Milestone 2: Scalable Storage Service

DEADLINE: 26th Feb, 2021

Cloud Databases

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. The recently proposed 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.

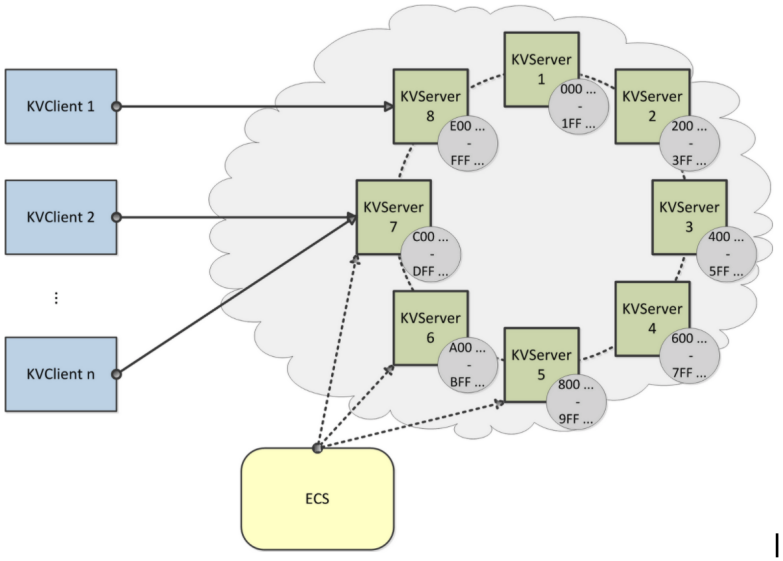
The objective of this milestone is to **extend the storage server of Milestone 1** into a dynamically controllable, scalable **storage service** (cf. the figure below). Data records (i.e., **key-value pairs**, a.k.a., **tuples**) are distributed over a number of storage servers by exploiting the capabilities of **consistent hashing**. A single storage server is only responsible for a subset of the whole key 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 (e.g., the command line shell.) In order to forward requests to the storage server that is responsible for a particular tuple, the client library maintains metadata about the current state of the storage service. Metadata 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 server, the

storage server will answer with an appropriate error and the most recent version of the metadata. After updating its metadata, the client will 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 ring (i.e., the ring, as depicted in the below figure.) The position implicitly defines a subrange within the complete hash range.

The storage servers are monitored and controlled by an **External Configuration Service (ECS)**. The ECS we specify is a simplification; in practice additional services like Apache ZooKeeper, deployed over a number of nodes to achieve availability and reliability, supports the responsibilities of ECS. By means of the configuration service, an administrator is able to initialize and control the storage service (i.e., add/remove storage servers and invoke reconciliation of metadata at affected storage servers).



Learning objectives

With this milestone, we pursue the following learning objectives:

- Understand & build an incrementally scalable storage service based on the software artifacts developed in the previous milestone: Independent set of storage servers to provide horizontal scalability based on consistent hashing

- Understand how to coordinate the storage service with an external configuration service (ECS), ZooKeeper (https://zookeeper.apache.org/) . Our specification ignores the availability and reliability mechanisms provided out of the box in ZooKeeper, which grossly simplifies the ECS (an oversimplification not acceptable in practice); we use the standalone version of ZooKeeper (ZK) (i.e., a single running ZK instance.)
- Learn to access the storage service from an application via a library that provides a client-level abstraction for interacting with a storage server (similar to the client library developed in the earlier milestone for a single storage server)
- Learn how to invoke and control applications on remote servers
- Understand the concept and advantages of consistent hashing
- Get exposed to caching in order to reduce network traffic
- Continue to practice unit testing and apply regression testing as well as logging
- Conduct performance measurements at moderate scale

Detailed milestone description

Provided infrastructure

Together with this handout, we provide you with a project stub 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.

Assigned development tasks and deliverables

We recommend that on your first read of this milestone specification, you only glance at the deliverables below, then, read and understand the rest of the specification, before coming back and reading this deliverable section in more detail.

Develop or extend the following key components based on your components from the previous milestone.

External configuration service (ECS)

- The ECS is bootstrapped with a configuration file that specifies the IP addresses and ports of *n* storage servers under its control. These storage servers provide the KVServer interface available at the specified IP and port. The storage servers are launched by the ECS through SSH calls. **Note that you may use the same IP address, simply varying the ports for each one of the *n* storage servers** (a simplification).

- Remotely launch a storage service comprised of *m* storage servers (*m* ≤ *n*)
  1. Compute key-range partitioning for initial setup (initial metadata)
  2. Assemble metadata
  3. Launch storage servers with initial metadata
- Incrementally add a storage server
  1. Compute key-range position of the storage server within the service
  2. Launch storage server at a given IP:Port with updated metadata
  3. Inform neighbour to initiate key-value hand-off
  4. Update metadata of affected storage servers
  5. Read requests are always served
- Remove a storage server
  1. Re-compute key-range position for affected storage servers
  2. Inform neighbour to initiate key-value hand-off
  3. Update metadata of affected storage servers
  4. Shutdown storage server at the given IP:Port
  5. Read requests are always server

Client library (KVStore)

- Cache metadata of storage service. (Note: this metadata might not be the most recent)
- Route requests to the storage server that coordinates the respective key-range
- Metadata updating might be required which is initiated by the storage server if the client library, that caches the metadata, contacted a wrong storage server (i.e., the request could not be served by the storage server identified through the currently cached metadata) due to stale metadata
- Update metadata and retry the request

Storage server (KVServer)

- The KVServer process is launched with an SSH call by the ECS but does not start serving client requests immediately. All client requests are responded to with SERVER_STOPPED messages. Messages by the ECS are processed as usual. Only the ECS is able to configure the KVServer and activate it for client interaction (i.e., serving of client requests).
- Admin interface provides functions to

1. Assign a key-range to the storage server and incorporate the storage server to the storage service (**activate** KVServer, handle client requests)
2. Hand-off data items to another server (in case of reorganizing the storage service due to added/removed storage servers)
3. Update the metadata

- Request processing
  1. All requests are processed locally
  2. 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 metadata

JUnit Tests

- Integration of the JUnit library into your project (as in Milestone 1)
- Automate the running of the given test cases
- Do regression testing, i.e., make sure that relevant Milestone 1 tests still pass
- Test functionality such as create connection/disconnect, get/put value, update value (existing key), get non-existing key (check error messages)
- **Add at least 10 test cases of your choice that cover the additional functionality and features of this milestone** (e.g., ECS, consistent hashing, metadata updates, retry operations, locks, etc.)
- Compile a short test report about all test cases, especially your own cases (submit as appendix of your design document).

Performance evaluation, data set, and metrics for testing

- Evaluate the performance of your storage service implementation. Use the Enron Email data set **(http://www.cs.cmu.edu/%7Eenron/)** to populate your storage service and run experiments. Measure latency and throughput for read and write operations in varying scenarios. Evaluated at least the following:
- Different number of clients connected to the service (e.g., 1, 5, 20, 50, 100)
- Different number of storage servers participating in the storage service (e.g., 1, 5, 10, 50, 100)
- Different KV server configurations (cache sizes, strategies)

Also, consider evaluating the process of scaling the system up and down, i.e., how long does it take in the different scenarios to add/remove 1 ... *n* KVServers (you chose *n*). Your report should contain the quantitative results (e.g., plots) and an explanation of the results.

Design Document

A design document that is no longer than 3 pages detailing your design, any design decisions you made and includes the performance evaluation and report; the documentation of your test cases is not counted among the page count.

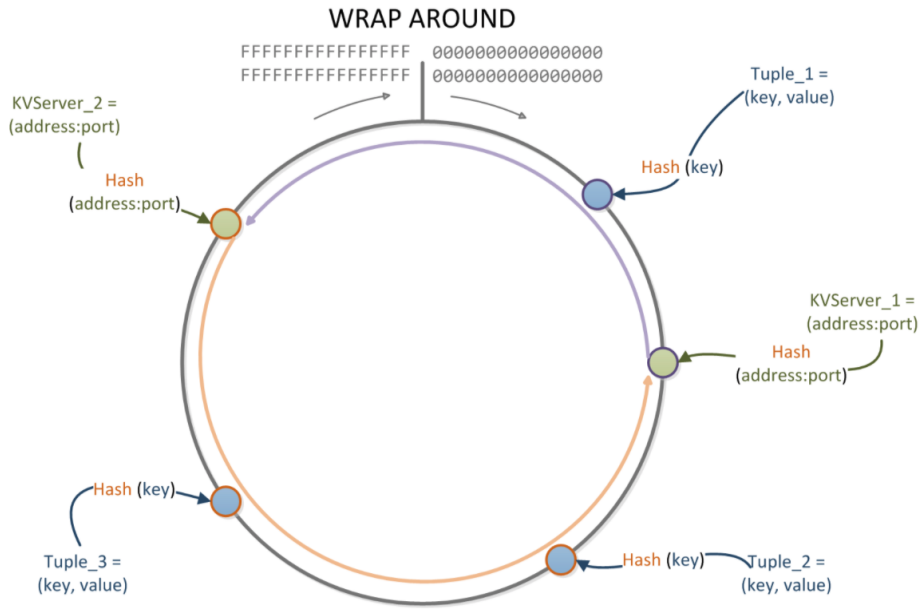
Consistent Hashing

The figure below outlines the basic ideas behind consistent hashing as it applies to this milestone (a lecture unit is also dedicated to explaining this concept). All storage servers (KVServers) are arranged clockwise in a logical hash ring (the underlying physical topology may differ.) Each position in the ring corresponds to a given 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 milestone, 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 **(http://docs.oracle.com/javase/7/docs/api/java/security/MessageDigest.html)**).

The position of a given storage server is calculated by hashing its address and port (<ip>:<port>). Similarly, the position of a (key, value)-pair tuple is determined by hashing the respective key.

As a result, both servers and tuples are assigned distinct positions within the hash 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 server 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 metadata in the context of this milestone is formed 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.

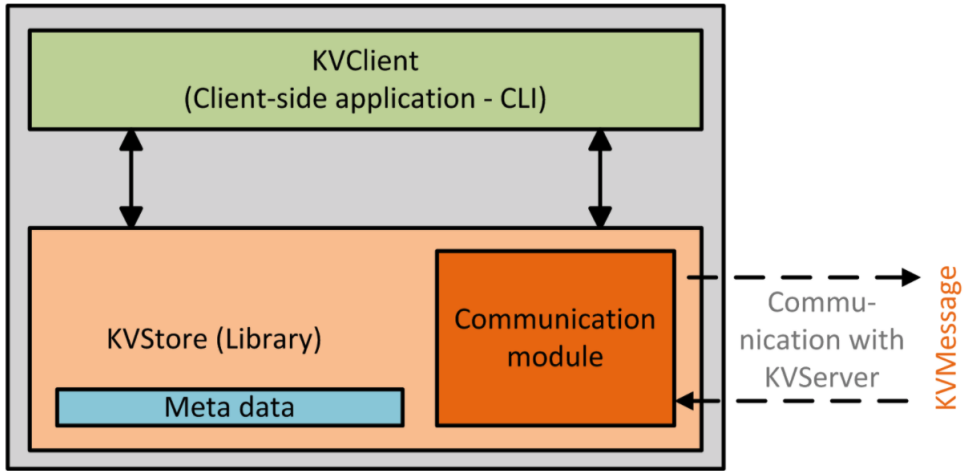


Client library (KVStore) & application (KVClient)

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, typical KV-API (cf. KVCommInterface) as in Milestone 1, 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, metadata about the storage service is maintained to identify the respective server. Initially, there is no metadata 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 client library (user) has to know at least one of the storage servers). 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 server and could not be processed. In such cases, the server would answer with a specific error message (see below) that also contains the most recent metadata. As a consequence, the library would update its metadata, determine the correct storage server, connect to it and retry the request. Retry operations are transparent to the client application.

KVStore API - KVCommInterface



KVStore API - KVCommInterface

```
/**
 * Establishes a connection to the storage service, i.e., to an arbitrary
 * instance of the storage servers that makes up the storage service.
 * @throws Exception if connection could not be established.
 */
public void connect() throws Exception;

/**
 * Disconnects the client from the storage service (i.e., the currently
 * connected server).
 */
public void disconnect();

/**
 * Inserts a data record into the storage service.
 * @param key the key that identifies the given value.
 * @param value the value that is indexed by the given key.
 * @return a message that confirms the insertion of the tuple or an
 * error.
 *
 * @throws Exception if put command cannot be executed
 * (e.g., not connected to any storage server).
 */
public KVMessage put(String key, String value) throws Exception;

/**
 * Retrieves a data record for a given key from the storage service.
 * @param key the key that identifies the record.
 * @return the value, which is indexed by the given key.
 * @throws Exception if put command cannot be executed
 *     (e.g., not connected to any storage server).
 */
public KVMessage get(String key) throws Exception;
```

NOT_RESPONSIBLE is used by the storage server to respond to requests that could not be processed by the respective storage server because the requested key is not within its range. If the storage server creates such a message, it also has to send the current metadata attached with the message. Such messages **should not** be passed back to the client application but invoke a retry operation of the client library. Define a suitable representation (e.g., a Map) for the metadata 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 servers to the respective hash-ranges. Extend your message format to incorporate this data if the storage server has to answer with NOT_RESPONSIBLE.

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 servers.

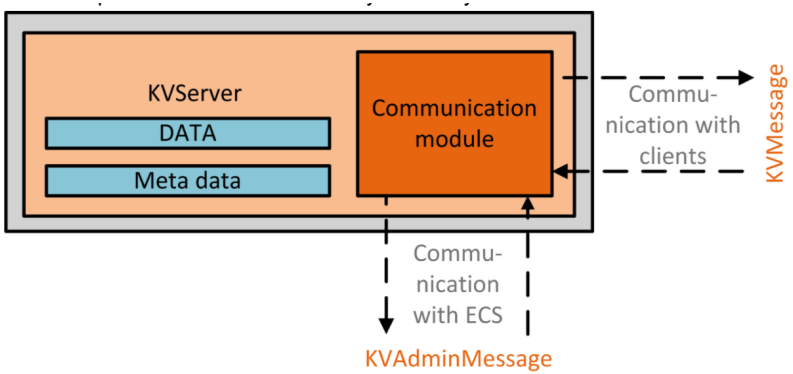
SERVER_STOPPED indicates that currently no requests are processed by the server since the whole storage service is initializing. Hence, from the client's perspective, the storage server is stopped for serving requests.

```
  /**
   * @return the key that is associated with this message,
   *          null if no key is associated.
   */
  public String getKey();

  /**
   * @return the value that is associated with this message,
   *          null if no value is associated.
   */
  public String getValue();

  /**
   * @return a status string that is used to identify request types,
   * response types and error types associated to the message.
   */
  public StatusType getStatus();
```

```
public enum StatusType {
  GET,            /* Get - request */
  GET_ERROR,      /* requested tuple (i.e. value) not found */
  GET_SUCCESS,    /* requested tuple (i.e. value) found */
  PUT,             /* Put - request */
  PUT_SUCCESS,    /* Put - request successful, tuple inserted */
  PUT_UPDATE,     /* Put - request successful, i.e., value updated */
  PUT_ERROR,      /* Put - request not successful */
  DELETE_SUCCESS, /* Delete - request successful */
  DELETE_ERROR,   /* Delete - request successful */

  SERVER_STOPPED,         /* Server is stopped, no requests are processed */
  SERVER_WRITE_LOCK,      /* Server locked for write, only get possible */
  SERVER_NOT_RESPONSIBLE  /* Request not successful, server not responsible for
key */
}
```

Storage server (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 1, this data is stored persistently on disk. A KVServer communicates with the KVStore library of client applications and processes their request (similar to Milestone 1). 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 metadata of the storage service. Note, it is necessary to hold the **complete** metadata to be able to inform clients in case of incorrectly addressed requests. Metadata updates on the KVServer are only invoked by the ECS.



In addition to the query functionality (put, get), the storage 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 suitable admin message format to pass the commands along with the parameters.

initKVServer(metadata)	Initialize the KVServer with the metadata and block it for client requests, i.e., all client requests are rejected with an SERVER_STOPPED error message; ECS requests have to be processed.
start()	Starts the KVServer, all client requests and all ECS requests are processed.
stop()	Stops the KVServer, all client requests are rejected and only ECS requests are processed.
shutDown()	Exits the KVServer application.

lockWrite()	Lock the KVServer for write operations.
unLockWrite()	Unlock the KVServer for write operations.
moveData(range, server)	Transfer a subset (range) of the KVServer's data to another KVServer (reallocation before removing this server or adding a new KVServer to the ring); send a notification to the ECS, if data transfer is completed.
update(metadata)	Update the metadata repository of this server

Once the KVServer application has been launched remotely by an SSH invocation of the ECS, it is in the stopped state. That means it is able to accept client connections but will answer with SERVER_STOPPED. Before the storage server can be started by the ECS, it has to be initialized with metadata. Therefore, the ECS connects to the respective server and sends a message that invokes initKVServer(metadata). After the server has been initialized, the ECS can start the server (call start()).

External Configuration Service (ECS)

The purpose of ECS is to deploy a storage service instance and control it (add/remove storage servers, allocate metadata among the storage servers, etc.).

The ECS is initialized with a configuration file ("ecs.config", see below) that contains a list of servers which the ECS could draw on to set up and scale the storage service.

```
server1 127.0.0.1 50000
server2 127.0.0.1 50001
server3 127.0.0.1 50002
server4 127.0.0.1 50003
server5 127.0.0.1 50004
server6 127.0.0.1 50005
server7 127.0.0.1 50006
server8 127.0.0.1 50007
```

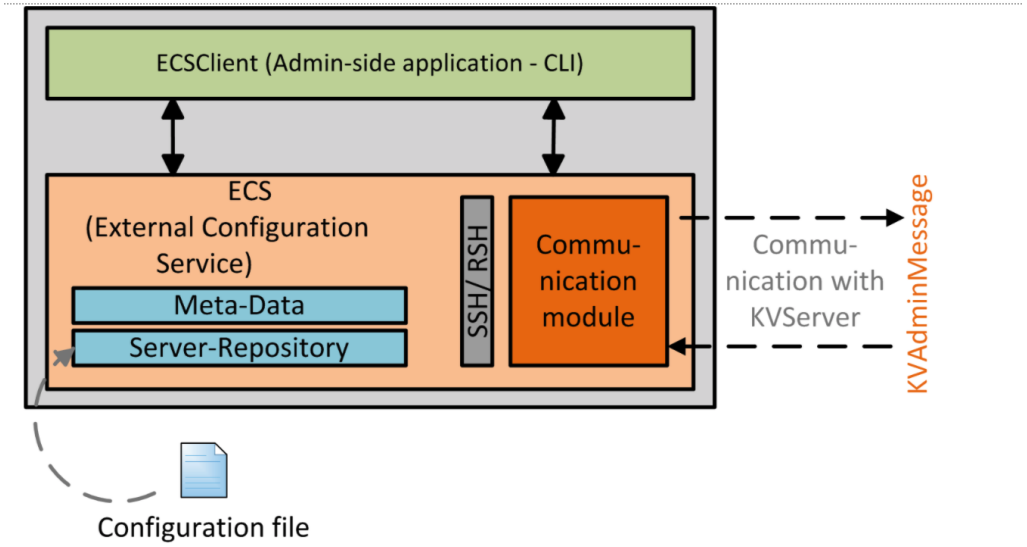
At startup, this file is parsed and has to be maintained in memory by the ECS.

java -jar ECS.jar ecs.config

Basically, the ECS provides the following interface to the ECSClient application:

addNodes (int numberOfNodes)	Randomly choose <numberOfNodes> servers from the available machines and start the KVServer by issuing an SSH call to the respective machine. This call launches the storage server. For simplicity, locate the KVServer.jar in the same directory as the ECS. All storage servers are initialized with the metadata and any persisted data, and remain in state stopped.
start()	Starts the storage service by calling start() on all KVServer instances that participate in the service.
stop()	Stops the service; all participating KVServers are stopped for processing client requests but the processes remain running.
shutDown().	Stops all server instances and exits the remote processes.
addNode()	Create a new KVServer and add it to the storage service at an arbitrary position.
removeNode(<index of server>)	Remove a server from the storage service at an arbitrary position.

The ECSClient is a simple command-line-based user interface (i.e., an admin console or shell) to control the storage service. It resembles the KVClient application which interacts with the KVStore library.



Init storage service: initService(numberOfNodes)

In order to initialize the storage service, the ECS randomly chooses the required number of machines from the server repository (i.e., from “ecs.config”) and sends an SSH call to these machines, launching an instance of KVServer. You need to use SSH on your machine and ensure that you don’t need password authentication (e.g., setup public key authentication (http://askubuntu.com/questions/46930/how-can-i-set-up-password-less-ssh-login)). In addition, the ECS has to determine the positions of each server within the ring topology to set up the service metadata. Once all server ranges are determined (i.e., the initial metadata is complete), it sends initialization messages that contain the metadata to all storage servers.

SSH call to remote server

To implement an SSH call from the machine that runs the ECS to the servers that run the KVServer, you can use a bash script that you invoke from the ECS application.

Sample bash script (script.sh):

```
ssh -n <host> nohup java -jar <path>/ms2-server.jar 50000 ERROR &
```

Call script from ECS:

```
Process proc;
String script = "script.sh";

Runtime run = Runtime.getRuntime();
try {
  proc = run.exec(script);
} catch (IOException e) {
  e.printStackTrace();
}
```

Start storage service

After all servers acknowledged the metadata reception, the ECS should be able activate the storage service by sending messages to all participating servers and calling their start()-method.

Add storage server

The steps for adding a new KVServer to the storage service by the ECS can be summarized as follows:

- If there are idle servers in the repository (i.e., in the list provided in “ecs.config”), randomly pick one of them and send an SSH call to invoke the KVServer process.
- Determine the position of the new storage server within the ring by hashing its address.
- Recalculate and update the metadata of the storage service (i.e,, the ranges for the new storage server and its successor)
- Initialize the new storage server with the updated metadata and start it.
- Set write lock (lockWrite()) on the successor server;
- Invoke the transfer of the affected data items (i.e., the range of keys that was previously handled by the successor storage server) to the new storage server. The data that is transferred should not be deleted immediately to be able to serve read requests in the

meantime at the successor while transfer is in progress (while the newly added storage server is “write-locked”.)
  - successor.moveData(range, newServer)
- When all affected data has been transferred (i.e., the successor notified the ECS)
  - Send a metadata update to all storage servers (to inform them about their new responsibilities)
  - Release the write lock on the successor server and finally remove the data items that are no longer handled by this server

Remove storage server

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

- Select one of the storage servers to be removed
- Recalculate and update the metadata of the storage service (i.e., the range for the successor storage server)
- Set the write lock on the storage server that has to be deleted.
- Send metadata update to the successor storage server (i.e., the successor is now also responsible for the range of the storage server that is to be removed)
- Invoke the transfer of the affected data items (i.e., all data of the storage server that is to be removed) to the successor. The data that is transferred should not be deleted immediately to be able to serve read requests while the transfer is in progress
  - serverToRemove.moveData(range, successor)
- When all affected data has been transferred (i.e., the storage server that is removed notified the ECS)
  - Send a metadata update to the remaining storage servers
  - Shutdown the respective storage server

ZooKeeper - Use for ECS and Implementation

ZooKeeper **(https://zookeeper.apache.org/)** is an open-source coordination service for supporting common functionalities of distributed applications. It provides simple interfaces and services to the clients such as naming, configuration management, and synchronization. It is designed for ease of use and uses a data model much like the typical file system in an operating system. Keep in mind that ZooKeeper is a production-strength “tool” with a lot of features, many of which you may not need in this milestone. You will have to familiarize yourself with ZooKeeper and design the required ECS functionality by using ZooKeeper.

Typically, in order to ensure reliability, ZooKeeper runs on multiple machines. However, for the purpose of this milestone **you should consider only running one instance** since our main purpose is to familiarize ourselves with ZooKeeper, not to run a production storage service.

Useful Links with Further Information about ZooKeeper::

ZooKeeper Getting Started:

https://zookeeper.apache.org/doc/r3.4.11/zookeeperStarted.html **(https://zookeeper.apache.org/doc/r3.4.11/zookeeperStarted.html)**

ZooKeeper Overview:

https://zookeeper.apache.org/doc/trunk/zookeeperOver.html **(https://zookeeper.apache.org/doc/trunk/zookeeperOver.html)**

ZooKeeper Programmer's Guide:

https://zookeeper.apache.org/doc/r3.4.11/zookeeperProgrammers.html
(https://zookeeper.apache.org/doc/r3.4.11/zookeeperProgrammers.html)

ZooKeeper Internals:

https://zookeeper.apache.org/doc/r3.4.11/zookeeperInternals.html **(https://zookeeper.apache.org/doc/r3.4.11/zookeeperInternals.html)**

Getting the project files

The project files for Milestone 2 are located in the **/cad2/ece419s/M2/** directory on the EECG UG machines. The Milestone 2 starter package has one archive file containing source code: *m2-stub.tgz*. The stub contains the new libraries, interfaces, and build script. To retrieve this file, make a new directory for Milestone 2 and run the following commands:

```
> cp /cad2/ece419s/M2/* ./
> tar -xzf m2-stub.tgz
```

You can either build off of this starter code, or make the necessary modifications and use your Milestone 1 code.

You can also retrieve the Enron dataset (saved as enron_mail_20150507.tar.gz) from the same directory.

Marking guidelines and marking scheme

Your code must build and execute on the UG machines Debian 9.6 with Java 1.8.0_181 without any further intervention on our part and provide the specified functionality.

Deliverables & Code submission

By the deadline, you must hand in your software artifacts that implement all the coding requirements and include all necessary libraries and the build script. See the above specified deliverables for what needs to be submitted. Submission Instructions can be found here **(https://q.utoronto.ca/courses/202854/pages/submission-instructions-and-normalized-environment)** .

Additional resources

- Log4j: http://logging.apache.org/log4j/2.x/ **(http://logging.apache.org/log4j/2.x/)**
- JUnit: http://www.junit.org/ **(http://www.junit.org/)**
- Ant build tool: http://ant.apache.org/ **(http://ant.apache.org/)**
- Consistent Hashing: **(http://tools.ietf.org/pdf/rfc20.pdf)** http://www.codeproject.com/Articles/56138/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 **(http://www8.org/w8-papers/2a-webserver/caching/paper2.html)** . 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 **(http://thor.cs.ucsb.edu/%7Eravenben/papers/coreos/kll+97.pdf)** . STOC 1997: 654-663

Document revisions

Changes to the milestone handout after posting it are tracked here.

Date	Change
3 Feb 2021	Deadline updated from 20th Feb to 26th Feb, hurray!
8 Feb 2021	Cacheing is an optional component, you might want to play around to see some performance benefits with it. However, your grade would not depend on it