Cloud Databases

Milestone 2: Persistent Storage Server

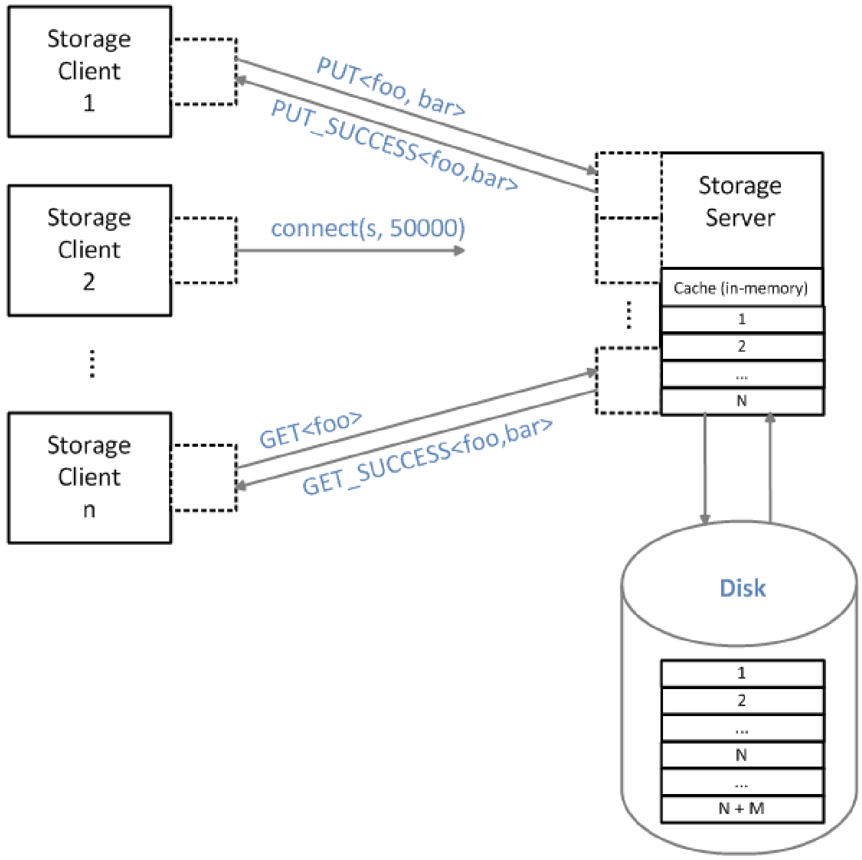
Overview

In recent years, several major internet players, such as Amazon, Facebook, Google, and Yahoo! have developed storage platforms that underlie their online properties and services, which are characterized by a unique set of requirements.

These storage platforms are commonly based on a data model that knows about keys and associated values (e.g., a user's email address and a corresponding value that captures information about the user.) Thus, data is represented as (key, value)-pairs, also known as tuples, such that the actual data item, i.e., the "value", is indexed by a unique "key". Essentially, the query interface of such data stores consists of methods for the insertion (put) and the retrieval (get) of data items. The simplicity of this model enables the development of storage server architectures that are highly scalable, reliable in the face of failures, and efficient in supporting worldwide data availability. It is not untypical for these systems to need to reliably serve petabytes of data, which is achieved through large-scale distribution and replication among a large set of storage servers.

The **objective of this assignment is to implement a simple storage server** that persists data to disk (e.g., a file). In addition, a small fraction of the data, i.e., a configurable number of key-value pairs can be cached in the servers' main memory. The storage server should provide the typical key-value query interface. To achieve this objective, **the echo client from Assignment 1** should be extended to be able to communicate with the storage server and query it. These tasks require the development of an application communication protocol and a suitable message format.

In this assignment, a single storage server will serve requests from multiple clients, whereas in the next assignment, we'll experiment with deploying a number of storage servers based on the artifacts developed in this and the previous assignment. The logical client-server architecture of this assignment is depicted below.



Learning objectives

With this assignment, we pursue the following learning objectives.

- Understand the client-server paradigm and get exposed to more advanced socket programming from the client's & server's perspective
- Understand how multiple clients interact with a single server
- Implement cache displacement strategies for offloading data to disk (i.e., FIFO, LRU, LFU)
- Learn to develop a basic protocol made up of messages between client and server
- Learn about and rigorously apply unit testing and religiously deal with failures
- Conduct performance evaluations

Detailed assignment description

Provided infrastructure

Together with this handout that specifies the storage server interface, we provide you with a project stub in GitLab repository including a set of JUnit test cases, which your program should pass before submitting, some interfaces (see below) and the build script for building and testing the program. You should use this stub to start your implementation.

Storage Server API

The storage server should provide the following interface:

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>`

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

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

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

The requests and responses are kept in order based on the granularity of the TCP connection. As an example, if the client sends a “put k foo”, a millisecond later a “put k bar”, it should receive in order the responses “put_success k” and “put_update k”.

Assigned development tasks

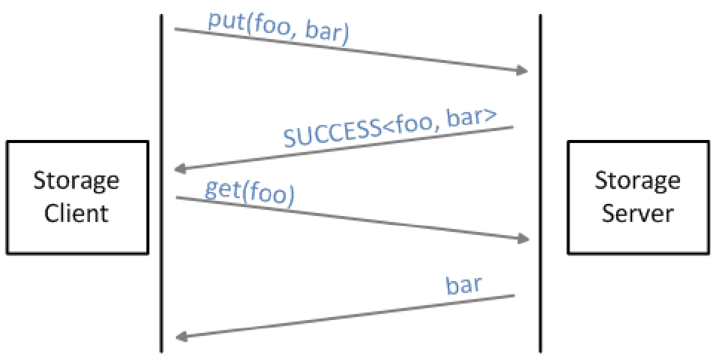
In this assignment, the following components need to be developed:

- Extend communication logic from Assignment 1
    - Implement the parsing of the commands on both client side and server side.
    - Handle all error cases
- Storage server comprised of

    - Multithreaded or non-blocking storage server. Independent of which method is used, disk access should not block network communication and vice versa.
    - Failure handling to gracefully deal with errors, exceptions and failure conditions.
    - Caching mechanism (i.e., in memory data maintenance) for a configurable number of key-value pairs.
    - Implementation of a persistence mechanism for key-value pairs to store data permanently on disk
    - Implementation of 3 different cache displacement strategies (FIFO, LRU, LFU).
    - Separate logging capability to log server behavior and interactions.
- Storage client program comprised of
    - Application logic: Command line shell to interact with the server and support queries to the storage server.
    - Communication logic: Socket-based communication by reading and writing from/to a byte buffer.
    - Clean separation of application logic and communication logic
    - Separate logging capability to log client behavior and interactions.
- JUnit5 Tests
    - Integration of the JUnit library into your project.
    - Automate the running of the given test cases
    - Add at least 3 test cases of your choice that cover relevant system behavior not yet tested in the given test cases
    - Compile a short test report about all test cases, especially your own ones (short pdf document).

Communication logic

The figure below shows an excerpt of a possible communication session between a client and the storage server.



The first two messages represent a typical put interaction. Essentially, the put request message consists of at least the two parameters “key” and “value” and a flag that identifies the actual operation (i.e., put in this example). After the tuple has been successfully inserted into the storage, the request is going to be acknowledged by the server (put_success <key>).

The second message pair represents the retrieval functionality. The client issues a message that contains at least the query key and again, the type of operation (i.e., get in this case). As a result, the server looks up the value for the desired key and hands it back to the client. If the server is not able to find the requested value, it will return an error message.

Messages

Since the communication between client and server is based on the Telnet protocol, hence in clear text (ISO-8859-1) and delimited with a carriage return and line feed (\r\n), we must encode arbitrary byte data in a different format, similar to Base64. Base64 is not ideal as it leaves many other useable characters, e.g. (&%§%…), but is a good starting point.
Good programming practice suggests reusing message parsing to concrete objects on both client and server.

Protocol

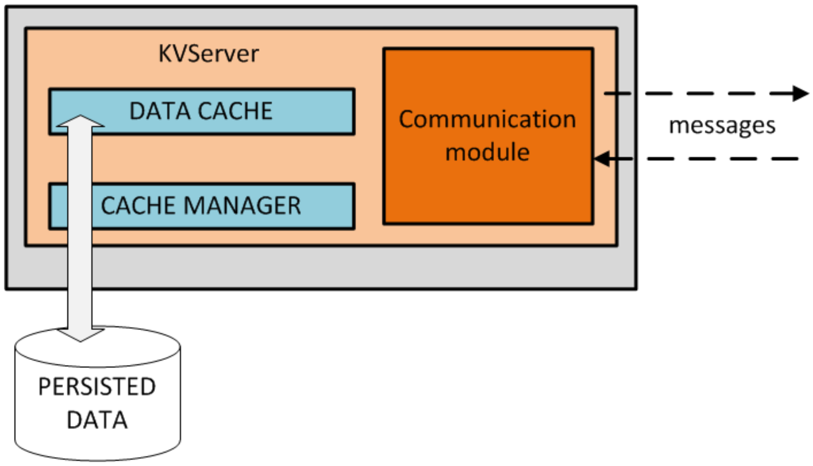
Among others, consider the following scenarios:

- During a put request:
  - The server successfully inserted a tuple into its data structure and has to confirm this to the client.
  - The server may be unable to insert the tuple due to some unpredictable event (e.g., error), hence it should inform the client about this situation.
- During a get request:
  - The tuple may not be present at the storage server. Hence an error has to be passed back.
  - In the case of a successful look-up, the data is returned.

Think of other possible failure scenarios and consider them. Note, it is important that the ordering is guaranteed so that the client can correlate the server reply with a particular request it previously issued.

Server program

The storage server is responsible for maintaining the key-value pairs and providing access to clients. As already indicated above, data must be stored durably on a disk. The directory where the data should be stored is specified on the command line. In addition to storing each key-value pair on disk, a configurable number of pairs should be also stored in memory to speed-up the lookup process for GET requests (parameter ‘cacheSize’ of KVServer). The general architecture for a storage server is depicted in the figure below.



Persistence & Caching

Think of an efficient data-structure for caching as well as an efficient file-structure for disk persistence to manage key-value pairs (i.e., fast insertion and lookup) and implement the following three strategies to displace data from cache to disk.

- FIFO (First in First Out)
- LRU (Least Recently Used)
- LFU (Least Frequently Used)

Cache displacement of key-value pairs is necessary in two distinct cases:

1. If a GET-request by a client causes a cache miss, the respective key-value pair is looked-up on disk and transferred to the cache. If the cache is already full, one particular tuple is displaced to disk (and potentially updated there) according to the currently selected strategy.
2. If a PUT-request by a client is received by the server and the cash is already full, then again one particular tuple is displaced according to the currently selected strategy.

Concurrency

Furthermore, the concurrent handling of multiple client connections is an inherent prerequisite for our storage server and for other server implementations in general. A generic way to realize concurrency can be found in the implementation of our skeleton code (i.e., the echo-server from the previous assignment). If you start with the Java NIO based or the thread per connection approach, the main goal to reach is that network communication is not blocked by disk operations or CPU intensive computations. Note, that it is very important to guarantee the consistent manipulation of shared data in the face of concurrent accesses.

The server should also contain a logging facility that logs all the relevant program behavior during its execution to a specific log file (server -l logs/server.log). The log level of the server is controlled by a command line parameter (server -ll INFO).

The port number (int), which the storage server should listen at, the cache size (int), and the displacement strategy (String), i.e., either “FIFO”, “LRU”, or “LFU”, should be passed as a command line parameter as well.

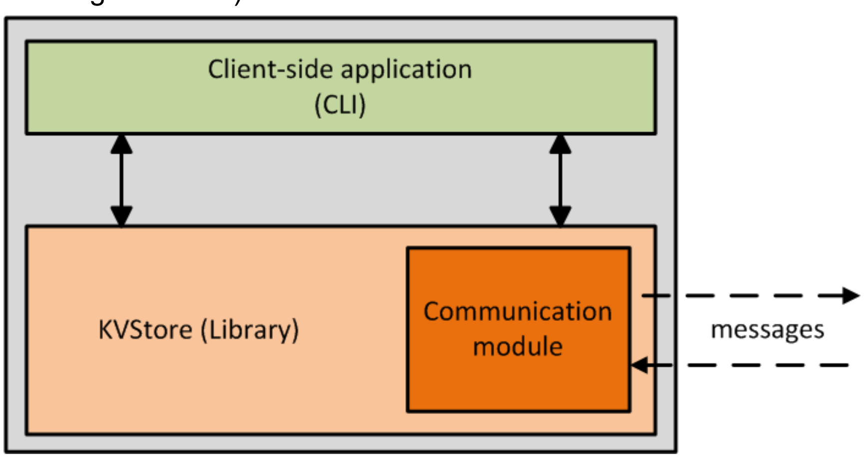
Command line parameters

In order to test your submissions, we standardize the command line parameters

-p	Sets the port of the server
-a	Which address the server should listen to, default is 127.0.0.1
-b	Bootstrap broker (Used by the client as the first broker to connect to and all other brokers to bootstrap in the next Milestones)
-d	Directory for files (Put here the files you need to persist the data, the directory is created upfront and you can rely on that it exists)
-l	Relative path of the logfile, e.g., “echo.log”.
-ll	Loglevel, e.g., INFO, ALL, …
-c	Size of the cache, e.g., 100 keys
-s	Cache displacement strategy, FIFO, LRU, LFU,
-h	Displays the help

Client program

In order to guarantee the highest possible degree of reusability, the client program should be implemented in a very modular way. Altogether it should be comprised of at least three components (cf. the figure below).



First, the **client-side application** is mainly comprised of a command line interface that manages the interaction with the user and utilizes the KVStore module. You should adjust the client program from Assignment 1 (“Echo-Client”) such that it supports the commands listed in the table below.

Second, the **KVStore module** acts as a program library for client applications in general and encapsulates the complete functionality to use a KV storage service running somewhere on the Internet. Note that this library should be independent of any particular client-side application. Thus, it has to provide a well-defined, minimal, and universally applicable interface to application programmers.

Finally, the **communication module** encapsulates the complete communication logic. Predominantly, this incorporates connection management, message sending, and message receiving. Here, you should be able to use much of the code from Assignment 1, but try to define a proper interface for this module.

Don’t forget to set up the logging facility for the client application as already defined and required in Assignment 1.

The table below specifies the commands supported by the storage server. New commands introduced in this assignment are shaded in grey.

Command	Informal description	Parameters	Shell output
`connect <address> <port>`	Tries to establish a TCP- connection to the storage server based on the given server address and the port number of the storage service.	**address**: Hostname or IP address of the storage server. **port**: The port of the storage service on the respective server.	**server reply**: Once the connection is established, the client program should give a status message to the user. (*Note, if the connection establishment failed, the client application should provide a useful error message to the user*);
`disconnect`	Tries to disconnect from the connected server.	-	**status report**: Once the client got disconnected from the server, it should provide a suitable notification to the user. (*Note that the connection might also be lost due to a connection error or a breakdown of the server*);

`put <key> <value>`	**Inserts** a key-value pair into the storage server data structures. **Updates** (overwrites) the current value with the given value if the server already contains the specified key. **Deletes** the entry for the given key if <value> equals null.	**key**: arbitrary String (max length 20 Bytes) **value**: arbitrary String (max. length 120 Kbyte)	**status message**: provides a notification if the put-operation was successful (SUCCESS) or not (ERROR)
`get <key>`	Retrieves the value for the given key from the storage server.	**key**: the key that indexes the desired value (max length 20 Bytes)	**value**: the value that is indexed by the given key if present at the storage server, or an error message if the value for the given key is not present.
`logLevel <level>`	Sets the logger to the specified log level	**level**: One of the following log4j log levels: (ALL | DEBUG | INFO | WARN | ERROR | FATAL | OFF)	**status message**: Print out current log status.
`help`		-	**help text**: Shows the intended usage of the client application and describes its set of commands.
`quit`	Tears down the active connection to the server and exits the program.	-	**status report**: Notifies the user about the imminent program shutdown.
`<anything else>`	Any unrecognized input in the context of this application.	`<any>`	**error message**: Unknown command; print the help text.

The client should log relevant information to its own file (logs/client.log). As in Assignment 1, logging should be dynamically controllable (ALL | DEBUG | INFO | WARN | ERROR | FATAL | OFF) by the command `logLevel`. (see table above).

Graceful failure handling

A vital prerequisite for the client is its ability to handle failures gracefully. Please make sure that your program is robust to any kind of wrong or unintended user input (e.g., wrong/unknown commands, number and format of parameters, etc.). In addition to that, consider problems that might occur in communication, i.e., handle exceptions properly and watch out for connection failure and (controlled) shutdown of the storage server. In addition to error messages, also print out status messages to the shell, if it makes sense (e.g. if the client is connected or disconnected from the server.)

Server feedback for requests

In the table below you will find a set of interaction schemes that your client-server application should at least support.

Request	**Result at server side**	**Consequence**
put <key> <value>	tuple successfully inserted	send acknowledgement to client: put_success <key>
put <key> <value>	unable to insert tuple	send error message to client: put_error <key>
get <key>	tuple found	get_success <key> <value> to client.
get <key>	tuple not found	send error message to client: get_error <key>
any	message format unknown, message size exceeded, etc.	send error message to client: failed <error description>

General considerations

- Document your program properly using JavaDoc comments.
- Stick to the common Java coding conventions ([Link]).
- Pay attention to good program design (e.g., decoupling of UI and program logic.)

Testing with Junit5 and Mockito

1. Create connection / disconnect
2. Set value
3. Get value
4. Update value (set with existing key)
5. Get non-existing value (error message)
6. Test client library methods
7. Test individual components using Mocking

Suggested development plan

First of all, review the project stub from GitLab, and update your project structure accordingly. The stub already contains the necessary libraries, the interfaces and several example JUnit test cases. You are free to change the project architecture as well as any files in the project in any way that fits your requirements. Do not change the communication protocol and the command line arguments.

Deliverables & Code submission

You must commit and push all the deliverables to the GitLab by the deadline (Change the remote URL of your Git repository to your Milestone 2 Git repository on GitLab by executing "git remote set-url origin [...]/milestone2/grX.git"). 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 14,* without any further interference and provide the specified functionality.

We test the following way:
First, we test Blackbox the whole implementation in an automatic way

1. We run "mvn package" in the root of the project. This also executes the tests.
2. We start the server "target/kv-server.jar" with the specified commands (e.g., directory, port, listenaddress, ...)
3. We start our own command line client, connect it to the server using the listenaddress and port
4. We execute several commands put, get, delete and check if we get the correct results

Second, we look at the individual implementation manually.

Additional resources

- Cache displacement strategy: https://developpaper.com/three-cache-elimination-algorithms-lfu-lru-fifo/
- JUnit: http://www.junit.org/
- MS2 Base: Github