ECE419 Milestone 4 Design Document

Team 24: Chenhao Liu, Jue Liu, Ze Ming Xiang

(All members have equal contribution)

1 Overview

We implemented a replicated, scalable, distributed, transactional key-value storage server application based on our Milestone 3 codebase. The new features, improvements, and modifications added to our Milestone 3 implementation are documented in the sections below. This iteration of our application features support for ACID transactions as well as a client-side interpreter for drafting transactions.

Additionally, see `README.md` in the source code for instructions on environment setup.

1.1 Server Changes

To support ACID transactions, the KVServer implemented support for various new KVMessage request statuses. When the server receives a `TRANSACTION_BEGIN` request, the `ClientConnection` receiving the request will set its inTransaction variable to true, meaning that the server should now expect transaction operations (`TRANSACTION_GET` and `TRANSACTION_PUT`). Transaction state terminates when either `TRANSACTION_COMMIT` or `TRANSACTION_ROLLBACK` is received, or when enough time has passed since `TRANSACTION_BEGIN` is received. During transaction, `TRANSACTION_GET` and `TRANSACTION_PUT` requests will be recorded in a KVStorageDelta object (see Milestone 3), instead of being directly applied to storage. This means that normal `GET` or `PUT` requests from any client will not be affected by transaction modifications until the transaction is complete. Furthermore, `TRANSACTION_GET` requests will first read from the delta object, and then read from the actual storage if the delta object does not contain a value for the key. This is to ensure read-your-writes consistency for the ongoing transaction. When `TRANSACTION_COMMIT` is received, the server will iterate over all tuples stored in the delta object and apply them to the actual storage. When `TRANSACTION_ROLLBACK` is received, the delta object will be discarded, which means no change will be applied to the database.

In order to preserve consistency within a transaction, all keys read or modified during a transaction will be locked from modification by all other clients except the current one. This is achieved using a synchronized hashmap at the server level, which will store all keys that are locked along with the client that locked it. Additionally, a TimeoutWatcher thread will run every 2 seconds to check if any client has started a transaction but did not commit or rollback for more than 5 seconds. If so, all keys locked by that client will be unlocked, and the ongoing transaction by this client will be cancelled.

The support for transactions guarantees the ACID properties: Atomicity is guaranteed through the commit and rollback operations; any intermediate failure means the entire transaction will not be applied; additionally, key-locking prevents other clients from being able to interrupt any ongoing transaction operations. Consistency is guaranteed by the key-locking mechanism, as well as the fact that only coordinators can serve transaction requests. Isolation is guaranteed since the key-locking mechanism can safely allow two transactions to be applied in parallel if they do not affect the same keys. Durability is guaranteed since each commit operation immediately writes the results to disk.

1.2 Client Features

1.2.1 KVStore Transaction API

The API support for transactions has been implemented on the KVStore, which is the main logical interface for KVClient to communicate with a KVServer. Specifically, KVStore implements a `runTransaction` method, which takes a `TransactionRunner` object (usable with Java lambda expressions) as transaction and attempts to execute it on the server.

Within `runTransaction`, the KVStore will be in transaction mode. In transaction mode, KVStore will only accept `transactionPut` and `transactionGet` instead of the usual `get` and `put` methods. These "transaction versions" of the methods will handle special logic dedicated to performing the transaction correctly, such as using `TRANSACTION_GET` instead of the usual `GET` status in the request message. In a transaction get/put method, if the target server has not been requested before within the current transaction, a `TRANSACTION_GET` request will first be sent to the server to signal the start of a transaction. In the event of a failure, the KVStore will send `TRANSACTION_ROLLBACK` to all servers affected during the transaction up to this point. The KVStore will also monitor for server topology changes potentially affecting the transaction by tracking the target server for each key requested during a transaction. If the target server changes for any given key, the transaction is automatically rolled back and retried. Finally, if the runner executes without error, the KVStore will send `TRANSACTION_COMMIT` to all servers affected during the transaction, and return from `runTransaction`.

The KVStore also supports automatic transaction retry. The most common case for retry happens when another client is concurrently performing a transaction on one or more keys affected during the current transaction. Subroutines in KVStore will throw a special `RetryTransactionException` when a detected failure should be handled with an automatic retry. If this exception is caught, the KVStore will rollback the transaction and attempt it again up to a maximum limit of 5 retries.

1.2.2 KVClient Transaction Interpreter

The KVStore transaction API is programmatic, and is not friendly to command-line interface users. Therefore, a simple command-line interpreter for transactions is developed to allow KVClient users to define and execute transactions with a simple shell-like language. When the user enters the `transaction` command, the KVClient will enter transaction definition mode. The user can then define and execute a transaction, similar to defining a function and executing it in a programming language shell. The following is an example command sequence entered by the user to transfer 10 dollars from Alice's account into Bob's:

```
KVClient> transaction
.. $a = get alice
.. $a = $a - 10
.. put alice $a
.. $b = get bob
.. put bob $b + 10
.. end
```

Once the user enters the transaction mode, the prompt will change, and users can then enter statements to be executed in the transaction. The first statement, `$a = get alice`, retrieves the value

from the database corresponding to the key “alice”, and stores it in a variable “$a”. A special variant of the “get” expression is supported to supply a default value if the key is not found, for example: `$a = get alice default 0`. The second statement, `$a = $a - 10`, subtracts 10 from the variable “$a”. Currently, all 4 arithmetic operators are supported, with the limitation that both operands must be integers, and the “/” operator will perform integer division instead of floating-point division. The third statement, `put alice $a`, sends a request to the database to set the value of the key “alice” to the value stored in the variable “$a”. The fourth and fifth statements are similar to the first three, but modifies the account balance of Bob instead. Finally, the user enters “`end`”, which tells KVClient to finish defining the transaction and executes it.

The interpreter is implemented as a simple AST parser. As the user enters statements, the parser first tokenizes the line, and recursively parses the array of tokens into special objects representing AST nodes (such as `InfixOperation` or `Assignment`). For example, the statement `$a = $a - 10` is parsed into an `Assignment` object, which stores an `InfixOperation` object by parsing the substring `$a - 10`. The `InfixOperation` object further stores the left operand as a `Variable` object by parsing `$a` and the right operand as a `Constant` object by parsing `10`. When the statement is executed, it recursively evaluates its sub-expressions, and executes its own logic based on the return values from the sub-expressions. The variable assignments and evaluations are backed by a simple hashmap storing all variable mappings in the global scope. The “get” and “put” commands will call the `transactionGet` and `transactionPut` methods respectively in the `KVStore` and use their return values.

Once a transaction is defined, a `TransactionRunner` object will be generated and executed, which simply evaluates the defined statements in order. The user can optionally use the `def transaction <name>` command instead of `transaction` to define a transaction without executing it, and later executes it with `transaction <name>`.

2 Performance Report

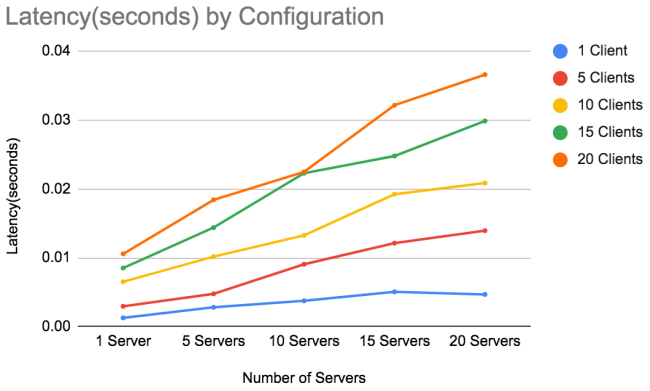
2.1 Method

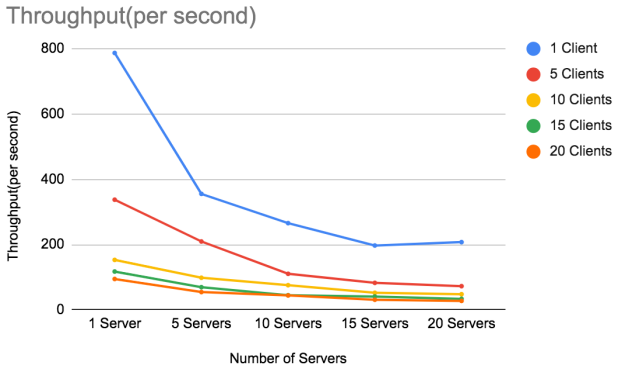
The average latency and throughput of our application is profiled for different numbers of servers and clients. We tested for latency and throughput using 1, 5, 10, 15 and 20 servers. These servers then served the requests from 1, 5, 10, 15 and 20 clients. Our Benchmarker script measures the latency and throughput for each of the 25 combinations, by having each client repeatedly send transaction requests. For each transaction from each client, we send a transaction that reads the value of a random key (default to “0” if the key does not exist), subtracts the value by 10, puts the value back, reads the value of another random key (default to “0” if the key does not exist), increases the value by 10, and puts the value back. Each transaction conserves the total sum of all values (which should be 0). Each benchmark lasts for 10 seconds, with the results from the first 1 second ignored as warm-up.

The correctness of our transaction mechanism is also tested, by checking the sum of all values stored in the server at the end of each benchmarking run. **Our implementation successfully passed the test by having a sum of 0 at the end of all benchmarking runs.**

The tests are run on a mid-2015 MacBook Pro, with a 2.2 GHz Intel Core i7 processor and 16GB memory.

2.2 Result





The results of our experiments are shown in the above figures. There are two trends immediately visible from the graphs: 1. As the number of clients increase, the performance of the entire system decreases; 2. As the number of servers increases, the performance of the entire system decreases. The first trend can be explained by the fact that more clients implies more requests made per second, which increases the number of requests processed per server. The second trend can be explained by the fact that with more servers, the probability of each transaction needing to interact with more servers increases, which increases the expected overhead. Additionally, the system is executed on a single computer, which has a limited number of independent processor cores. When the number of servers gets larger, the computing power of the processors start to become depleted, and the overhead of coordination starts to take over, resulting in decreased performance.

Appendix A: Test Report

All Milestone 1, Milestone 2, and Milestone 3 tests, including ConnectionTest, InteractionTest, ApplicationTest, and AdditionalTest (from M1, M2, and M3), are modified to work with our Milestone 4 codebase. These act as regression tests to ensure that the functionalities in Milestone 1, 2, and 3 are intact.

Tests added in AdditionalTest.java for Milestone 4:

Test Name	Test Summary	Test Details
testKeyLock	Test the correctness of locking write operations on specific keys.	Create a server instance with 2 clients, lock 1 key per client, check if write operations are correctly blocked if a client attempts to write to a key locked by another client. Write operations should succeed if the key is locked by the same client.

testTimeoutChecker	Test the correctness of the timeout checker sentinel thread.	Create a server and lock a few keys. Wait for a certain duration, and check if the keys are automatically unlocked by the timeout checker thread.
testKVStorageDeltaGet	Test the correctness of the “get” method of the delta object.	Create a KVStorageDelta object, use “put” to add changes to it, and check if the return values from subsequent “get” calls are correct.
testKVStorageDeltaClear	Test the correctness of the “clear” method of the delta object.	Create a KVStorageDelta object, use “put” to add changes to it, then call “clear”, and check if the delta object no longer contains any entries.
testTransactionBegin	Test the correctness of the TRANSACTION_BEGIN request.	Create a server, send a TRANSACTION_BEGIN request to it, and check if the server responds with TRANSACTION_SUCCESS.
testInvalidTransactionPutGet	Test whether the server will reject TRANSACTION_GET and TRANSACTION_PUT if TRANSACTION_BEGIN has **not** been sent.	Create a server, send TRANSACTION_GET and TRANSACTION_PUT requests without sending TRANSACTION_BEGIN first. Check if the response statuses are FAILED.
testTransactionPutGet	Test whether the server will correctly handle TRANSACTION_GET and TRANSACTION_PUT if TRANSACTION_BEGIN has been sent.	Create a server, send TRANSACTION_BEGIN request, then TRANSACTION_GET and TRANSACTION_PUT requests, check whether the responses are successful.
testTransactionRollBack	Test the correctness of the rollback operation on transactions.	Create a server, begin a transaction and use TRANSACTION_PUT to issue a put request on the transaction, then send TRANSACTION_ROLLBACK. Check whether the value from GET request after rollback is still the same value before the

		TRANSACTION_PUT.
testTransactionCommit	Test the correctness of a successful transaction.	Create a server, begin a transaction, use TRANSACTION_PUT to issue a put request on the transaction, and finish the transaction with TRANSACTION_COMMIT. Check whether the subsequent GET responses have the new value.
testTransactionDelete	Test the correctness of a successful transaction involving deleting a tuple.	Create a server and put a tuple into the server. Start a transaction, use TRANSACTION_PUT with a null value to delete the tuple. Commit the transaction and check whether the tuple is deleted.
testKVStoreTransaction	Test the correctness of the transaction API on KVStore.	Create a server, create a KVStore, and use the runTransaction method on the KVStore to execute a simple transaction. Check whether the subsequent GET responses have the new value.
testTransactionIsolation	Test whether intermediate values during a transaction attempt is not visible to other clients.	Create a server and 2 clients; begin a transaction on client 1 and send a TRANSACTION_PUT request without committing. Check whether GET request on client 2 still returns the old value.