Cloud Data Bases

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. 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.

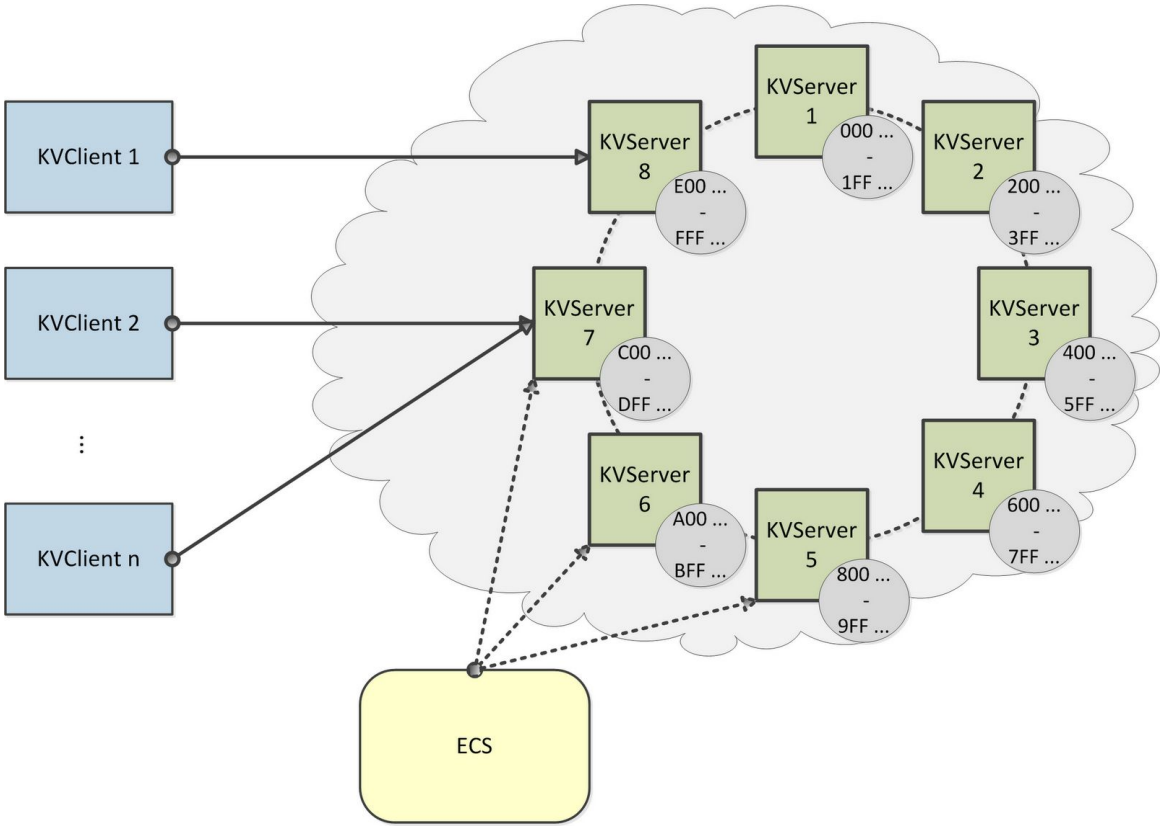
Key-value stores often relax the traditional ACID (atomicity, consistency, isolation and durability) transactional model of database management systems and offer a BASE model (basically available, soft state, and eventual consistency) to trade off performance and availability for strict consistency. 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 assignment is to extend the storage server architecture from Milestone 2 into a dynamically controllable, **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 meta-data 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).



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)
- 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
- Conduct performance measurements

Detailed assignment 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 [Link], some interfaces (see below), and the build script [Link] for building and testing the program. You are to use MD5 for all hash operations!

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)
2. 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.
    - Remotely launch a storage service comprised of m storage server nodes (m <= n)
        i. Compute key-range partitioning for initial setup (initial meta-data)
        ii. Assemble meta-data
        iii. Launch nodes with initial meta-data
    - Incrementally add a node
        i. Compute key-range position of the server within the service
        ii. Launch node at a given IP:Port with updated meta-data
        iii. Inform neighbour to initiate key-value hand-off
        iv. Update meta-data of affected storage nodes
        v. Read requests are always served
    - Remove a node
        i. Re-compute key-range position for affected storage nodes
        ii. Inform neighbour to initiate key-value hand-off
        iii. Update meta-data of affected storage nodes
        iv. Shut down node at the given IP:Port
        v. 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)

- 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

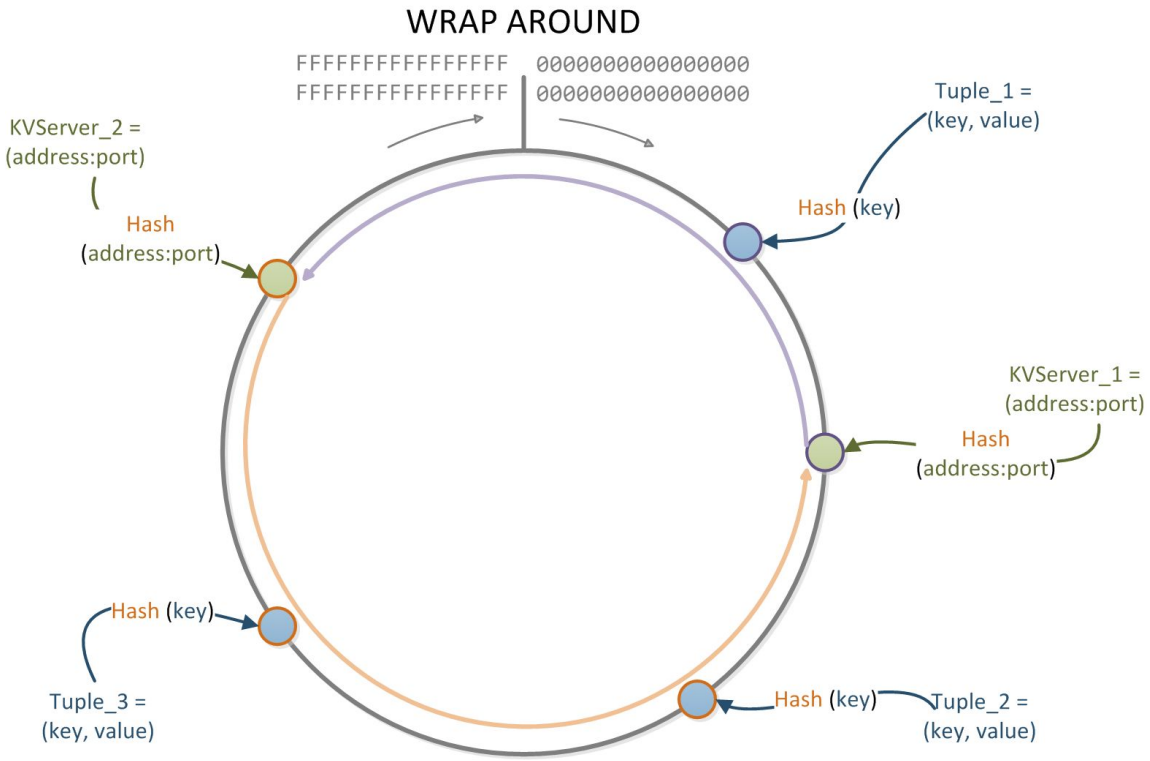
4. Storage server node (KVServer)
    - The KVServer process is launched with a SSH call by the ECS but does not start serving client requests immediately. All client requests are responded with a SERVER_STOPPED messages. Messages by the ECS are processed as usual. Only the ECS is able to configure the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

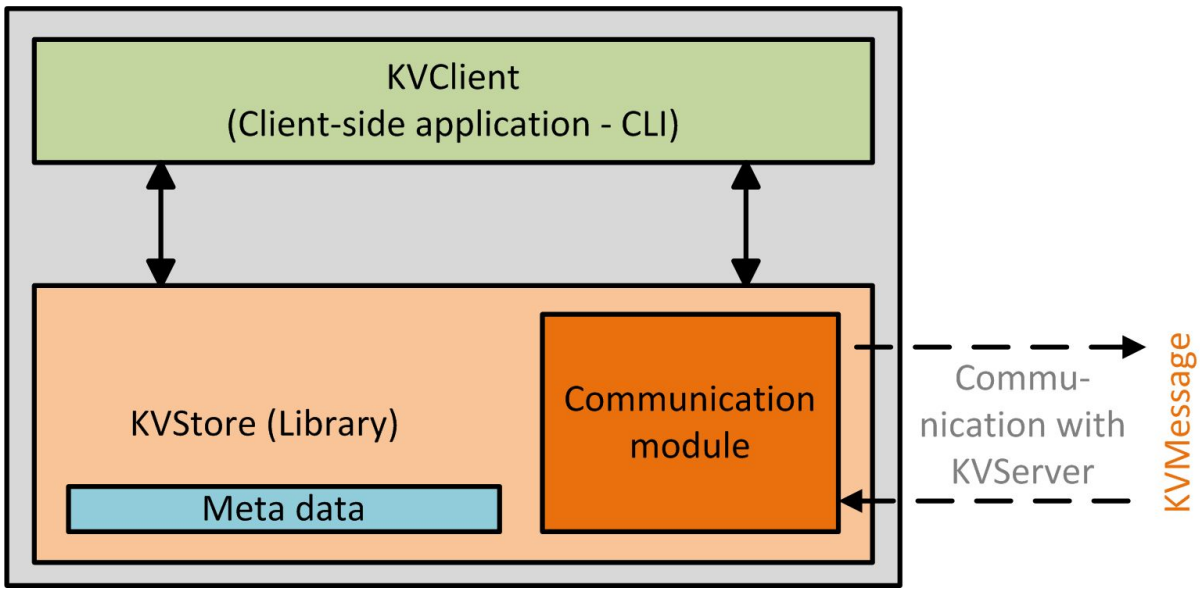


Client library (KVStore) & application (KVClient)

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, typical KV-API (cf. KVCommInterface) as in Milestone 2, 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 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 node 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 meta-data. As a consequence, the library would update its meta-data, determine the correct storage server, connect to it and retry the request. Retry operations are transparent to the client application.



KVStore API - KVCommInterface

```
  /**
   * Establishes the connection to the storage service, i.e., an arbitrary
   * instance of on 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;
```

The StatusType-enumeration contains some new flags.

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. If the server creates such a message, it also has to send the current meta-data 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 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. Extend your message format to incorporate this data if the 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 nodes.

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.

```
/**
 * @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();
```

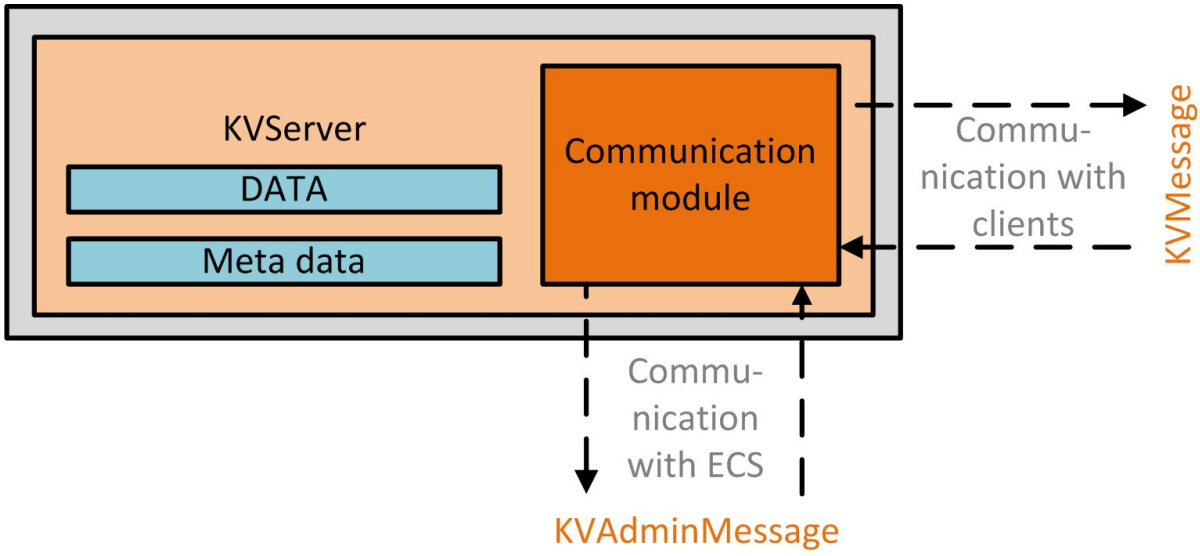
```
  /**
   * @return a status string that is used to identify request types,
   * response types and error types associated to the message.
   */
  public StatusType getStatus();
```

```
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 out, only get possible */
  SERVER_NOT_RESPONSIBLE  /* Request not successful, server not responsible for
key */
}
```

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 suitable admin message format to pass the commands along with the parameters.

initKVServer(metadata, cacheSize, displacementStrategy)	Initialize the KVServer with the meta-data, it's local cache size, and the cache displacement strategy, 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 meta-data repository of this server

Once the KVServer application has been launched remotely by the SSH call of the ECS, it is in the state stopped. That means it is able to accept client connections but will answer with SERVER_STOPPED. Before the storage node can be started by the ECS, it has to be initialized with the meta-data. Therefore, the ECS connects to the respective server and sends a message that invokes initKVServer(meta-data). 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 nodes, allocate meta-data 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 can use to build up and scale the storage service.

```
node1 127.0.0.1 50000
node2 127.0.0.1 50001
node3 127.0.0.1 50002
node4 127.0.0.1 50003
node5 127.0.0.1 50004
node6 127.0.0.1 50005
node7 127.0.0.1 50006
node8 127.0.0.1 50007
```

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

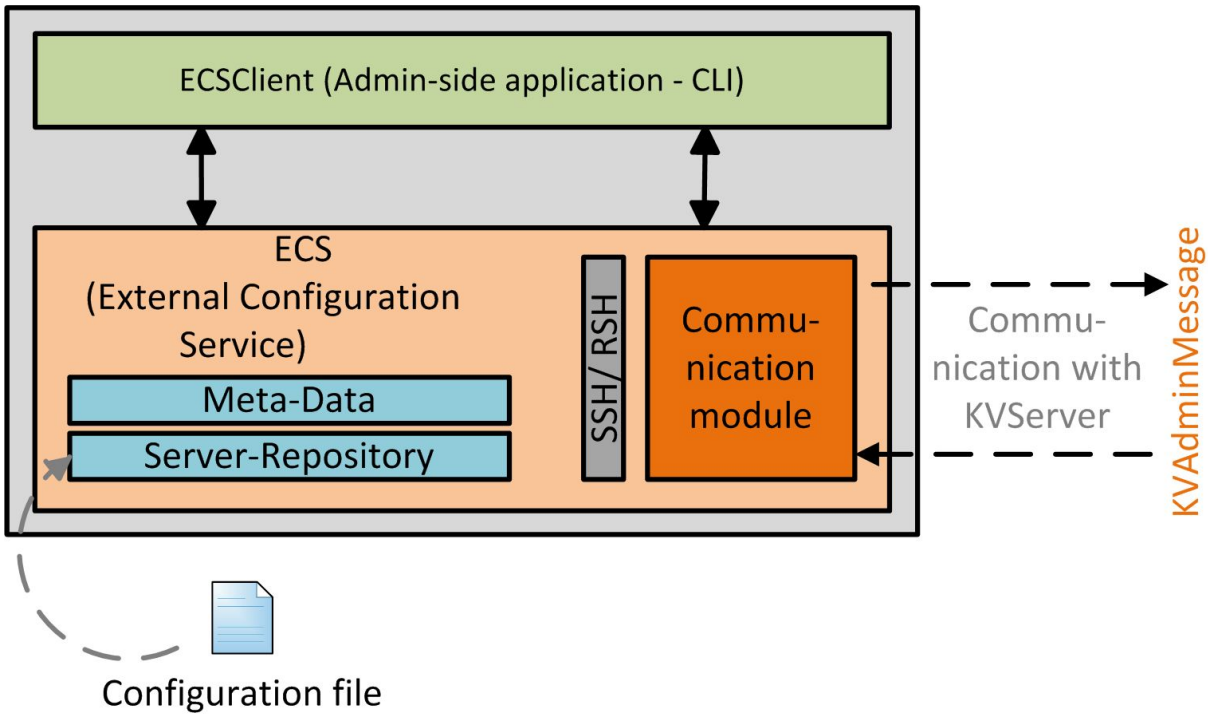
```
java -jar ECS.jar ecs.config
```

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

initService (int numberOfNodes, int cacheSize, String displacementStrategy)	Randomly choose <numberOfNodes> servers from the available machines and start the KVServer by issuing a SSH call to the respective machine. This call launches the server with the specified cache size and displacement strategy. You can assume that the KVServer.jar is located in the same directory as the ECS. All servers are initialized with the meta-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(int cacheSize, String displacementStrategy)	Create a new KVServer with the specified cache size and displacement strategy and add it to the storage service at an arbitrary position.
removeNode()	Remove a node from the storage service at an arbitrary position.

The ECSClient is a simple command-line-based user interface (i.e., admin) 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 and sends an SSH call to these machines, launching an instance of KVServer. You have to install SSH on your machine and ensure that you don't need password authentication (e.g., setup public key authentication). In addition, the ECS has to determine the positions of each server within the ring topology to setup the service meta-data. Once all server ranges are determined (i.e., the initial meta-data is complete), it sends initialization messages that contain the meta-data 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>/ms3-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 meta-data reception, the ECS should be able activate the storage service by sending messages to all participating servers and calling their start()-method.

Add storage node

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, 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 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 and start it.
- Set write lock (lockWrite()) 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
    - successor.moveData(range, newServer)
- 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:

- Randomly select one of the storage servers.
- 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
    - serverToRemove.moveData(range, successor)
- 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.
    - Shutdown the respective storage server.

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 the provided testcases, **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. 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

First of all, download the project stub [Link] and base your implementation on the provided package structure. The stub already contains the necessary libraries, the interfaces, the build

script and the JUnit test cases. You should integrate your implementation of MS2 into this project structure and try incorporate the extensions for the purpose of this milestone.

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.

Marking guidelines and marking scheme

All the code you submit must be compatible with the build scripts, interfaces and test cases that we propose for the respective milestones. In addition, your code must build and execute on lxhalle (*Ubuntu 10.04.4;current java version: 1.6.0_26)* without any further adjustments and provide the specified functionality.

Additional resources

- Log4j: http://logging.apache.org/log4j/2.x/
- JUnit: http://www.junit.org/
- Ant build tool: http://ant.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

Document revisions

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

Date	**Change**
None	