Cloud Databases

Milestone 4: Distributed & Replicated Storage Service

Overview

Reliability and performance are vital prerequisites for large-scale web data platforms. A very prominent concept to enhance these properties is replication.

Replication introduces redundancy to the data by storing single data items on various (usually physically detached) locations. This increases availability, since single failing nodes can be tolerated. Furthermore, this forms the basis for enhanced performance. Client requests can be distributed among the different replicas and consequently the load on single servers is reduced.

The objective of this assignment is to extend the storage service from Milestone 3 by means of replication. Data records (i.e., key-value pairs) are still distributed over a bunch of storage servers via consistent hashing. However, the storage servers are no longer just responsible for their own subset of the data, but also serve as replicas for data items of other servers. In this sense, a storage server assumes different roles for different tuples. A storage server may be a

- **coordinator node**, if it is directly responsible for the tuple, following the concept of consistent hashing. This means it has a position in the ring, such that it is the closest successor server in the ring topology according to the tuple position (cf. Milestone 3).
- or a **replica node**, if the tuple is coordinated by another storage node and the data is just replicated on this node. It is either *replica_1* if it is first successor of the coordinator or *replica_2* if it is the second successor.

Each data item should be replicated exactly on the two storage servers that are following the coordinator node in the ring topology. Replication is invoked and managed by the coordinator node. This means that at least 3 active and responsive storage servers are needed. As long as just 1 or 2 nodes are in the system, no replication is used.

The focus of this milestone is to implement a replication strategy that guarantees **eventual consistency**. It should also provide a **reconciliation mechanism for topology changes.** So far (as completed in Milestone 3) the key ranges are rebalanced. In this milestone, the replication is rebalanced when the topology changes.

Learning objectives

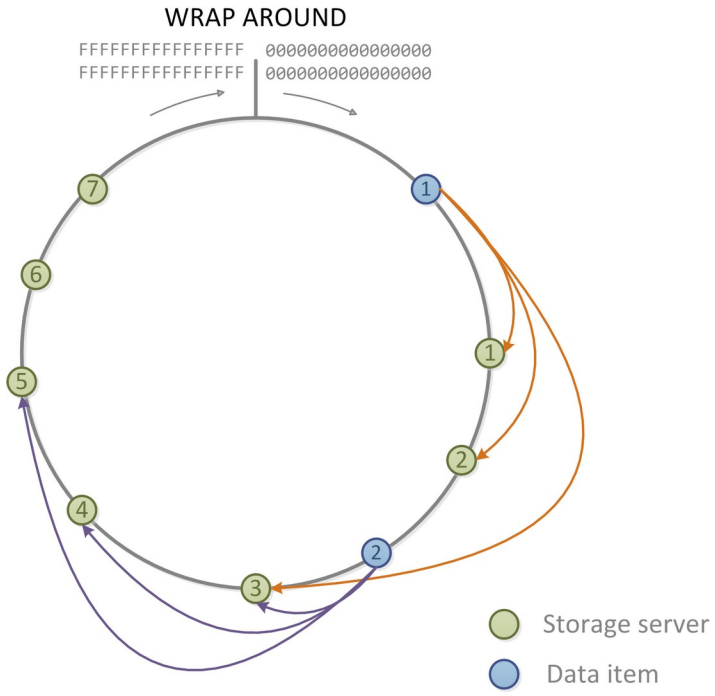
With this assignment, we pursue the following learning objectives:

- deal with a distributed system of enhanced complexity
- understand the necessity for replication in large-scale distributed systems, and
- get exposed to the concept of eventual consistency.

Detailed assignment description

Assigned development tasks

Extend ECS and KVServer by a means to implement replication by guaranteeing eventual consistency.



Eventual Consistency

Eventual consistency is a weaker form of consistency and mostly suitable for read dominant systems. It ensures that, provided there are no updates for a “long” time, all replicas will gradually become consistent.

System Reconciliation under Scaling

To maintain the replication invariant (i.e., every tuple is replicated on the two successor nodes of the coordinator) even in cases of adding or removing nodes from the storage service, the ECS must reallocate not only the data on the coordinator node but also on replicas. Think of an efficient mechanism to implement this behavior. The ECS should coordinate the task but the rereplication should be between two KVStores alone. Also consider that multiple new KVStores can join while others are leaving and loss of data should be minimized.

Failure Detection & Reconciliation

Similar to the behavior under up/down scaling, the system must maintain the replication invariant also in the case of failure (i.e., crashing nodes).

After a failing node has been detected, the ECS does the following things: The storage service is reconstructed according to the new set of available nodes (i.e., recalculation of responsibilities, update of metadata, and reallocation of data to coordinator and replica nodes.)

Extension of Client library

The command “keyrange\r\n” from the previsous milestone returns the servers which can be used for read/writes or put and gets. The client library is extended with a new command “keyrange_read\r\n” which returns all ranges and corresponding kvstores which can fullfill get requests. That way read heavy workloads can also read from multiple replicas. All puts from the client should always go to the coordinator and gets should go to random responsible node.

Command	Informal description	Parameters	Shell output

`put <key> <value>`	Sends a key value pair to the server	**key**: String (ISO-8859-1) (may not contain delimiter \r\n)
		
		value: String (ISO-8859-1) (may not contain delimiter \r\n), e.g. use Base64 encoding
			server reply:
			`put_success <key>`
			`put_update <key>`
			`put_error <key> <value>`
			
			`server_stopped`
			`server_not_responsible`
			`server_write_lock`

`get <key>`	Gets the value of a key	**key**: String (ISO-8859-1) (may not contain delimiter \r\n)	**server reply**:
			`get_error <key> <msg>`
			`(e.g., key not found)`
			`get_success <key> <value>`
			
			`server_stopped`
			`server_not_responsible`

`delete <key>`	Delets a value	**key**: String (ISO-8859-1) (may not contain delimiter \r\n)	**server reply**:
			`delete_success <key>`
			
			`delete_error <key>`
			
			`server_stopped`
			`server_not_responsible`
			`server_write_lock`

`keyrange`	Keyrange of	**No parameter**	`keyrange_success`

	the KVServers		`<range_from>, <range_to>,<ip:port>;...`
			
			`server_stopped`

`keyrange_read`	Keyrange of the KVServers including replicas	**No parameter**	`keyrange_read_success <range_from>, <range_to>,<ip:port>;...`
			
			`server_stopped`

`Any`	Arbitrary request		**Server reply:** `error <description>`

Guidelines for implementation:

Within this milestone, **you have the opportunity to make most design decisions on your own**. The only requirement is that the API for the client is extended with the command keyrange_read.
However, you should adhere to the following guidelines:

- Write requests need to be served by the coordinator node, whereas read requests may also be served by any replica. Note, since the implementation should only guarantee eventual consistency the result value may be stale).
- Replication is invoked and managed by the coordinator node. Note, that the coordinator also has to manage deletion and updates to the data, i.e., if a tuple has been deleted the coordinator also has to ensure deletion of the data on the replicas.

Suggested development plan

Change the remote URL of your Git repository to your Milestone 4 Git repository on GitLab by executing "*git remote set-url origin Milestone4GitRepo.git*". You should base design and implementation of your replication strategy on your results of Milestone 3.

Deliverables & Code submission

You must commit and push all the deliverables to the GitLab by the deadline. After the deadline, your GitLab repository's role will change from ***Developer*** to ***Reporter***. You will be able to pull the code from GitLab, but you won't be able to push new changes to the repository. You must hand in your software artifacts that implement all the coding requirements and include all necessary libraries and the build script.

Marking guidelines and marking scheme

All the code you submit must be compatible with the build scripts, interfaces and test cases that we propose with the respective assignment. In addition, your code must build and execute on *java version 11,without any further interference, additional libraries beyond those specified in the provided pom.xml (see Milestone 3), and provide the specified functionality.*

Testing procedure:

1. We build your project using “mvn package”.
2. Start of ecs, e.g., “java -jar target/ecs-server.jar” -l ecs.log –ll ALL -a 192.168.1.1 -p 5152
3. Start of two KVStores, e.g., “java -jar target/kv-server.jar -l kvserver1.log -ll ALL -d data/kvstore1/ -a 192.168.1.2 -p 5153 -b 192.168.1.1:5152”
4. Connect our own kv client executing a test workload, e.g., multiple puts, gets, …
5. As long as just 2 KVStores are available, no replication is active, hence keyrange and keyrange_read should be the same
6. Add another KVStore (3 in total, replication should start)
7. Check the range of keyrange_read since it should be possible to read from all since every value is replicated twice. Get a key from all nodes.
8. Adding another KVStore
9. Kill one KVStore (no graceful shutdown)
10. Add multiple KVStores and remove multiple at nearly the same time.
11. General source code assesment, tests, ecs communication, …
12. *Checking the source code using code duplication tools by comparing to submissions from previous semesters*