University of Toronto

Faculty of Applied Science and Engineering

ECE419

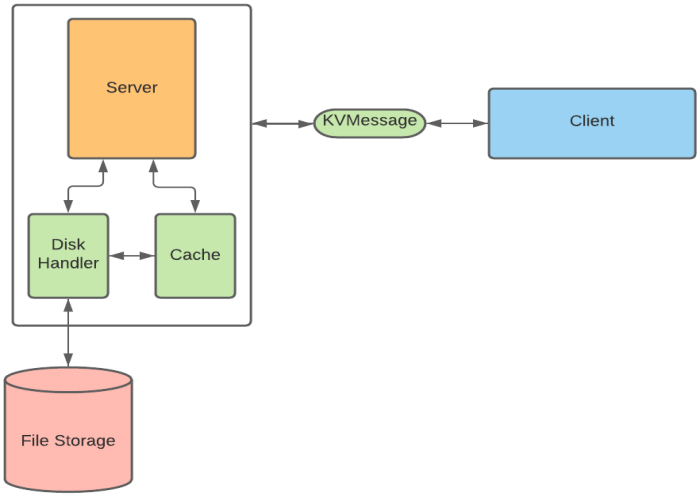
M1:Client and Persistent Storage Server

Team 21	Rui Zhou (1002894754) Hong Xu (1002079305) Shuoer Wang (1005345128)	**Date**	2021-01-28

1.0 Design Overview

The objective of the design is to develop a server that takes the key-value pairs from the client-side and persistently stores them into the hard disk. Thus, the scope of the project includes the perspective of client, server, and storage. From the client-side, it should be able to bind to a port and send the data based on the user input. The server should be able to accept the incoming data and make corresponding actions or execution regarding the command received from the client. In the meanwhile, the server should also be able to handle the persistent storage functionality, which means it needs to perform the I/O operation to store the key-value pairs into the disk. The following section would elaborate on the detailed design and implementation.

Figure 1.0 System diagram



1.1 Client Logic

The *KVClient* in app_kvClient is the application of the client that interacts with the users. It uses *KVStore* as the library of the key-value store service. The shell commands of *KVClient* follow the description of the handout, and the API of *KVStore* follows the start code except for the logic for delete by *put* command. The team added a new enum *StatusType* element *DELETE* and the corresponding delete API in *KVCommInterface*. So now it is the application's responsibility to detect the special case of *put* and invoke the *delete* in *KVStore* Class that has implemented the *KVCommInterface*. For general *put* command, the first argument is the key, and all arguments after the key are the string of the value. So the key-value service supports value with space inside.

1.2 Server Logic

The KVServer in app_kvServer is the application that provides the connection between the client and the storage disks. The KVServer will handle the commands from clients including GET, PUT, and performing corresponding actions on the data.

1.3 Communication Logic

For the message sent between client and server, the team designed the class *KVBasicMessage* which implemented the interface KVMessage. It has the key-value pair and the status(type) of the message as its field. In the key-value store service, this is used both by the server and client for all kinds of communications.

The team found that it is easy to format the input of the command-line shell to a string by separating the key, value, and the type of message with space. Then, we decided to make use of the *TextMessage* class from M0 to serialize the string of the message to bytes array. The *TextMessage* class has methods to convert between bytes array and string, and use CR to indicate the termination of the bytes array, which is nice. All we have to do is format the string of the message when sending, and read the bytes until CR occurs when receiving.

In addition, We designed an abstract class *KVMsgProtocol* worked as a protocol for communication, which is basically responsible for the two jobs just mentioned. It has the input and output stream of the *socket* as its field, and is extended by *KVStore* (client) and the *KVServerConnection* (server). When it receives the task for sending a task to send a *KVBasicMessage*, it will convert this message to a "*StatusType* + *Key* + *Value*" format of the string, and use the *TextMessage* to serialize the string and add return char at the end, and send through the socket stream. When it receives the byte string from the socket stream, it reads the byte until seeing a return char, and then create the *KVBasicMessage* object

It is worth mentioning that this design is based on the fact that the client and server are both implemented in Java and both of their messages sent can use the *KVMessage* structure. If in the future, server and client are developed by different languages as different projects, the bytes based communication can still assure the compatibility of most code, but the *TextMessage* and *KVMsgProtocol* may be implemented separately on client and server sides.

1.5 Storage Logic

The storage functionality of the server should be able to make the server persistently store the key-value pairs dataset. Thus, the system designed a file storage functionality to handle this requirement. Whenever the server receives a connection from the client, it will initialize a new text file to store the key-value pairs data. When the server tries to do a put execution, the server will first check if the key already exists in the file. Then the server will append this key-value pair at the end of the file if it does not exist in the file. Otherwise, it will update the original data with the new incoming value. In terms of getting a request, it will perform an interaction of the entire entry on the file to check if it exists and return null if it does not.

2.0 Performance Evaluation

Our performance test runs 10000 times of *get/put* operations in total, and in each iteration, it has different probabilities to run *get* or *put*, so the *get/put* operations occur randomly among this 2000 times. The *key* of the operations approximately range from 0 to 99, to simulate the real-world usage, the key we use in the 10000 operations fits to the normal distribution, so the numbers in the middle of the range have greater chances to be used.

The following table illustrates the performance evaluation done by the team. We have Implemented two caching strategies to compare the effectiveness between them. The team also measures the performance of the persistent storage as the baseline for the system. In order to have a broad view of the different performance and accommodate diverse usage of the system, the team uses the metrics of average latency, average put latency, average get latency, and operation per sec.

Number of random keys generated: 100	**Cache: FIFO**			**Cache: LRU**			**Disk**		
Put / Get percentage	80% / 20%	50% / 50%	20% / 80%	80% / 20%	50% / 50%	20% / 80%	80% / 20%	50% / 50%	20% / 80%
Cache Size: 20							**Cache Size: 0**		
Avg Latency (ms)	2.4441	2.6198	2.4323	2.5187	2.7051	2.5228	2.77	2.4611	2.3914
Avg put Latency (ms)	2.44	2.6433	2.4426	2.5115	2.703	2.5274	2.9174	2.8199	3.0575
Avg get Latency (ms)	2.4487	2.5905	2.4269	2.5344	2.7011	2.5185	2.1562	2.0924	2.2172
operations per sec	409.15	381.71	411.13	397.03	369.67	396.38	361.01	406.32	418.17
Cache Size: 100							**Cache Size: 0**		
Avg Latency (ms)	1.968	1.9755	1.9501	1.8719	2.0656	2.0207	\	\	\
Avg put Latency (ms)	1.9682	1.9751	1.9699	1.8688	2.0825	2.1014	\	\	\
Avg get Latency (ms)	1.9711	1.9417	1.9548	1.8718	2.0423	1.9965	\	\	\
operations per sec	508.13	506.2	512.79	534.22	484.12	494.88	\	\	\
Cache Size: 200							**Cache Size: 0**		
Avg Latency (ms)	1.6478	1.709	1.6613	1.7825	1.851	1.7218	\	\	\
Avg put Latency (ms)	1.6438	1.7143	1.6576	1.7768	1.8507	1.7228	\	\	\
Avg get Latency (ms)	1.6537	1.6996	1.6597	1.7894	1.8457	1.7186	\	\	\
operations per sec	606.87	585.14	601.94	580.21	540.25	580.79	\	\	\

3.0 Appendix

Besides the performance evaluation, the team also performs the unit test (via JUnit) to verify the functionalities of the system. Overall, the system passes all the given unit tests and additional unit tests (21/21).

Test Suite	**Test Explanation**	**Status**
Concurrency Test	Test suite is additional, and it purposes to test the ability of the server while handling the multiple client connections. The successful test should have a server capable of handling all the requests from multiple clients.	Passed
Connection Test	Test suite is given, and it tests the connection between server and client. The successful test should be able to connect to the server with port 5000.	Passed
FIFOCache Test	Test suite is additional, and it tests whether the cache can work properly. It tests whether the cache will update the disk if the cache is out of capacity.	Passed
Interaction Test	Test suite is given, and it tests the functionalities of the CRUD requests. The successful test should be able to perform all the requests without errors.	Passed
Storage Test	Test suite is additional, and it tests the functionalities for persistent storage, includes CRUD operations, and the ability of handling cases of different languages and symbols.	Passed
Additional Test	Test suite is additional, it tests the length checking of the key and value from client input, and it tests the put and get operation when the value contains spaces.	Passed