Cloud Databases

Milestone 3: Scalable Storage Service

Overview

Large-scale, web-based applications like social networks, online marketplaces, and collaborative platforms have to concurrently serve millions of online users and handle huge amounts of data while being available 24/7. Today's database management systems, while powerful and flexible, were not primarily designed with this use case in mind. Key-value stores try to fill this gap by offering a simpler data model, often sufficient to support the storage and query needs of web-based applications.

Key-value stores often relax the traditional ACID (atomicity, consistency, isolation and durability) 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.

The objective of this assignment is to extend the storage server architecture from Milestone 2 into a elastic, **scalable storage service** (cf. the figure below). Data records (i.e., key-value pairs) are distributed over a number of storage servers by exploiting the capabilities of consistent hashing. A single storage server is only responsible for a subset of the whole data space (i.e., a range of successive hash values.) The hash function is used to determine the location of particular tuples (i.e., hash values of the associated keys).

The **client library** (KVStore) enables access to the storage service by providing the earlier defined KV-Storage interface (connect, disconnect, get, put). Furthermore, the library keeps the distribution of data transparent to the client application. The client application only interacts with the storage service as a whole, while the library manages the communication with a single storage server. In order to forward requests to the server that is responsible for a particular tuple, the client library maintains metadata about the current state of the storage service. Meta-data at the client side may be stale due to reorganizations within the storage service. Therefore, the library has to process requests optimistically. If a client request is forwarded to the wrong storage node, the server will answer with an appropriate error and the most recent version of the meta-data. After updating its meta-data, the client will eventually retry the request (possibly contacting another storage server.)

Each **storage server** (KVServer) is responsible for a subset of the data according to its position in the hash space (the ring, as depicted in the below figure.) The position implicitly defines a sub-range of the complete hash range.

The storage servers are monitored and controlled by an **External Configuration Service (ECS)**. By means of this configuration service an administrator is able to initialize and control the storage system (i.e., add/remove storage servers and invoke reconciliation of meta-data at the affected storage servers). In this milestone this is single point of failure.

Learning objectives

With this assignment, we pursue the following learning objectives:

- Understand & build an incrementally scalable storage service based on the software artifacts developed in the previous milestones: Independent set of storage servers to provide horizontal scalability based on consistent hashing
- Control the storage service with an external configuration service (ECS)
- Access the storage service with a library that provides a client-level abstraction in the storage server (similar to the client library developed in earlier milestones for a single storage server)
- Understand the concept and advantages of consistent hashing
- Get exposed to caching in order to reduce network traffic
- Conduct performance measurements

Detailed assignment description

Assigned development tasks

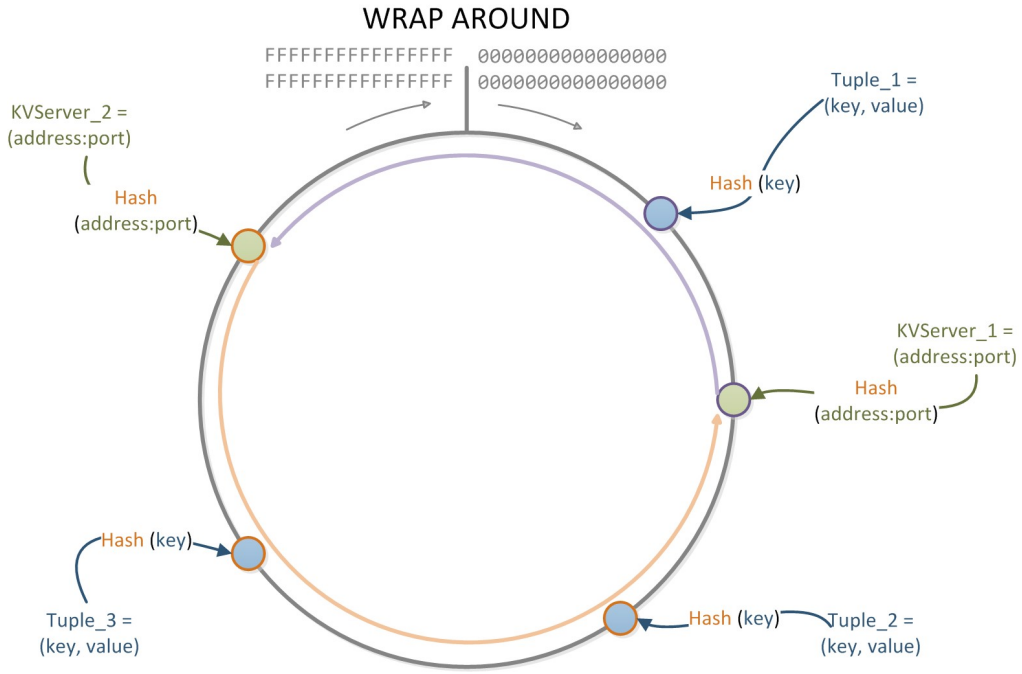
1. Develop or extend the following key components based on the components from previous milestones
    - Client library (KVStore)
    - Storage server node (KVServer)
    - External configuration service (ECS)
    - The protocol between the ECS <-> KVStores and KVStore <-> KVStore is not defined. You should come up with your own protocol. You can handle the KVStore<->KVStore communication if necessary on a separate random free port (see Tests Util.getFreePort()). It should consider the following
2. External configuration service (ECS)
    - The ECS is started first at a specified ip and port (e.g., -a 192.168.1.24 -p 5144). KVServers are started using the ECS as bootstrap server and provide the KVServer interface at the specified IP and port. (e.g. -b 192.168.1.24:5144 192.168.1.24 -p 5153)
        i. Joining of a new KVStore
        ii. Removal of a KVStore
        iii. Starting of rebalancing
        iv. Once finished, activate KVStores for KVClient interaction

        v. Keep alive to detect KVStore failures (The keys are lost in that case)
    - Once the first KVServer connects to ECS
        i. Compute key-ranges for the KVServer and disseminate meta-data
    - Incrementally add a new KVServer
        i. Compute key-range position of the KVServers
        ii. Inform neighbor to initiate key-value hand-off
        iii. Update meta-data of affected storage nodes
        iv. Read requests are always served
    - Remove a node
        i. Re-compute key-range position for affected KVStores
        ii. Inform neighbor to initiate key-value hand-off
        iii. Update meta-data of affected storage nodes
        iv. Read requests are always served
3. Client library (KVStore)
    - Cache meta-data of storage service. (Note: this meta-data might not be the most recent. The assumption is that the meta-data is valid until the server returns a message "`server_not_responsible`")
    - Route requests to the storage node that coordinates the respective key-range
    - There might be meta-data updates, initiated by the storage server if the library contacted a wrong node (i.e., the request could not be served by the currently connected node) due to stale meta-data
    - Update meta-data and retry the request
4. Storage server node (KVServer)
    - The KVServer process is launched via commandline and ECS is set as bootstrap server (see Config.java). All client requests are responded with a "`server_stopped`" messages unitl key-ranges are balanced. Only the ECS is able to configure the KVServer and activate it for client interaction (i.e., serving of client requests).
    - Extensions needed for the KVServer
        i. Assign a key-range to the server and incorporate the server to the storage service (**activate** KVServer, handle client requests)
        ii. Hand-off data items to another server (in case of reorganizing the storage service due to added/removed storage nodes)
        iii. Update the meta-data
    - Request processing
        i. All requests are processed locally
        ii. If the storage server is not responsible for the request (i.e., key is not within its range), the server answers with an error message that also contains the most recent meta-data
5. Performance evaluation

Consistent Hashing

The figure below outlines the basic ideas behind consistent hashing as it applies to this assignment. All storage nodes (KVServers) are arranged clockwise in a logical ring topology (the underlying physical topology may differ.) Each position in the ring corresponds to a particular value from the range of a hash function. Note that with consistent hashing, values wrap around at the end of the range (i.e., the last hash value from the range is followed directly by the first one).

In this assignment we are going to use the Message Digest Algorithm 5 (**MD5**) for all hash operations. This hash function calculates a 128 bit digest (32 Hex digits, cf. figure) for arbitrarily large input data (i.e., array of bytes). The Java standard API already provides an implementation for this purpose (see java.security.MessageDigest).
The position of a particular storage server is calculated by hashing its address and port (<IP>:<Port>). Similar, the position of a (key, value)-pair is determined by hashing the respective key.



As a result, both servers and tuples are assigned distinct positions within the ring. Each server is responsible for (i.e., has to store) all tuples between its own position (inclusive) and the position of its predecessor (exclusive) in the ring. Note, according to the wrap-around characteristics of the ring, it may occur that the predecessor node has a higher position than the actual server. For example, the predecessor for KVServer_1 is KVServer_2, although KVServer_2 logically succeeds KVServer_1 (cf. figure below). The meta-data in the context of this assignment is formed together by a mapping of KVServers (defined by their IP:Port) and the associated range of hash values. Essentially, a range is defined by a start index (i.e. position) and an end index.

Client library (KVStore) & application (KVClient)

The client library (KVStore) provides an abstraction to query the storage service. The actual configuration of the storage service is completely transparent to the client application. Since KVStore provides the same functionality there is no need for changes in the client application (KVClient).

However, due to the distribution of the complete data set, the library has to forward each client request to the storage server that is responsible for the associated key. Therefore, meta-data about the storage service is maintained to identify the respective server. Initially, there is no meta-data available and the client application has to manually connect to one of the servers participating in the storage service (i.e., we assume that the user has to know at least one of the KVServers). Once a connection has been established, all requests are forwarded to this server (optimistic querying). Since data is distributed, it might occur that the request has been sent to the wrong storage node and could not be processed. In such cases the server would answer with a specific error message (see below). As a consequence, the library would update its meta-data (keyrange), determine the correct storage server, connect to it and retry the request. Retry operations are transparent to the client application.

The communication between the KVStore Client Library and KVServer has to be extended. Red denotes the protocol extensions compared to previous milestone. All messages on the wire are clear text in ISO-8859-1 delimited with "\r\n".

not_responsible is used by the storage server to respond to requests that could not be processed by the respective server because the requested key is not within its range. Such messages **should not** be passed back to the client application but invoke a retrieval of the keyrange. Define a suitable representation (e.g., a Map) for the meta-data that you keep consistent throughout all components of the whole system (i.e., KVStore, KVServer and ECS). This data structure maps the addresses of storage nodes to the respective hash-ranges.

server_stopped indicates that currently no requests are processed by the server since the whole storage service is under initialization. Hence, from the client's perspective the server is stopped for serving requests. The client retries several times using exponential back-off with jitter.

server_write_lock indicates that the storage server is currently blocked for write requests due to reallocation of data in case of joining or leaving storage nodes.

keyrange_success returns the ranges and which KVStores are responsible for the range. The range is expressed in hex (128bit). The list of ranges and the corresponding servers are returned as a list of semicolon separated triples <kr-from>, <kr-to>, <ip:port>; <kr-from>, <kr-to>, <ip:port>;...

```
"keyrange_success 6aa27ce2f47205...,98f6bcd4621d3...6,192.168.1.2:5120;
98f6bcd4621...7,a3...,192.168.1.3:5120; … \r\n"
```

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 the KVServers	**No parameter**	`keyrange_success`
			`<range_from>,`
			`<range_to>,<ip:port>;...`
			
			`server_stopped`

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

Storage node (KVServer)

According to its position in the ring, the storage server (KVServer) has to maintain only a subset of the whole data that is present in the storage service. As in Milestone 2, this data is stored persistently disk using one of the cash displacement strategies. A KVServer communicates with the KVStore library of client applications and processes their request (similar to Milestone 2). However, in order to determine if a particular request has to be processed (i.e., the requested key is in the range of this KVServer), it has to maintain the meta-data of the storage service.

Note, it is necessary to hold the **complete** meta-data to be able to inform clients in case of incorrectly addressed requests. Meta-data updates on the KVServer are only invoked by the ECS.

In addition to the query functionality (put, get), the server has to provide the following control functionality to the ECS. Since all these operations have to be invoked by the ECS remotely you have to define a messages and commands to pass the commands along with the parameters.

Once the KVServer application has been launched it is in the state stopped. That means it is able to accept client connections but will answer with SERVER_STOPPED. Then the KVServer connects to the ECS to retrieve the meta-data.

External Configuration Service (ECS)

The purpose of ECS is to control the KVStores and rebalance the hashring and initiate the transfer of key ranges between the KVServers.

The ECS uses the parameters defined in the sample project from the previous milestone.

New KVServer

When a new KVServer connects to ECS, the ECS updates:

- Determine the position of the new storage server within the ring by hashing the address of the server port used to communicate with the clients.
- Recalculate and update the meta-data of the storage service (i.e., the ranges for the new storage server and its successor)
- Initialize the new storage server with the updated meta-data.
- Set write lock on the successor node;
- Invoke the transfer of the affected data items (i.e., the range of keys that was previously handled by the successor node) to the new storage server. The data that is transferred should not be deleted immediately to be able to serve read requests in the mean time.
- When all affected data has been transferred (i.e., the successor sent back a notification to the ECS)
    - Send a meta-data update to all storage servers (to inform them about their new responsibilities)
    - Release the write lock on the successor node and finally remove the data items that are no longer handled by this server

Remove storage node

The steps for removing a KVServer instance from the storage service by the ECS can be summarized as follows:

- The KVServer sends the ECS a message that it shuts down. (Make sure to add a shutdown hook)
- Recalculate and update the meta-data of the storage service (i.e., the range for the successor node)

- Set the write lock on the server that has to be deleted.
- Send meta-data update to the successor node (i.e., successor is now also responsible for the range of the server that is to be removed)
- Invoke the transfer of the affected data items (i.e., all data of the server that is to be removed) to the successor server. The data that is transferred should not be deleted immediately to be able to serve read requests in the mean time
- When all affected data has been transferred (i.e., the server that has to be removed sends back a notification to the ECS)
    - Send a meta-data update to the remaining storage servers.
- Continue shutting down the storage server.

Continous monitoring of KVStores

Ideally, the KVStore shut down gracefully. This means that when the KVStore is about to shutdown, the ECS is informed, the ECS can rebalance the keys and once finished the KVStore if finally shut down. To detect if KVStores are still active, the ECS continously pings each. In case of network errors, or other faults, the KVStore may not be reachable. If one KVStore is not available any more, it is considered shutdown.
A KVStore is not available if it fails to respond within 700ms. The ECS pings all KVStores every second. The key value pairs are considered lost.

Testing with JUnit

1. Create connection / disconnect
2. Set value
3. Get value
4. Update value (set with existing key)
5. Get non-existing value (error message)

Besides fixing the test cases from Milestone 2, **define at least 5 test cases on your own**. These cases should cover the additional functionality/ features of this milestone (e.g., ECS, consistent hashing, meta-data updates, retry operations, locks, etc.)

Data sets and metrics for testing

Evaluate the performance of your storage service implementation and compose a short report (max. 2 pages). You can use the Enron Email data set to populate your storage service and run experiments. (e.g. key: maildir/badeer-r/all_documents-1 value: content of the file). Measure latency and throughput for read and write operations in varying scenarios.

- different number of clients connected to the service (e.g. 1, 5, 20)
- different number of storage nodes participating in the storage service (e.g. 1, 5, 10)
- different KV server configurations (cache sizes, strategies)

Also think of evaluating the process of scaling the system up & down, i.e., how long does it take in the different scenarios to add/remove a KVServer.
Your report should contain the quantitative results (e.g., plots) and an explanation of the results.

Suggested development plan

- Change the remote URL of your Git repository to your Milestone 3 Git repository on GitLab by executing "*git remote set-url origin Milestone3GitRepo.git"*
- Adjust the *pom.xml* from Milestone 2 with the new provided build file.
    - The project does not contain substantial changes. Things which might help you is the extended NIO loop which allows adding of outgoing connections to the existing NIO loop, hence a single thread can handle all server connections and outgoing connections to the ECS or other KVStores.
    - In the
- You should integrate your implementation of Milestone 2 into this project structure and try incorporating the extensions for the purpose of this milestone.

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.

Testing, Marking guidelines and marking scheme

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 multiple 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. Removal of one of the KVStores, "Ctrl+C" (be sure to attach to the shutdown hook)
6. Putting keys in all ranges, observe if the configuration has changed
7. Adding another KVStore
8. Kill one KVStore (no graceful shutdown)
9. Add multiple KVStores and remove multiple at nearly the same time.
10. General source code assesment, tests, ecs communication, …
11. Checking the source code using code duplication tools by comparing to submissions from previous semesters

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 be tested on *java version: 14,* without any further interference, additional libraries beyond those specified in the provided pom.xml, and provide the specified functionality.

Additional resources

- JUnit: http://www.junit.org/
- Maven build tool: http://maven.apache.org/
- Consistent Hashing: http://www.codeproject.com/Articles/56138/Consistent-hashing
    - David R. Karger, Alex Sherman, Andy Berkheimer, Bill Bogstad, Rizwan Dhanidina, Ken Iwamoto, Brian Kim, Luke Matkins, Yoav Yerushalmi: Web Caching with Consistent Hashing. Computer Networks 31(11-16): 1203-1213 (1999
    - David R. Karger, Eric Lehman, Frank Thomson Leighton, Rina Panigrahy, Matthew S. Levine, Daniel Lewin: Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. STOC 1997: 654-663