KVStore Information and Overview

Initially, you will build a key-value storage system complete with code for a central key-value server (with caching), client-side libraries, and all necessary networking components.

Multiple clients will be communicating with a single key-value server in a given messaging format (`KVMessage`) using a `KVClient`. Communication between the clients and the server will take place over the network through sockets (`SocketServer` and `ServerClientHandler`). The `KVServer` uses a `ThreadPool` to support concurrent operations across multiple sets in a set-associative `KVCache`, which is backed by a `KVStore`. This checkpoint should be relatively straightforward, but it is IMPERATIVE you get it correct in order to be able to succeed in checkpoints 2 and 3.

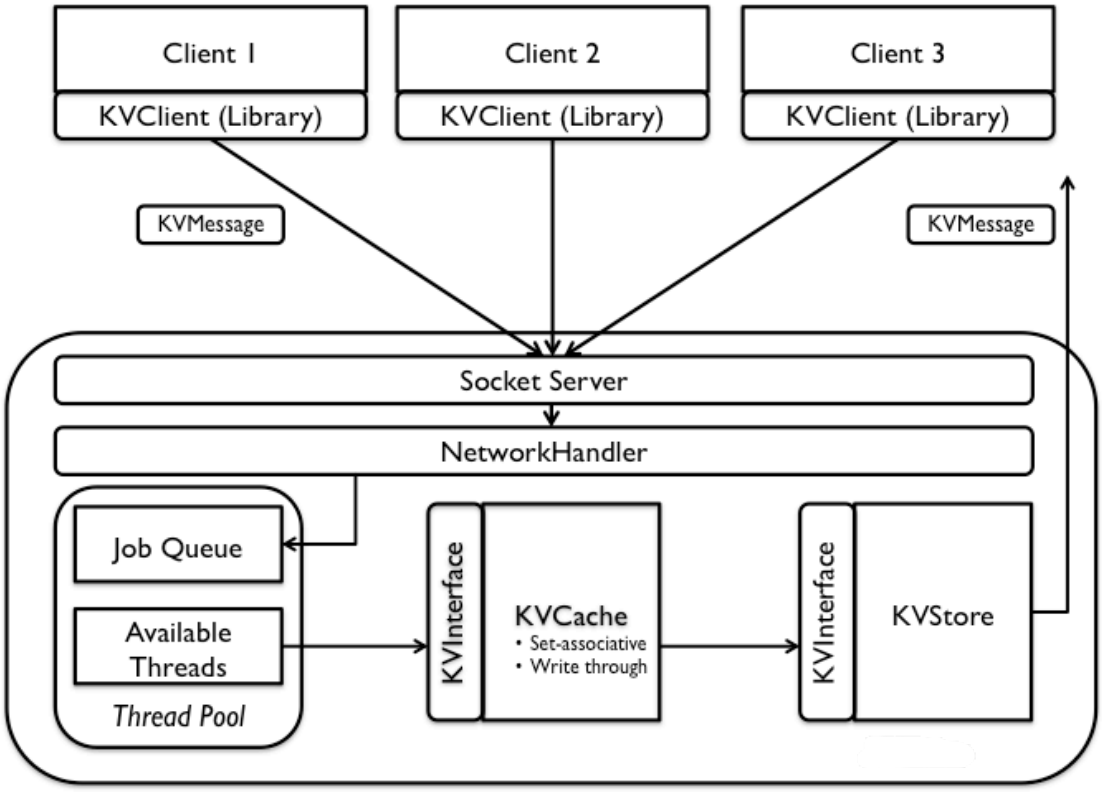


Figure: A single-node key-value store with three clients making simultaneous requests.

In later parts of the project, you will extend your single node key-value server to a distributed key-value store that runs across multiple nodes (KVServers) coordinated by a single Master (TPCMaster).

Multiple clients will be communicating with a single master server in a given messaging format (`KVMessage`) using a client library (`KVClient`). The master contains a set-associative cache (`KVCache`), and it uses the cache to serve GET requests without going to the key-value (slave) servers it coordinates. The slave servers are contacted for a GET only upon a cache miss on the master. The master will use the `TPCMaster` library to forward client requests for PUT and DEL to multiple slave servers and follow the TPC protocol for atomic PUT and DEL operations across multiple slave servers. `KVServer`s remain the same as checkpoint 1.

- Operations on the store should be atomic (succeeds completely or fail altogether) without any side effects, guaranteed by the TPC protocol.
- Data storage is durable, i.e. the system does not lose data if a single node fails. You will use replication for fault tolerance.
- The key-value store should operate in SERIAL on elements of the same cache set (Determined by the KVCache of the master) but should allow for CONCURRENT gets across different sets. Note that TPC operations (PUT and DEL) are always in serial.

Concepts to learn: two-phase commit, logging and recovery, consistent hashing, failure detection using timeouts

Figure: A distributed key-value store with replication factor of two.

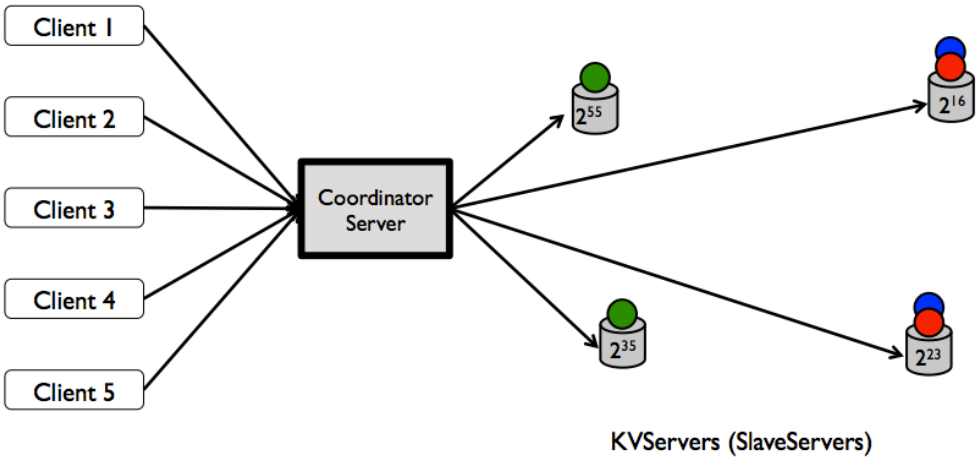
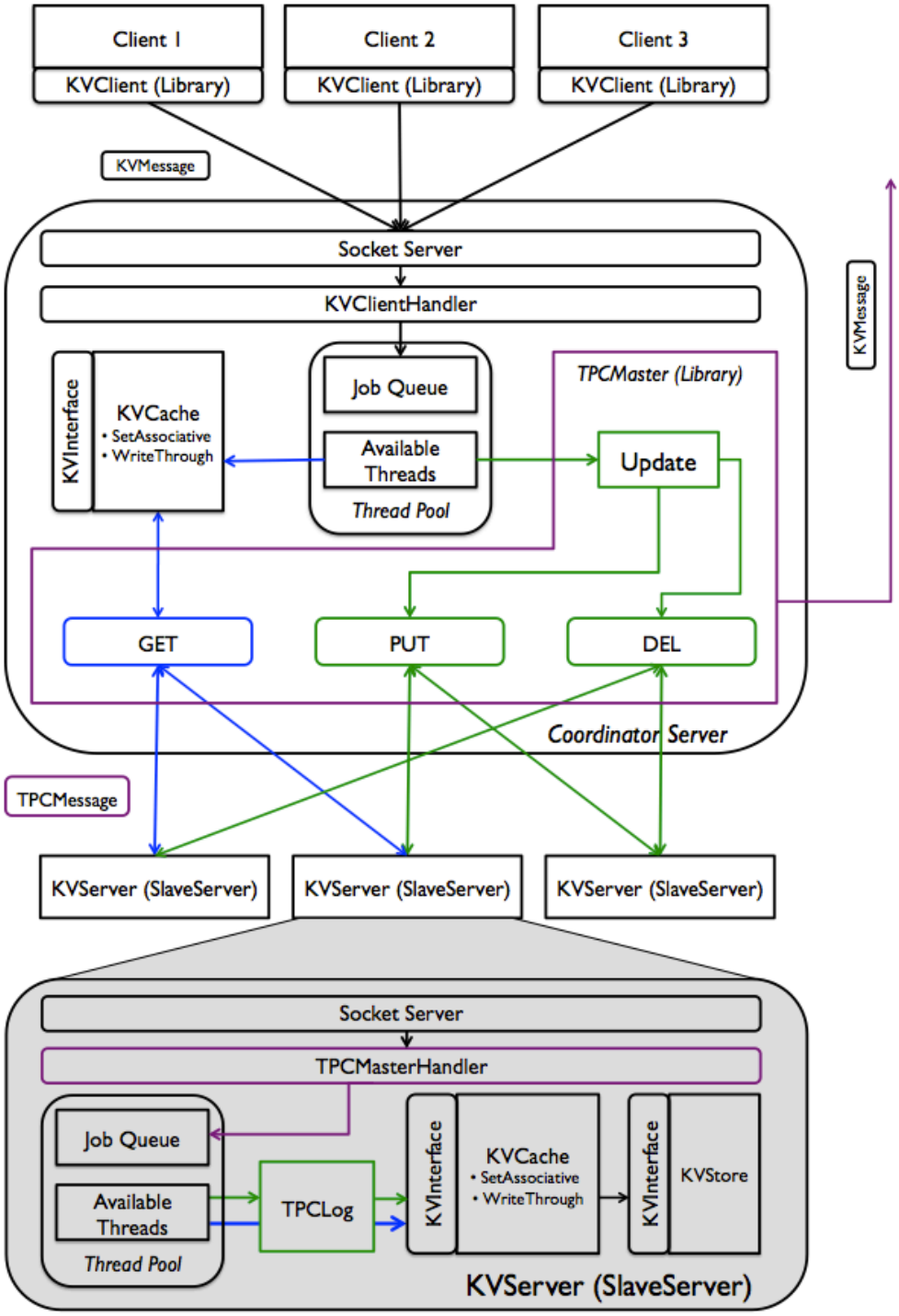


Figure: Functional diagram of a distributed key-value store. Components in colors other than black are the key ones you will be developing for this project. Blue depicts GET execution path, while green depicts PUT and DEL; purple ones are shared between operations. Note that, components in black might also require minor modifications to suit your purposes. Not shown in the figure: PUT and DEL will involve updating the write-through cache in the master.



Required Features

Read the following information very carefully. They are relevant to implementing all parts of the project correctly. We recommend you keep a checklist of implemented features as ignoring any one of these could cause you to fail tests. They are ordered in the sequence in which they should be completed.

Slave Server Registration

- Slave servers will have 64-bit globally unique IDs (Java `long`'s), and they will register with the master with that ID when they start. For simplicity, you can assume that **the total number of slave servers is fixed**. There will not be any slaves that magically appear after registration completes, and any slave that dies will revive itself.
- The master will listen for client requests from port 8080 and listen for registration requests from slaves on port 9090.
- When a slave starts it should start listening in a random *free* port for TPC requests and register that port number with the master so that the

the master can send requests to it. You should have already implemented the logic to handle free ports in project 3.

- A slave has successfully registered if it receives a successful response from the master in the form shown below.
- Note: When parsing the registration, remember that the SlaveServerID can be negative.

These are the message formats for registration. You are free to create the registration message itself with string concatenation.

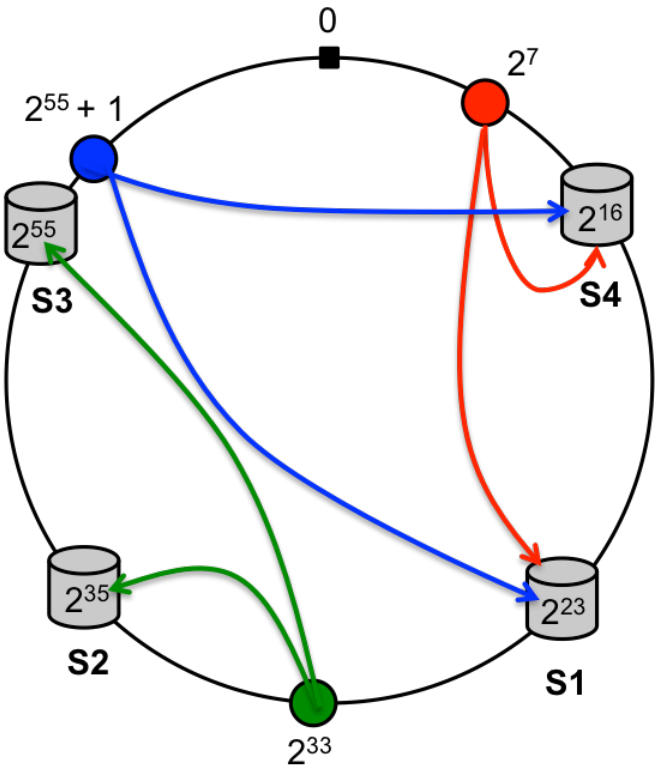
register	`<?xml version="1.0" encoding="UTF-8"?>`
	`<KVMessage type="register">`
	`<Message>`*`SlaveServerID@HostName:Port`*`</Message>`
	`</KVMessage>`

resp	`<?xml version="1.0" encoding="UTF-8"?>`
	`<KVMessage type="resp">`
	`<Message>Successfully registered `*`SlaveServerID@HostName:Port`*`</Message>`
	`</KVMessage>`

Consistent Hashing

- Each key will be stored using TPC in **two** slave servers; the first of them will be selected using consistent hashing, while the second will be placed in the successor of the first one. Note that the hash function is provided for you in TPCMaster.
- There will be **at least two** slave servers in the system. Each key-value (slave) server will have a unique 64-bit ID. The master will hash the keys to 64-bit address space.
- Each slave server will store the first copies of keys with hash values greater than the ID of its immediate predecessor up to its own ID. Note that each slave server will also store the keys whose first copies are stored in its predecessor. These IDs and hashes will be compared as *unsigned* 64-bit longs with functions provided in `TPCMaster`.

Figure: Consistent Hashing. Four slave servers and three hashed keys along with where they are placed in the 64-bit address space. In this figure for example, the different servers split the key space into S1:; [2^{16} + 1, 2^{23}], S2: [2^{23} + 1, 2^{35}], S3: [2^{35} + 1, 2^{55}] and finally note that the last server owns the key space S4: [2^{55} + 1, 2^{64} - 1] and [0, 2^{16}]. Now when a key is hash to say a value 2^{55} + 1, it will be stored in the server that owns the key space, i.e, S4 as well as the immediately next server in the ring S1.



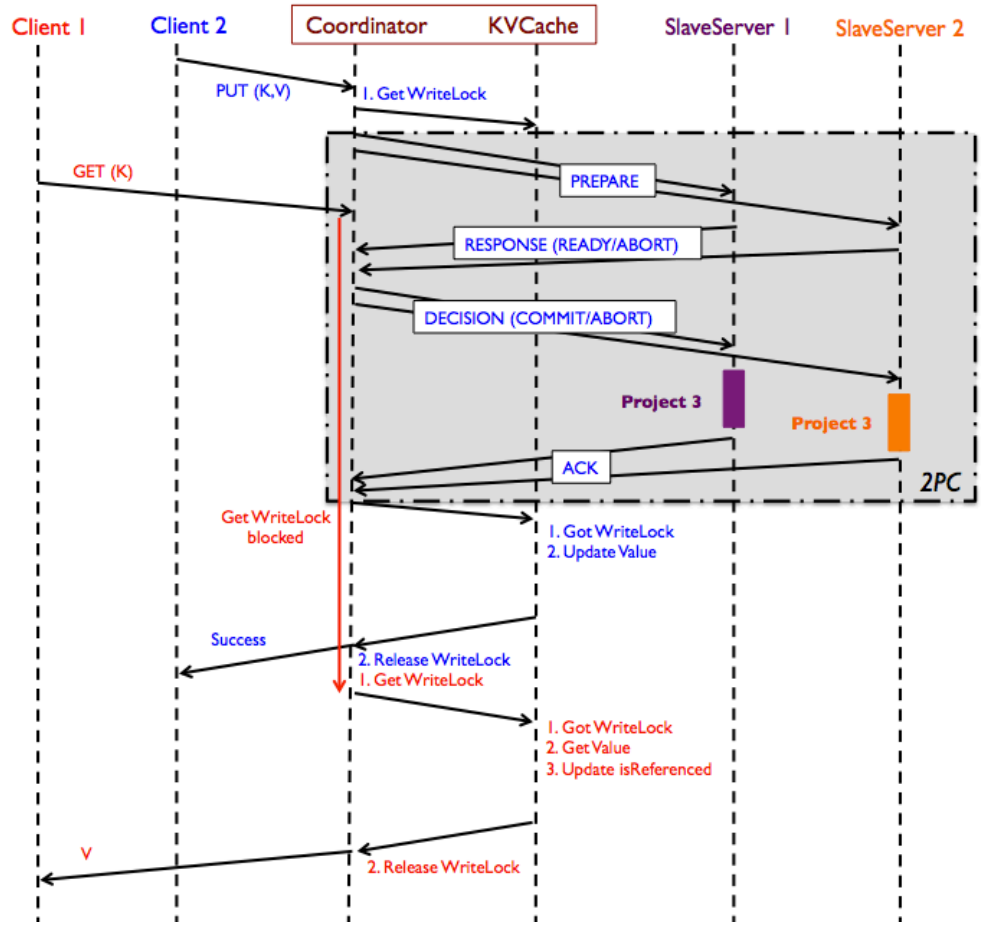
Two-phase Commit

- `TPCMaster` must select replica locations using consistent hashing. **Only a single two-phase-commit operation can be executed at a time** (`handleTPCRequest` is monitored by the Java keyword `synchronized`).
- You do not have to support concurrent update operations across different keys (i.e. TPC PUT and DEL operations are performed one after another), but retrieval operations (i.e. GET) of different keys must be concurrent unless restricted by an ongoing update operation on the same

set.

- The master will include a set-associative cache, which will have the same semantics as the cache you used before. If the master finds the key in its cache, it must not contact any slaves.
- You must wait until all slaves register before servicing any requests. This means you must block until `registerSlave` has been called successfully for `numSlaves` unique slaves.
- A slave will send VOTE-ABORT to the master if the key doesn't exist for DEL or an oversized key/value is specified for PUT.
- When sending phase-1 requests, the master must contact both slaves, even if the first slave sends an abort. You can do this by sequentially making the requests or concurrently by forking off threads. (Sequentially is obviously easier)
- In theory, if the master receives any response from the slave in phase-2, it should be an ACK (we ask for this guarantee from phase-1). However, for the case of "error handling", if the master receives anything besides an `ACK`, throw a `KVException` `ERROR_INVALID_FORMAT` and return this to the client.

Figure: Sequence diagram of concurrent read/write operations using the TPC protocol in project 4. Project 3 blocks in the diagram refer to the activities when clients write to a single-node slave server, where the master is the client to individual slaves. GET request from Client 1 is hitting the cache in the above diagram; if it had not, the GET request would have been forwarded to each of the slave servers until it is found.



Failures, Timeouts, and Recovery

At any moment there will be **at most one slave server down**. Upon revival, slave servers always come back with the same ID. For this particular project, you can assume that **the master will never go down**, meaning there is no need to log its state. Individual slave servers, however, must log necessary information to survive from failures.

Your code should be able to handle one slave server failing at any point in the TPC transaction. An example of setting up a slave failure situation is in TPCMurderDeathKillTest.java. It is sufficient to treat a slave failure as simply killing the thread in which that slave server is executing, resulting in deletion of its state.

- On failure, we assume our in-memory `KVStore` is wiped. When the slave comes back up, it will be rebuilt using the log that is has been updating. (TPCLog.java)
- When a slave comes back up, it does not contact the server or other slaves. It rebuilds from the log and should figure out if the last request it received was a phase-1 request from the log.
- If a slave crashes anytime during phase-2 (including before receiving the global decision), the master will need to keep trying to send the global

message to that slave until it gets a response (retry using timeouts).

- During phase-1, if master does not get a vote within a single timeout period, it should assume the slave voted `abort`.
- During phase-2, the master must retry (with timeout) until it receives a response to its decision. You must send a decision to slaves that you timeout on because they may be waiting on a decision once they reboot. Note that in the case that the slave restarts, it may bind to a new port and re-register. Your master node must retry with the latest host-port the slave has registered with. Remember that slaves always restart with the same ID. GET requests may be served concurrently during this time.
- If a slave finishes performing the operation in phase-2 but fails before sending an ack, the master will keep trying to send the decision. In this case, the slave will get a duplicate decision from the master. You should ensure that the behavior of your slave is IDEMPOTENT.
- It is up to you to figure out which messages you need to write to your log and at which points in the code you need to write them. Although we say a slave server can crash at any time, for simplicity, you may assume there are not crashes during calls to `TPCLog.flushToDisk()`. We will not be white-box testing the contents of your log. You MUST call appendAndFlush() to actually commit entries to the log.