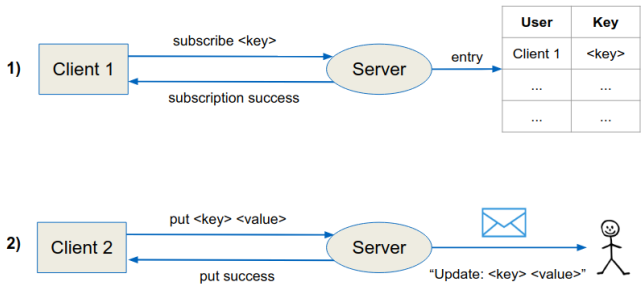

Figure 10: Subscription process.

The e-mail service implementation currently uses an interface provided by google to log into an existing account and send the instructed mails. In practice one would either deploy an smtp server of its own or use a commercial interface provided by a third party. E.g.: Amazon employs a service called "Amazon Simple E-Mail Service", which specializes in sending large amounts of e-mails to a vast amount of recipients without expecting a reply, but just to inform those addressed by the organization, which booked the service. We recommend using a third party solution, as this task is not database related.

8 DATABASE PERFORMANCE

The main metric of database performance used in the project is tuples per second, which enables a range of performance measurements based on the throughput of executed requests in a time frame.

8.1 Scaling Number of Clients

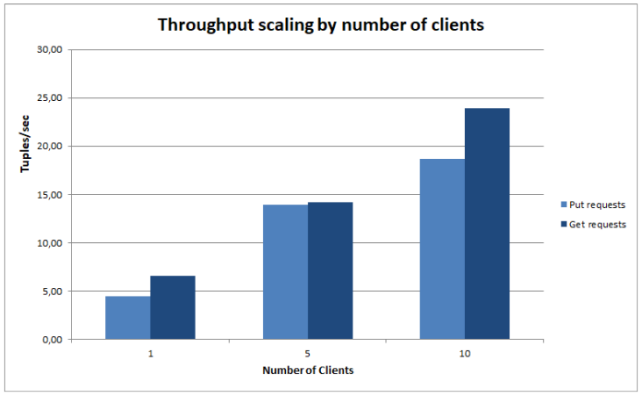

Figure 11: Throughput scaling by number of clients.

The first chart (Fig. 11) features the throughput of *put* and *get* requests based on the number of clients running. The figures assume a cluster of five servers constantly running on the back-end. The whole database system has to process a constant number of requests in each constellation. The constant number of requests is implemented by generating one set of random keys and distributing them evenly to the clients involved in the running scenario. With great numbers of keys, one can assume, that the hash converges against a homogeneous distribution among the servers as well.

We notice strong throughput increase with the number of clients scaling from one to ten. This result derives from each of the clients running in a separate thread, which enables all of the clients to send their requests simultaneously. Hence, the throughput of multiple servers adds up to the overall throughput figure.

When comparing the execution time of *put* and *get* requests to the database, *get* requests generally tend to be faster. There are two reasons for this development: Whenever a server receives a *put* request, it has to replicate it to both of its successors. This process takes a significant amount of time, as a new connection to each of the successors has to be established first. In the case of *get* requests, however, no changes to the database content are required. The database can generally process several *get* (read) requests at the same time. This is not the case for *put* (write) requests.

8.2 Scaling Number of Servers

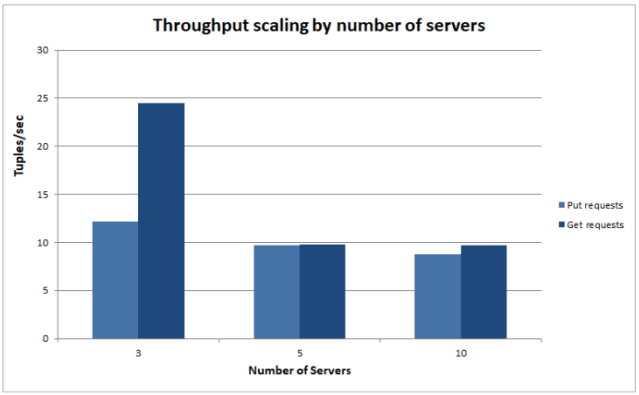

Figure 12: Throughput scaling by number of servers.

The second chart (Fig. 12) features the throughput of *put* and *get* requests based on the number of servers running. The figures assume five clients issuing the same number of requests to the servers cluster.

We observe significant throughput decrease when scaling up the cluster from three servers initially to five servers and a less significant throughput decrease when scaling up from five to ten servers.

One cause for this development is that the more servers are in the cluster, the more frequently each of the clients has to reconnect to a different server when trying to send a new *put* or *get* request. This happens due to the restriction that only the servers which are responsible for a specific range of keys are allowed to accept incoming requests to these keys.

Furthermore, a *put*request distinguishes from a *get* request by the necessity of sharing the same resource, being the allocated memory where the key-value tuples are stored. Simply put, several

processes have the ability to read, but only one write operation can be processed at the same time.

As previously observed in chart one (Figure 11), *get* requests tend to be executed faster for the reasons already mentioned above. In addition to that, when the cluster consists of three servers only, all of the servers contain the exact same data entries due to replication. Hence, the clients never have to reconnect to a different server when sending a *get* request, because all three servers are responsible for all read requests. This does not apply to *put* requests though, as there is always only one server responsible for writing a certain range of key-value entries to the database.

9 CONCLUSIONS

9.1 Database Evaluation

The evaluation results represent a typical performance scheme for lightweight key-value databases with *get* requests generally being faster and more efficient than *put* requests. Looking back at the previously introduced use cases, we figure that the implemented cloud database is indeed optimized for reading-heavy applications with a preferably small number of write-operations performed.

While there is a significant performance leap when extending the system from three to five servers, the throughput generally scales quite well with a growing number of servers from five servers and up. This enables the database administrator to significantly increase the database size whenever needed. Besides, when implementing a suitable test environment for our cloud system, we figured that the number of servers running can be very conveniently changed both programmatically and visually by using the ECS client console. This easy scalability is particularly useful when dealing with an ever growing user base or dataset in regard to modern cloud based Internet platforms.

Referring to the 'Related Work' section of the report, the cloud system implemented by our group appears to support and to verify the common understanding that databases which use the key-value store concept are generally very flexible. This attribute applies to both the database implementation - with regard to different programming approaches available at different development stages - as well as to the highly convenient scalability of the storage system.

Additionally, as presented our database provides all the necessary equipment to apply it to some sort of e-sport service. Where data about matches is constantly being collected and one may subscribe to a certain match or even player. If new data is available on the outcome of a match or the results of a player, the subscriber is then notified by e-mail.

9.2 Future Work

Despite the implemented database being a well-established and ready-to-use product, a range of valuable improvements to the system can be defined.

With the main functionality being implemented, it appears sensible to make the database more convenient for standard usage. At this point, a proper graphical user interface comes to mind. The improvement seems useful for both the client and the administrator (ECS) side of the system. A personal settings file on the user side could make sure the client always connects to their preferred main server with the first *connect* request being executed automatically at application startup.

From the database administrator point of view, a different graphical interface seems useful, which would view the currently running servers and their parameters. The latter could be for instance the number of clients currently connected, the status of the server (running, failed, stopped, locked for write requests, etc.), or the list of current user subscriptions.

When viewing the back-end extension possibilities, some stricter consistency models come to mind, such as data-centric or client-centric consistency for instance. While at the current development stage the implemented database functions properly, some rarely occurring system failures and data mismatches are not unthinkable of.

All in all, in order to deliver a "bulletproof" end product, further development of the presented database might be useful. Nevertheless, except for the possible improvements mentioned above, our group has managed to implement a fully-operational cloud based key-value storage offering high flexibility and functionality. The latter ranges from a user-friendly client application to complex back-end concepts like replication, failure detection and subsequent reconciliation mechanisms.

REFERENCES

[1] "Indu Arora and Dr. Anu Gupta". "2012". "Cloud Databases: A Paradigm Shift in Databases". *"International Journal of Computer Science Issues"* "9", "4" (jul "2012"), "1–4".

[2] April Reeve. [n. d.]. Big Data Architectures - NoSQL Use Cases for Key Value Databases. ([n. d.]). https://infocus.dellemc.com/april_reeve/big-data-architectures-nosql-use-cases-for-key-value-databases/

[3] Gerardo Rossel and Andrea Manna. [n. d.]. A Modeling methodology for NoSQL Key-Value databases. ([n. d.]), 2–5.