The (Toy) KVStore 1.0.0.1

1. An Overview of KVStore: Future Build and More.

The model of persistent storage for this project is based on LSM tree, suitable for large volume of W but small volume of R: CUD will be collected into the state variable inside key-value pair, bundled into the data.log file, and every once in a while, the database will delete the logs with DEL state and Merge the old ones with the new ones. (Since I/O is in order)
Our implementation modified the above approach, taking on a more radical change:
In RAM, every opening we read the whole file of data.log inside the disk to RAM, which is implemented in a hash table, so the READ takes constant time. In disk, we write the data in RAM into Disk whenever the server receives SET command.
[Added for dkvs] To ensure the broadcast scheme, the current version uses RAM for spawned process to access to the broadcast messages. Specifically, I define a class to keep a list of servers *kvs. Moreover, for different spawned go routine to access the messages sent from broadcast, I used a in-RAM map billboard, and goroutine can access it if it is also a member function of dkvs.

```
type dkvs struct {
    mu         sync.Mutex
    nServer    int
    serverList map[int]*kvs
    billboard  map[int]msg
}
```

This is a compromised decision, since time is tight. In the next build we will do a overhaul by replacing broadcast scheme implementation based on RPC library provided by golang.
In the present implementation, each process share the RAM, although they each have different data-*.log file for them to save data.

1.1. [Added for dkvs] Broadcast Implementation.

To ensure the broadcast, whenever broadcast is triggered, we will call `func (*dkvs).Billboard(i int, key string, value []byte)` since we want to use the in-RAM dkvs.billboard as pipeline, so we fill in message for each goroutine i, with following components:

```
type msg struct {
    active bool
    me     int
    key    string
    value  []byte
    ok     bool
}
```

msg.active checks if a message is issued, **msg.me** checks the sender index, msg.key and msg.value are self-evident, msg.ok provides the reply of receivers.
To receive this message, we spawn a goroutine `func (*dkvs).Broadcaster(i int)` for each process i, where it periodically checks if the dkvs.billboard.active is true, if so, then write the message in the server i(so we can't handle the failed write gracefully).
For sequential consistency, we will follow the implementation of ToB, i.e., for each write, we ToB the message, and wait for reply until all are informed. This is done in the function handleRequest, where we use go channel to ensure that we will write the data only if we got all msg.ok replies. This is evident in my experiment, where the Read is blocked by the Write when ToBing, where I put the broadcast cycle to be 1s so the effects are more obvious.

For Linearizable consistency, I add the ToB for read as well.

2. Functionalities…

The server end supports the functionality of

2.1 Set

This command takes the following form:

`set <key> <value>\n`

The server will take the key-value pair, saving in the RAM when the server is running, and write to disk whenever a set command is called.

This approach is quite ad-hoc, ideally I wish to implement the update to disk every several CPU ticks since the writing to disk is somewhat slow. However, since this version is still preliminary, I'd put this feature in a more stable release.

2.2 Get

This command takes the following form:

`set <key> <nbytes>\n`

Upon entering, the client end will prompt the user to enter the value, e.g.

```
Please enter <value>
>>
```

which will take a line of n bytes followed by \r\n.

For client process:

2.3 Other commands

kill The client sends the signal to kill the running server.

exit The client process will quit when receiving this command

3. Current Issues

3.1 Concurrency

a) A new one based on RPC library!(hopefully)

b) Can we handle the failure gracefully? No! In fact, we do not have a plan for a failed set/get operations…

c) Fault-tolerance. I think I might just do raft…

3.1 Scalability

Our implementation will have significant disadvantage for scalable issues, since our online storage of the data is in fact in RAM, the data could easily trigger pagefault and drag down the speed significantly.

The solution to this issue is that we could save the value of the key-value pair in disk, while replace the key-value hashmap in RAM with key-value_index hashmap, where value_index is used to look for value in disk. Moreover, we can put, as described in section 1, the values in disk as in form of

```
struct DataEntry{
value_index int64,
value []byte,
```

```
DeleteStatus bool}
```
, so when we read in order, we can only read the entries of DeleteStatus=false.
Another solution is of course the classical B tree or B+ tree, however this would be different than our approach.

3.2 Distributed Setting

The status is TBD. Much of concurrency issues still persists. To name few:

a. We did not implement the Lamport's lock yet, so in particular, if in the setting of high concurrency, the not-total-ordered messages written to the server could cause issues;
b. We need to rethink of the scalability issues for high volume of I/O: we may have "double layers of concurrency" to worry about when we keep the index hashmap and values saved in disk: since there's huge difference in I/O for RW operations between in disk and in RAM, while the addresser is still handling a single disk I/O, there might be thousands of SET commands already having modified the in-RAM key-value_index hashmap. The potential issues related to this could cause problems. I am still thinking about the solution.