Cloud Databases Report

MOHAMED HOUSSEIN ZAGHDANE, Technichal University Of Munich
SANSKAR GUPTA, Technichal University Of Munich
EMIR CHALGHAF, Technichal University Of Munich

CCS Concepts: • **Distributed key-value store**; • **Replication**; • **Consistency**; • **Availability**; • **Scalability**;

1 INTRODUCTION

With data becoming an essential aspect of our day-to-day lives, distributed databases reside at the heart of every organization's data infrastructure. The end-users who interact with a web service or a mobile application might not see a distributed database in action, it is the complex logic that sits behind the distributed databases that makes sure that it is viewed as a database residing locally even though the data might be distributed over multiple geographical locations. Some benefits distributed databases bring to the table are improved performance, massive scalability, delivering round-the-clock reliability.

Traditional SQL databases can guarantee ACID (Atomicity, Consistency, Isolation, Durability) consistency but these are not scalable making them unfit to handle big data. They are perfect for use cases like banking systems. Key-value stores often relax the traditional ACID transactional model of database management systems and offer a BASE model (basically available, soft state, and eventual consistency) to trade off performance and availability for strict consistency. The BASE model is a foundation for reliably scaling the database in an efficient manner. It enables massive distribution and replication of the data throughout a large set of servers.

In this project, we have implemented a distributed key-value store with features like replication, scalability, and key topic subscription in place. We will discuss the strengths and weaknesses of this implementation and possible enhancements that can be implemented in the future.

2 PROBLEM STATEMENT

In order to mitigate users spamming servers when sending massive requests to stay updated, which would in average yield a very little number of meaningful responses, and to avoid that users have stale data due to lazy replication, we want in this project to implement a replicated, scalabled and distributed key-value store where users get the most up to date data and avert overloading the servers with requests through the introduction of a subscription/notification feature.

3 APPROACH

With a focus on avoiding stale data, we decided to extend our system with a subscription/notification feature, where clients subscribe to a specific key in order to get informed when such a key is updated or deleted by another client. In addition, we offer a topic based subscription, where a user can subscribe to all keys corresponding to a specific topic and get notified if any key corresponding to that stated topic get updated or deleted. We speak in the next subsections about them in more detail.

Authors' addresses: Mohamed Houssein Zaghdane, ge72xim@mytum.de, Technichal University Of Munich; Sanskar Gupta, sanskar.gupta@tum.de, Technichal University Of Munich; Emir Chalghaf, emir.chalghaf@tum.de, Technichal University Of Munich.

3.1 Subscription/Notification

Clients subscribe to a key by the Eventpublisher by sending a subscribe(key) request. The Eventpublisher acknowledges the subscription receive and saves clients metadata that subscribed to a specific key in order to notify all matching listeners registered to it as an update or deletion occurs.

3.2 Topic based subscription

Clients can not only subscribe to a key but also to a specific topic. A key/value can exactly belong to one topic and a topic can have many key/values affiliated to it. In order to subscribe to a topic, clients send a request to the EventPublisher specifying the topic interested in. EventPublisher acknowledges the subscription receive and saves subsequently metadata of clients that want to get notified by the update or deletion of a key belonging to a specific topic.

4 ARCHITECTURE

The system is comprised of the following components:

4.1 External Configuration Service

The External Configuration Service (or ECS) is the main coordinator of the whole architecture. Its main function is to maintain a ring topology of all connected servers. The ring topology is a circle composed of storage servers that allow for an efficient distribution of all possible key values around all available servers. The ECS achieves this by:

- Update the topology metadata when a server joins the ring and quickly inform the affected servers of the next steps to re-balance their stored keys with the new server and readjust their responsibility range.
- Consistently monitoring servers and quickly updating the metadata in case of an unresponsive server. A gracious shutdown is also coordinated with the server, if possible, in order to preserve the server's storage data.

Servers can only join the ring by communicating with ECS. While servers handle the key transmission by communicating directly with each other, The ECS handles informing each server of the address of the neighbor they need to communicate with.

The ECS also registers the Event publisher similarly to the storage servers by saving their address in the metadata with a unique identifier to differentiate it from storage servers. The ECS also represents a single point of failure. In case of failure, the servers would immediately lose any way of joining the network, and communication between the different servers would become impossible.

4.2 The storage server

The storage server is the core of the architecture. The storage server's main functionality is to allow clients to quickly put, get and remove key and the value/topic mapped to it from the persistent storage. Multiple servers can run in parallel to split the workload. The server must communicate with all the different parts of the system to correctly handle storing and delivering up to date values to the clients. The storage server runs two separate sockets running on different which are used for communication with respectively Clients and the rest of the architecture parts. This allows for a higher availability to answer client requests in case of a heavy key transmission being in progress.

For our extension, the server acts as a middle-point that handles forwarding forwarding any successful key update to the event publisher which then handles the subscription distribution. The server also was extended to be able to store the key topic in addition to its value. The topic is entirely optional and the server can also handle topic-less keys. Client are allowed to override topics when updating a key. The storage server features are briefly described in the following subsections :

4.2.1 ***Storage***. The storage server has two types of storage :

- A volatile cache that stores a limited amount of keys depending on the strategies used, which are respectively "Least recently used" (LRU), "First in first out" (FIFO) and "Least frequently used" (LFU). This allows for quick access to frequently accessed keys.
- A persistent storage file that contains all the keys the server is responsible for and a separate storage file for replication.

Both types of storage keep a consistent data set.
The Topic is stored as a prefix to the value using a special flag so it can be easily parsed and unparsed when retrieved from storage.

4.2.2 ***horizontal Scaling***. Multiple servers can run in parallel each handling a subset of the stored keys used consistent hashing. The server recognizes if a key is their responsibility or not and allows the client to update their metadata so they can also recognize which server to contact. The storage servers also are able to transmit and receive keys from neighbouring servers when the ring topology changes in coordination with the ECS.

4.2.3 ***Replication***. Replication adds another layer of protection to the data by replicating it on different neighboring storage servers. Replicas are allowed to answer get requests but not put requests for the replicated keys. The replication is lazy, which means users receive a put success response message before replication is achieved and successful. This means users may receive outdated key values when requesting them from a replica server.

4.2.4 ***Failure Detection***. In case of a failure, the server tries to achieve a gracious shutdown. This is achieved by quickly transmitting all the stored keys to a neighbor server in coordination with the servers. This shutdown mechanism isn't perfect and keys may be lost, especially in the case of servers having a large dataset.

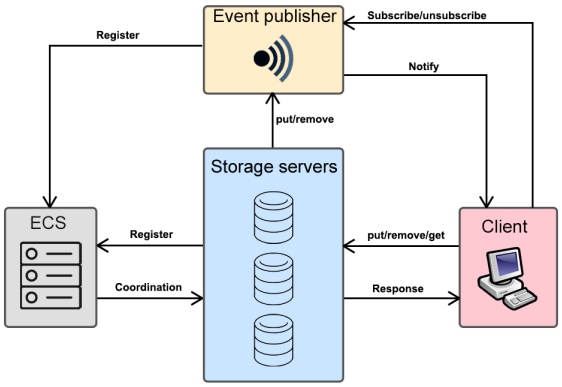


Fig. 1. The system's architecture flow

4.3 The client Library

The client library allows any user to communicate with the storage servers easily and observe the server output. The client interface displays the multiple functionalities offered by the servers and discards badly written requests or modifies user commands to transfer needed information to the servers for the treatment of the requests. The user must know the address of a single storage server where the initial metadata will be pulled, the rest of the addresses will be automatically parsed by the library. The event publisher address is parsed from the metadata. Possessing the event publisher address allows the user to access the notification system for any key or topic they desire. The client library also has a non-blocking socket running in the background where the Event publisher will inform it of any subscribed keys or topics updates.

4.4 Event Publisher

The event publisher represents the core part of our new feature. Its unique function is to notify subscribers about a particular key and its optional related topic of any update to its value.

The event publisher offers these two new functions to the client :

- Subscribe -t key
- Forward any key updates received from the storage servers to the subscribed clients. If the key-value pair

It is started as a separate service if one has to use the subscription feature. It handles the topic and key subscriptions by maintaining the data about the topic, key, and current subscribers. It provides an interface having methods like subscribe, unsubscribe and notify. Whenever there are changes in the client subscriptions whether topic or key, the client is notified about the changes.

5 EXTENSIONS

The pub/sub pattern is a pragmatic way of constructing message exchanges among entities such as services, for example. Instead of communicating directly with each other, in the publisher/subscriber pattern services can communicate via a message broker(EventPublisher in this project). This approach decouples the concerns of publishers

and subscribers: publishers can focus only on publishing, and subscribers can focus only on which keys or topic they're subscribed to. The resulting setup allows for asynchronous sending and receiving of messages by a broker dedicated to the task, which is one of several ways you can implement event-based systems.

Some technologies that can be used for pub/sub messaging include Redis, RabbitMQ, and Apache Kafka.

5.1 Notifications using publisher subscriber pattern

5.1.1 ***Problem.*** One of our goals is that clients always have the newest updated data. However, our system is using eventual consistency in that case replicas may send stale data which would lead to old values being sent to clients. In order to prevent that and in order to guarantee that our clients receive as soon as possible the last up-to-date data, we use a subscription notification model where clients subscribe to keys by the Eventpublisher and get notified in case of deletion or updation.

5.1.2 ***Feature description.*** This feature enables a client to subscribe to the key and receive regular updates about that key. The updates include deletion and updation of the key to which the client has subscribed to. The key component in this mechanism is the EventPublisher. It receives messages like subscribe and unsubscribe from the client with keys. It maintains the current subscription data which includes subscribers and keys. The Eventpublisher also receives messages from storage nodes namely 'putsuccess' and 'deletesuccess'. After receiving these messages the event publisher goes through the subscriber data and notifies the clients subscribed to the key which just got changed. It is to be noted that the EventPublisher is started as a separate Nio server where EventPubCommandProcessor is responsible for communication between clients(subscribers) to EventPublisher and storage nodes(publishers) to EventPublishers.It also acts as a single point of failure for the notification mechanism. When a client connects to a storage node, notification receiver is started as well to receive notifications from the event publisher in case there are any key subscriptions.

5.1.3 ***Feature implementation.*** The event publisher has been implemented as a separate server. The event publisher behaves similarly to a storage server at start time, it registers to the ECS, is added to the metadata and listens and treats any request received. The Event publisher offers the possibility to subscribe and unsubscribe to clients that contact it directly. Clients can obtain the event publisher's address by acquiring the metadata. These subscriptions are saved to a map, where each key name is mapped to a list of subscribed clients, Similarly to an Observer pattern.
When a client successfully updates a key, the server contacts the event publisher with the updated value. Servers are completely unaware if there is a subscriber to the updated key, which means also useless updates are sent to the event publisher. The event publisher then simply redirects the updated value to all subscribers. Neither the subscribers nor the publishers possess any knowledge of each other and the event publisher acts as a coordinator, this allows us to easily add and remove both publishers and subscribers.

5.1.4 ***Possible improvements.***

- The event publisher represents a new single point of failure for the whole feature. This means the whole architecture now suffers from two points of failure (the ECS and the event publisher). The system could first be improved by adding backups of subscribers to prevent the loss of the addresses in case of failure. A way to remove both single points of failure is also to make each server the functionalities of the ECS, storage server and event publisher with a leader elected and updated in case of failure. Each update would then be propagated in the ring and each server would send its set of notifications.
- If a user is disconnected when an update happens, they have a way of getting the missed updates when reconnecting. This is not ideal for clients that want to track value updates over time. An additional update history stack can be kept which would allow the user to request all missed notifications from the event publisher when reconnecting.

5.2 Topic-based Matching

5.2.1 ***Problem.*** The notification implementation, while offering the option to receive values updates without having to constantly pull from the server, still requires the user to subscribe to each single value manually with a request to the server. For inventory or tracking type of applications, where multiple values are tracked and constantly updated, it is impossible to find information about a topic of interest without knowing exactly the key, and a high amount of tracked keys means a high amount of subscriptions which is a burden for both the event publisher and the client.

5.2.2 ***Feature description.*** Topic-based matching and subscription allow users to subscribe to a topic instead of a key. Multiple keys can share a topic. The feature allows for an easy way to categorize keys in subgroups and for users to subscribe to multiple keys in a single request. When a key is updated, the event publisher matches the updated data's key and topic subscribers and notifies them of the new value

5.2.3 ***Feature implementation.*** As a basis to implementing topic-based matching, the possibility of adding topics to key/value pairs was first implemented. To achieve this goal both server and client libraries were extended to enable this feature to be possible :

- Server The server's permanent storage implementation only allows key-value pairs to be stored. To achieve assigning topics without rewriting the whole logic, topics were appended to the values with a special flag (-t=). This allows the server to recognize the topic from the value list when reading them from the storage file. It is possible to update key topics and it is also possible to update a key's value without changing its topic.
- Client The client library basic commands were extended with flags, which allow the user to optionally specify a topic when doing a put request and to specify the type of subscription (key or topic) when sending a request to the event publisher. The client's socket was also extended to correctly parse the data's key, value and topic (if the key is not topic-less) when getting a notification.

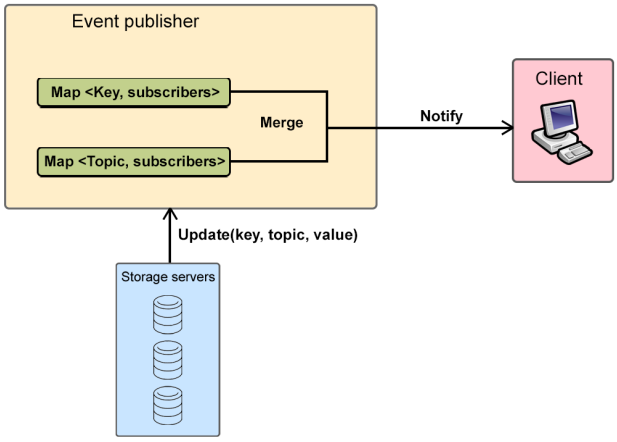


Fig. 2. Event publisher flow

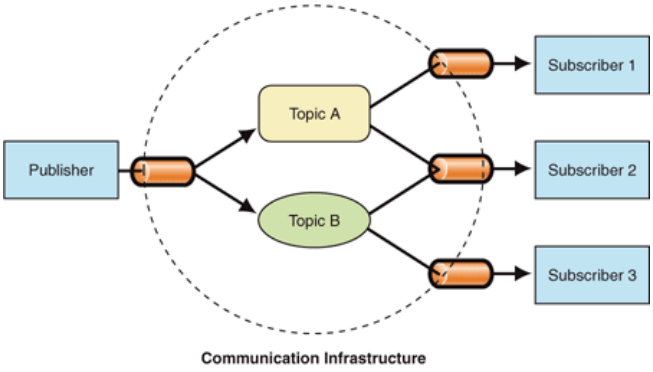


Fig. 3. Topic subscriptions

The event publisher was extended to keep a separate map for topic subscriptions to prevent collisions between topic and key names. This also means the key and topic subscription systems are decoupled and any can be easily disabled without breaking the functionality of the other. when a message is received from the storage servers, both the key and topic subscribers to the received update are matched and merged to prevent any duplication, and the resulting subscribers and then contacted with the update as seen in the figure. 2

5.2.4 *Possible improvements*.

- Topics open the door for a lot more attributes that could be used to define each key which would allow for more complex and specific subscriptions with content-based matching.
- Any user can now assign any value to any key. When extending keys to have a topic, the question of who should be allowed a topic was an issue. The whole system could be appended with an authorization system to protect topics or keys from being modified by unauthorized users.

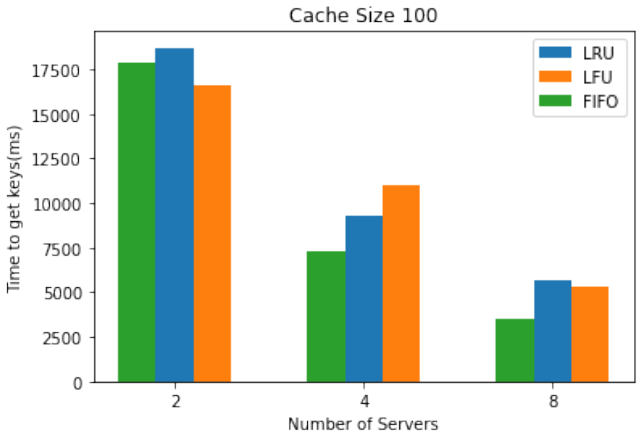
6 PERFORMANCE ANALYSIS

The tests were performed on the following system with specifications specified in Table 1

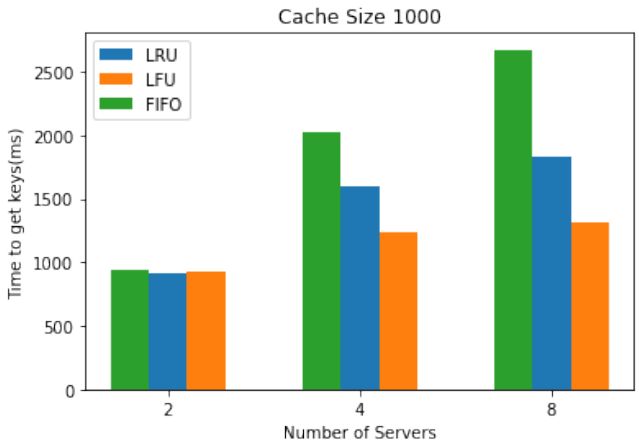
Table 1. Specifications

Operating System	Ubuntu 20.04.2 LTS
Java Version	16
Processor	i7-9750H CPU
Clock Speed	2.60 GHz
Number of Cores	6
Number of Threads	12
RAM	16GB

6.1 GET REQUESTS THROUGHPUT



The above plot shows the get requests being performed on 1600 keys. From the above plot, we can infer that as the number of servers is increasing the performance improves. This can be attributed to the fact that every server has a separate storage, hence the problem of accessing a single storage while performing bulk get operation diminishes. As cache size is small as compared to the number of keys FIFO cache strategy performs best because there are not many operations on cache and most of the data is written to file storage.

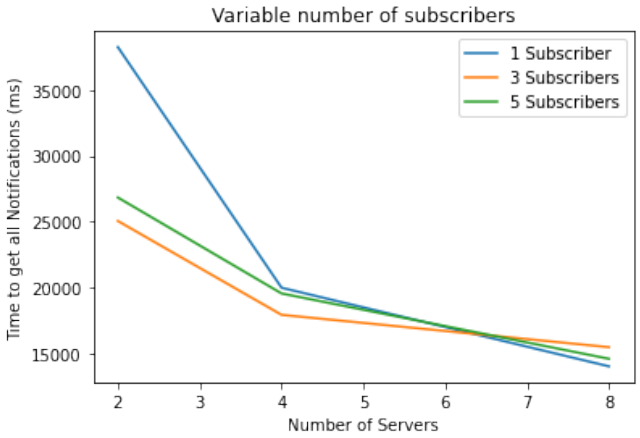


The above plot shows the same get requests throughput but with cache size as 1000 which is comparable to the number of keys.As most of the keys are accessed from an in-memory cache, increasing the number of servers decreases the overall get throughput. It is to be noted that "Least-Frequently used" cache strategy performs the best in the scenario where a cache is used extensively which is true in the above scenario.

6.2 KEY SUBSCRIPTION PERFORMANCE

A test was performed to determine the time elapsed to notifications for all subscribed keys. A client was bootstrapped which inserted 1600 keys. This client was made to subscribe to all the keys. A second client deleted all the keys which triggered notification for every key. The time to receive notifications for all the deleted keys was recorded for a variable number of servers. It is to be noted that the performance is increased by adding more servers as data gets evenly distributed up to 8 servers hence key update/deletion becomes faster which in turn triggers notification quickly through the event publisher.

6.3 TOPIC SUBSCRIPTION PERFORMANCE



In this experiment, a client was bootstrapped which inserted 1600 keys where all the keys were assigned a topic. The client was made to subscribe to this topic. A second client was started which performed a delete operation on all the keys inserted. The notifications generated as a result of key deletions were recorded on the first client. The time to receive all notifications as part of topic subscriptions was recorded for a variable number of servers. The performance increased again by increasing the number of servers due to the keys being distributed evenly. The test was also performed for a different number of subscribers subscribed to the same topic. It was noted that the number of active subscribers and time to receive all notifications are not much correlated.

7 CONCLUSION

The Milestone 5 brought us a wide range range of ideas that could have been implemented. But due to time constraints, we decided to go with the key subscription notification feature which we extended to topic-based subscriptions as well. In the current system, we have 2 single points of failures namely ECS and the event publisher which will be rectified in future works. As far as consistency is concerned, the system is eventually consistent providing with the base consistency and is horizontally scalable which was clearly evident from the performance testing where the addition of more servers increased throughput in some scenarios. There is always a trade-off between availability and consistency and one must optimize both as per the current design and use cases of the system. This project made us realize that it can be tricky to design and architect a distributed system that can provide stronger consistencies while keeping a simple logic and event flow.

While the project reached a consistent performance and functionality, implementing the different parts of the system made us realize the different features the implementation can still be improved with. Besides the previously mentioned possible improvements, the system lacks security and all parts of the system could be improved with multiple layers of authorization as a next step.