Vincente Campisi
Dennis La
Robert Fritze
Cloud Computing WS2017/18

Assignment 4 - Cloud Storage Systems

Introduction:

For this assignment, we have created a distributed hash-based storage structure. It supports permanent key-value pair storage through a dictionary-based API. We used the Kotlin and Java programming languages, Gradle build tool, Spring framework, Github, Gitlab and TravisCI repository and continuous integration tooling, and the Fuel HTTP library along with Azure cloud services to implement and deploy our solution.

Problem description:

We want to implement a service that allows the user to permanently store, search and retrieve key-value pairs through a dictionary-based API and hash-based distributed storage system.

Design concerns:

The service should
x) Consist of several interacting components
x) Be distributed
x) Support a defined API
x) Offer permanent storage
x) Be accessible through a client
x) The offered client should automatically perform a set of predefined tasks
x) Log interesting/relevant activity

The project has the following conceptual modules:

1) Client
2) Web server
3) Storage Units (Remote & local)

Client refers to the project named “cloudstorage”, hosted in the master branch of the group2 repository, within the “CloudStorageFinal” folder. The client project is organized into three packages - app, core and http.

The app package holds the automated and manual client applications. Once started, ClientApplication.kt will automatically execute the tasks outline in the project

description in order and produce the associated log file. The core package holds the Client, FileParser and DictionaryAPI classes, responsible for handling user input, lexing and parsing input data files and communicating with the web server, respectively. The rest package holds the request and response handling classes, responsible for constructing and sending requests and handling server responses.

Web server refers to the project named “webserver”, hosted in the same CloudStorageFinal folder and repository location. The backend implementation for Storage Units is in the “backend” package in the web server project.

Our application architecture adheres to the standard Client-Server paradigm. The client sends automated requests, which the web server delegates to the storage units to be carried out. Status updates regarding the data set and the performed operations are relayed back to the client, where all information is logged into a log file named CloudStorageGroup2 in the project root directory. The client requests are in accordance with the project specification, which also determined the format and content of the log messages.

From a more practical perspective, it is possible to view our storage system as having the following elements:

1) Automated Client
2) Dictionary-based API
3) Web server
4) Storage units manager
5) Individual storage unit

In the following, the elements and their implementations will be described:

1) **Automated Client**

The automated client parses an input file and invokes various storage operations through the web server, while outputting all relevant information to a log file. It transitively communicates with the web server through (REST) HTTP endpoints and completes the defined tasks sequentially, before terminating. It utilizes the underlying DictionaryAPI to invoke the necessary data operations.

When the ClientApplication is executed, the runTasks function is invoked, which calls the other tasks in sequential order. The ClientApplication terminates once all steps are complete.

First, the given input file is parsed and an insert request is made for each of the entries. Next, three entries are chosen at random and three corresponding search requests are made. Two entries are randomly chosen and a delete request is made for each. The last task randomly chooses a start and end key, then executes a range query using this

information. Successes and failures are recorded in the log output and failed requests will not halt the overall execution. The only exception to this is if no input file could be successfully parsed.

The input file contains the key-value pairs to be stored in the cloud storage service. Each entry exists on a new line and the integer key and string value data are separated by a colon character ":". The provided test file, testEntries.txt contains 20 key-value pairs and is located in the project resources directory.

To parse the file, the FileParser attempts to create a File object of the given path using the java class loader. It splits the overall document into lines and splits each line using the colon separator. This eventually returns a string map of all parsed entries. If no valid file with the provided name exists, a FileNotFoundException is thrown and the application terminates abruptly, because the other steps are meaningless without valid input data.

Once the file has been parsed, insert requests are made for each element of the data set. After each request and the respective result has been logged, an additional request is made to retrieve the current status of all elements and storage units in the system.

The next task requires three elements to be selected at random and searched for within the stored data set. Once three element keys have been chosen, three requests are made to the storage service - one for each. To perform the deletion task, two keys are selected at random and two deletion requests are performed.

The final task requires the execution of a range query. The client randomly selects two keys and sends one range request with both. It receives a message outlining all stored elements with keys ordered between the two given ones. As we use an integer value as the key value, this corresponds to all key-value pairs with keys existing within the numerical range between the provided keys.

2) Dictionary-based API

This component consists of the DictionaryAPI class and the classes RequestHandler and ResponseHandler (from the rest package). The DictionaryAPI is responsible for calling the necessary functions for building, sending and receiving the request and response for an invoked API function. Its overall purpose is delegation.

The respective function of the RequestHandler is called, generating the appropriate response. This object is then passed to the ResponseHandler, which sends it and receives a response. The response is then packaged into a data class and passed back to the client. The data class is used so that the client can easily interpret the result.

3) Web Server

The web server is a Spring boot application which exposes a REST endpoint, therefore, requests are received and responses are sent to the client through HTTP. Web servers (e.g. Apache) normally are configured to run in a multiprocess and/or multithreaded mode, therefore, each request (up to a predefined limit) will be assigned to a new thread.

The web server contains the backend-related classes, from which the main manager class is registered as a Spring component. This is done through the use of the Spring "@Component" annotation in the above class, as well as the "@Autowired" annotation where the class is instantiated within the controller.

The service as a whole, consists of only one controller, with methods mapped to the respective service endpoints. Input data, such as keys, are sent as path variables and automatically injected into the invoked methods by use of method annotations.

Once an operation has been invoked, the associated method of the storage unit manager instance will be called. The web service receives either a CloudLogger object or a List<String> object, after such invocations, depending on the operation.

4) Storage Units Manager

The service consists of one class that is used as gatekeeper. It implements the methods required in the assignment for the required automated tasks. These consist of insert, search, delete, range as well as a few others such as list all entries from all buckets, list all from one bucket and determine in which bucket the data should be stored.

This class holds an array of a predefined number of buckets, each one responsible for a subset of the data. From the key, which can be chosen arbitrarily, a 32-bit hash-value can be computed. In our case, the key is an integer value. With a simple modulo-operation the bucket can be selected. The list with the hash-buckets is read only – therefore no synchronization is needed.

It uses a CloudStorageLogger object to record information regarding the success and details of an attempted operation. This is crucial in propagating the correct runtime information back to the client for logging purposes.

5) Individual Storage Unit

The buckets are classes that work as gatekeepers for the information stored in the bucket. For the buckets we have implemented different approaches:

In one approach, all data stored in a bucket is stored in one single file. This way there is much data transfer from and to the hard disk. If files become large, they might not be stored in consecutive units on the disc. This will make data transfer even slower.

The second implementation uses one file per key. Each bucket stores a map that is associates every key with a filename. The Map <key,filename> has read/write access, and therefore must be locked with a semaphore. Simple synchronization with the synchronized statements on data structures is not sufficient as long as complex operations of more than one instructions are involved.

Synchronization doesn‘t have to be implemented with semaphores: monitors can also be used of course. The Java Map data structure allows rather fast access to its members in constant time, whereas a linked list is too slow. An array is not flexible enough, although there are also expandable array-like data structures available in java.

During any operation on a single bucket, the access to this bucket is locked. Subsequent requests to the same bucket must wait until the semaphore is released. Subsequent requests to other buckets can be processed in parallel and do not block one another.

Backend and Storage (Further Info)

The backend consists of six classes:

- MyKeyValue,
- AzureStorage,
- CloudStorageLogger,
- CloudStorageBucket,
- CloudStorageHashDirectory and
- CloudStorageManager

The following contains a short description of each:

MyKeyValue.class
This class is essentially a data class which holds the key and the value of each key/value pair. By doing this, it is easier and clearer to pass the key/value pair through the methods and classes. Besides the getter and setter methods, it also overrides a toString method to assist with the log messages.

AzureStorage.class
The AzureStorage class is defined as a Singleton. Further, it is used to create the connection - with the corresponding access keys - to the storage account on the Azure platform. Further information about storage, see the Storage section.

This class also implements the methods for uploading and downloading the files from on and from Azure.

CloudStorageLogger.class
The CloudStorageLogger consists only of one boolean variable and one String variable. It functions as a wrapper class for these values and is passed as a return object to the web server.

Each time the server inserts, delete or search for a key/value pair, it will get a CloudStorageLogger object back, which contains the status of the operation. If it succeeded, the boolean value will be true, false otherwise. The message provides more details regarding the operation execution; for example, which bucket was accessed for retrieving data.

CloudStorageBucket.class
This class is implemented as a bucket where the key/value pairs are inserted, deleted and searched. The CloudStorageBucket consists of a hash map in which the key/value pairs are stored.

Besides the insert, delete and search method, it also has a getEntries method, which returns all entries in the hash map. There is also a write and read method, for when the files are generated, if they have not existed yet. In these files, the hash map is written to or read from of it.

Furthermore, all these methods are thread safe. A semaphore locks the method when in use and prevents any concurrent access attempts.

CloudStorageHashDirectory.class
This class is defined as a Singleton for this project and has a static List size of CloudStorageBucket.

For this project, it functions as an intermediate object between the CloudStorageManager and the CloudStorageBucket in which the key/value pair or only the key are passed to the bucket.

CloudStorageManager.class
The CloudStorageManager class is also defined as a Singleton. This ensures that there is only one entity that can execute the insert, delete and search methods and maintain critical consistency.

When this class is instantiated it gets an instance of CloudStorageHashDirectory which will already have a List of CloudStorageBucket. For each insert, delete and search call, the CloudStorageManager already calculates the hash value, to know to which bucket it should pass the key or the key/value pair as a MyKeyValue object. The hash function in use is the modulo operator.

And for each of the above-named methods, it will return a CloudStorageLogger with the status of the operation as a Boolean and a corresponding message as String. For the range query, it will return a List of String of the wanted key/value pairs within the given range.

Storage

We are storing the buckets as .txt files on Microsoft Azure, therefore, we are using the Storage Account in which we used the file service. Within this file service we created a file share named **cca4buckets.** In this file share we are storing the buckets, but they are also stored locally on the web server.

Every time the insert, delete and search methods are called in the CloudStorageBucket class the files are downloaded, where the locally stored files are overridden, and uploaded after the new key/value pairs are written to the file.

Our implementation with respect to the CAP-theorem:

Our implementation asserts strict consistency: A read after a write will always issue the new value. If the requests arrive simultaneously, the implementation of the semaphore will determine whether the old or the new value is issued. In the outlook section a possible improvement is discussed.

We have a single point of access, one web server. Multiple users can read and write concurrently only if (1) they are using different buckets and (2) there is a free thread available on the server. Availability becomes better when there are more buckets and server cores, but it is clearly bounded.

Our implementation is multithreaded but does not scale on more than one node.

For some remarks on how our system could be expanded please see the following section.

Outlook:

Here, several suggestions and comments regarding possible improvements are discussed.

1. The storage could be split up in a peer-to-peer system. Every node would have a web server as access point for clients and each would hold a part of the data. Every peer would be linked to one or more other peers. A function that uses a key as input could output the peer responsible for the data, similar to our current setup using

storage buckets.

The request would then be forwarded to the node responsible for that specific data. This would allow better scalability. Routing could be used to forward a request from one peer to another. If information is stored in only one peer, this system would be easily scalable (number of contemporarily processed requests and storage).

2. Backups of the data could be stored either in peers or in a second server associated with a peer. This would provide fault tolerance, but it has the disadvantage of requiring overhead in order to maintain data consistency. Multiple requests would easily cause different data sets to exist in different states, causing one version to be “outdated”.

Amazon S3, for example, provides for almost all operation only eventual consistency. Only for certain write operations of completely new data does S3 assert read after write consistency. Such conventions must be considered when implementing data replication.

3. The Range-operation could be parallelized. This would query the required elements from each bucket in parallel and prevent larger wait times on sequential searches.

4. Some operations, such as search, need only read-only access to the map. Therefore, they can be done in parallel using standard concurrency-based constructs. Java provides ReadWrite-Semaphores, which allow concurrent read-only and mutual exclusive read/write access (according to semaphore required).

It is also possible to change the behavior of the RW-Semaphore: The RW-semaphore implemented by Java will always give read-processes precedence. As long as there are readers, the information will not get updated. It is possible to combine simple semaphores in order to build a RW-semaphore that prefers the writers. This is especially helpful in all cases where information is read often but seldom written. This special type of semaphore has to be implemented by hand, as Java doesn‘t provide an implementation for a combination of several simple semaphores.

5. If the information stored has a well-defined structure and length and is not too long, a database could be used instead of a disk.

6. Data could be encrypted to assert confidentiality. This would prevent unauthorized access through intercepted requests and responses.

Installation and Setup:

The project consists of two individual projects: the client which is called “cloudcomputing” and the server which is called “webserver”. The client can be run locally from the IDE, whereas the server can be locally run but is already hosted on Azure App Service.

The URL for the web service endpoint can be configured by providing the client with the appropriate URL. The default is localhost:8080, but the correct Azure URL has been passed as a parameter in the current source code revision.

Each of the two projects is a gradle-based IntelliJ project and should be imported as such. We have taken steps to ensure that the web server is actively running on an Azure App Service instance until after the grading period.

A possible way to set up the project would be the following:

- Clone the repository
- Open project CloudStorageClient with IntelliJ
- Open project webserver with IntelliJ
- The Import Project from Gradle window will pop up
- Tick the auto-import
- Click on ok
- Wait for indexing and refreshing to complete
- Run webserver (if local, start the Spring boot application)
- Run the ClientApplication
  - The web server URL can be updated by providing the correct one as a Runtime Configuration parameter of the ClientApplication

You should see the relevant log data output to the runtime console within the IDE, as well as written to the log file in the client project root directory.

Log Output

The following is the log output given after running the ClientApplication using the standard test entries input file.

22:57:22.049 [main] INFO core.DictionaryAPI - Starting core.DictionaryAPI
22:57:22.053 [main] INFO core.Client - Starting core.Client
22:57:22.054 [main] INFO core.Client -
22:57:22.055 [main] INFO core.Client - Attempting to initialize the cloud storage service...
22:57:24.728 [main] INFO core.Client - Successfully initialized service
22:57:24.766 [main] INFO core.Client - Bucket 0, Bucket 1, Bucket 2, Bucket 3
22:57:24.767 [main] INFO core.Client -

22:57:24.768 [main] INFO core.Client - Attempting to read from file and store values...
22:57:24.768 [main] INFO core.Client - Parsing file: testEntries.txt
22:57:24.776 [main] INFO core.Client - File parsing complete.
22:57:24.776 [main] INFO core.Client - Starting insert requests...
22:57:24.776 [main] INFO core.Client - Executing insert request for key: 3 and value: toast
22:57:25.853 [main] INFO core.Client - - Successfully inserted - Key: 3 with value: toast will be stored in bucket 3
22:57:25.853 [main] INFO core.Client - Executing insert request for key: 9 and value: newton
22:57:26.632 [main] INFO core.Client - - Successfully inserted - Key: 9 with value: newton will be stored in bucket 1
22:57:26.632 [main] INFO core.Client - Executing insert request for key: 17 and value: dressing
22:57:27.482 [main] INFO core.Client - - Successfully inserted - Key: 17 with value: dressing will be stored in bucket 1
22:57:27.482 [main] INFO core.Client - Executing insert request for key: 7 and value: custard
22:57:28.303 [main] INFO core.Client - - Successfully inserted - Key: 7 with value: custard will be stored in bucket 3
22:57:28.303 [main] INFO core.Client - Executing insert request for key: 8 and value: salad
22:57:29.122 [main] INFO core.Client - - Successfully inserted - Key: 8 with value: salad will be stored in bucket 0
22:57:29.123 [main] INFO core.Client - Executing insert request for key: 15 and value: oil
22:57:29.942 [main] INFO core.Client - - Successfully inserted - Key: 15 with value: oil will be stored in bucket 3
22:57:29.942 [main] INFO core.Client - Executing insert request for key: 2 and value: jam
22:57:30.823 [main] INFO core.Client - - Successfully inserted - Key: 2 with value: jam will be stored in bucket 2
22:57:30.823 [main] INFO core.Client - Executing insert request for key: 11 and value: finn
22:57:31.684 [main] INFO core.Client - - Successfully inserted - Key: 11 with value: finn will be stored in bucket 3
22:57:31.684 [main] INFO core.Client - Executing insert request for key: 12 and value: meringue
22:57:32.499 [main] INFO core.Client - - Successfully inserted - Key: 12 with value: meringue will be stored in bucket 0
22:57:32.499 [main] INFO core.Client - Executing insert request for key: 4 and value: pudding
22:57:33.422 [main] INFO core.Client - - Successfully inserted - Key: 4 with value: pudding will be stored in bucket 0
22:57:33.422 [main] INFO core.Client - Executing insert request for key: 13 and value: ice

22:57:34.243 [main] INFO core.Client - - Successfully inserted - Key: 13 with value: ice will be stored in bucket 1
22:57:34.244 [main] INFO core.Client - Executing insert request for key: 14 and value: cake
22:57:35.061 [main] INFO core.Client - - Successfully inserted - Key: 14 with value: cake will be stored in bucket 2
22:57:35.061 [main] INFO core.Client - Executing insert request for key: 20 and value: salt
22:57:35.909 [main] INFO core.Client - - Successfully inserted - Key: 20 with value: salt will be stored in bucket 0
22:57:35.909 [main] INFO core.Client - Executing insert request for key: 1 and value: pie
22:57:36.670 [main] INFO core.Client - - Successfully inserted - Key: 1 with value: pie will be stored in bucket 1
22:57:36.670 [main] INFO core.Client - Executing insert request for key: 19 and value: sorbet
22:57:37.557 [main] INFO core.Client - - Successfully inserted - Key: 19 with value: sorbet will be stored in bucket 3
22:57:37.557 [main] INFO core.Client - Executing insert request for key: 6 and value: juice
22:57:38.441 [main] INFO core.Client - - Successfully inserted - Key: 6 with value: juice will be stored in bucket 2
22:57:38.445 [main] INFO core.Client - Executing insert request for key: 16 and value: pudding
22:57:39.467 [main] INFO core.Client - - Successfully inserted - Key: 16 with value: pudding will be stored in bucket 0
22:57:39.467 [main] INFO core.Client - Executing insert request for key: 5 and value: cheesecake
22:57:40.284 [main] INFO core.Client - - Successfully inserted - Key: 5 with value: cheesecake will be stored in bucket 1
22:57:40.284 [main] INFO core.Client - Executing insert request for key: 10 and value: jam
22:57:41.101 [main] INFO core.Client - - Successfully inserted - Key: 10 with value: jam will be stored in bucket 2
22:57:41.102 [main] INFO core.Client - Executing insert request for key: 18 and value: pie
22:57:41.931 [main] INFO core.Client - - Successfully inserted - Key: 18 with value: pie will be stored in bucket 2
22:57:41.933 [main] INFO core.Client - Finished executing insert requests.
22:57:41.934 [main] INFO core.Client - Retrieving current state of all files...
22:57:42.699 [main] INFO core.Client - Bucket 0, 16 : pudding, 4 : pudding, 20 : salt, 8 : salad, 12 : meringue, Bucket 1, 17 : dressing, 1 : pie, 5 : cheesecake, 9 : newton, 13 : ice, Bucket 2, 2 : jam, 18 : pie, 6 : juice, 10 : jam, 14 : cake, Bucket 3, 3 : toast, 19 : sorbet, 7 : custard, 11 : finn, 15 : oil
22:57:42.700 [main] INFO core.Client -
22:57:42.701 [main] INFO core.Client - Randomly selecting and reading three values...

22:57:42.702 [main] INFO core.Client - Selecting three entries at random
22:57:42.704 [main] INFO core.Client - First random key: 8
22:57:42.705 [main] INFO core.Client - Second random key: 19
22:57:42.706 [main] INFO core.Client - Third random key: 15
22:57:42.706 [main] INFO core.Client - Making search requests...
22:57:42.707 [main] INFO core.Client - Searching for key: 8
22:57:43.253 [main] INFO core.Client - - Found key - Data: salad with key: 8 found in bucket 0
22:57:43.253 [main] INFO core.Client - Searching for key: 19
22:57:43.762 [main] INFO core.Client - - Found key - Data: sorbet with key: 19 found in bucket 3
22:57:43.762 [main] INFO core.Client - Searching for key: 15
22:57:44.277 [main] INFO core.Client - - Found key - Data: oil with key: 15 found in bucket 3
22:57:44.279 [main] INFO core.Client - Finished executing search requests.
22:57:44.280 [main] INFO core.Client -
22:57:44.282 [main] INFO core.Client - Deleting two entries at random...
22:57:44.282 [main] INFO core.Client - Selecting two entries at random
22:57:44.283 [main] INFO core.Client - First random key: 5
22:57:44.284 [main] INFO core.Client - Second random key: 15
22:57:44.284 [main] INFO core.Client - Making delete requests...
22:57:44.285 [main] INFO core.Client - Deleting entry for key: 5
22:57:45.097 [main] INFO core.Client - - Successfully deleted - Data with key: 5 deleted from bucket 1
22:57:45.097 [main] INFO core.Client - Deleting entry for key: 15
22:57:45.915 [main] INFO core.Client - - Successfully deleted - Data with key: 15 deleted from bucket 3
22:57:45.917 [main] INFO core.Client - Finished executing delete requests.
22:57:45.918 [main] INFO core.Client -
22:57:45.920 [main] INFO core.Client - Querying range for two entries at random...
22:57:45.920 [main] INFO core.Client - Selecting two entries at random
22:57:45.921 [main] INFO core.Client - First random key: 7
22:57:45.922 [main] INFO core.Client - Second random key: 20
22:57:45.922 [main] INFO core.Client - Making range request for key 1: 7 and key 2: 20
22:57:54.418 [main] INFO core.Client - - Successfully retrieved range for key 1: 7 and key 2: 20" - Data: custard with key: 7 found in bucket 3, Data: salad with key: 8 found in bucket 0, Data: newton with key: 9 found in bucket 1, Data: jam with key: 10 found in bucket 2, Data: finn with key: 11 found in bucket 3, Data: meringue with key: 12 found in bucket 0, Data: ice with key: 13 found in bucket 1, Data: cake with key: 14 found in bucket 2, Data with key: 15 not found., Data: pudding with key: 16 found in bucket 0, Data: dressing with key: 17 found in bucket 1, Data: pie with key: 18 found in bucket 2, Data: sorbet with key: 19 found in bucket 3, Data: salt with key: 20 found in bucket 0
22:57:54.420 [main] INFO core.Client - Finished executing range request.
22:57:54.421 [main] INFO core.Client -