FAULT-TOLERANT KEYVALUESTORE USING VIEWSTAMPED REPLICATION

VARUN RAMACHANDRA SEKAR

ABSTRACT

This report presents the implementation of Viewstamped replication protocol (shorthand - VR) to support a distributed key-value store application. It shows how client requests to the key-value store are replicated across the nodes using VR protocol, how leaders are changed when they stop responding and how lagging and crashed replicas are resynced with the leader.

1 INTRODUCTION

This report presents the implementation of VR protocol to support a distributed key-value store application. The project is fully written in Golang, including a basic client implementation to attempt a constant stream of randomized requests to the servers. A key-value store, much like a filesystem is a data dump and it's crucial that clients always get the latest value. Etcd[1], a distributed key-value store is fundamental to the operation of Kubernetes. It's important that such an application remains fault-tolerant in the presence of node failures.

The original paper on VR[2] discussed the protocol tightly coupled with the application (database) making it difficult to understand. Following the updated version of the VR protocol [3] was a lot easier as it was decoupled from the application-specific details. It also offered optimizations to improve upon the original protocol. Hence, my work here is based on the updated version of VR.

My implementation of VR consists of the following features taken from the updated Liskov paper[3]:

- Two-phase commit to handle client requests.
- View-change protocol to handle leader election.
- State transfer protocol to bring lagging replicas in-sync with the leader.
- Recovery protocol to handle replica crashes.

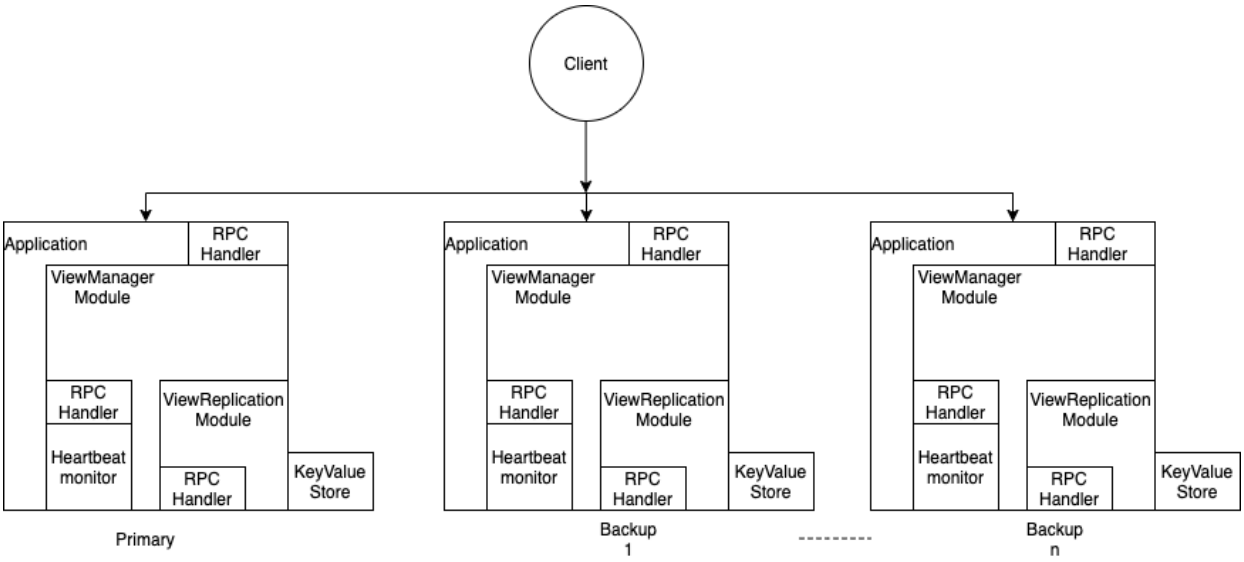
VR follows a non-byzantine consensus model i.e., there are no malicious activities happening in the cluster of replica nodes and assumes that the only way nodes fail is by crashing.

2 TECHNOLOGY STACK

The entire project is written in Golang and it uses the following libraries:

1. **net/http, net/rpc**: I've used Golang's native libraries for HTTP and RPC communication. This was a conscious design choice as this was the simplest and fastest means to quickly bring up a RPC server.
2. **html/template**: This project includes basic visualization of the state of the cluster at any point of time. To achieve this, I serve basic HTML populated with the in-memory state of the replicas using Golang's native templating engine.
3. **github.com/sirupsen/logrus, github.com/natefinch/lumberjack.v2**: Another means of visualization of the cluster that I've implemented is to write to logfiles, the communication exchanged between the replicas and the commands being executed against the replicas. To achieve this, I've used these popular logging libraries to write to two log files, op-<replica-id>.log and request-<replica-id>.log. And upon restart of the replicas, the logs are rotated.
4. **github.com/davecgh/go-spew/spew:** I've used this library for deep pretty printing structs into the logfiles.

3 ARCHITECTURE



The architecture consists of an encompassing application module that implements the RPCs to talk to the clients. Within it is the ViewManager module that implements the RPCs needed for view changes. This also runs a goroutine for the backups to monitor heartbeats from the primary and for the primary to periodically send heartbeats to backups.

The ViewReplication module contains the core logic for performing the two-phase commit for client requests. It also runs the various other protocols for syncing state between the primary

and backups like the state transfer and the recovery protocols. This module also stores information about the replicas and clients connected to it. When it is ready to commit a request, it forwards the command to be executed to the external key-value store module. As we can see, the key-value store itself is decoupled from the VR protocol and can be easily replaced by any other application. While the paper talks about having the application be a separate service that VR makes upcalls to, I've implemented the key-value store as a part of the same process for simplification.

4 DESIGN CONSIDERATIONS

The RPC communication between the replicas and the clients happen on top of TCP. The drawback of using Golang's native net/rpc library is that it only supports communication between Golang services. This was a conscious design choice for quickly bringing up the service with minimal plumbing.

For the heartbeat monitor, the timeouts needed to be chosen in a way that does not result in frequent view changes. The primary needs to send periodic commit messages at a rate faster than the timeout in the backups. Through experimentation, I came up with a difference of 1 second between the timeouts in primary and backups to allow for network delays and ensure normal operation of the cluster under normal circumstances.

5 IMPLEMENTATION

For the purposes of this project, I've implemented the key-value store as a library storing key-value pairs in memory. The key-value store provides four APIs for the VR to make upcalls to:

CREATE(string, string) error	This API creates a new key-value pair. It returns an error if the key already exists.
READ(string) (string, error)	This API retrieves the value of the specified key. It returns an error if the key does not exist.
UPDATE(string, string) error	This API updated the value of the specified key. It returns an error if the key does not exist.
DELETE(string) error	This API deletes the specified key from the store. It returns an error if the key does not exist.

Once VR is ready to commit a request, it translate the client command to one of the above API calls and executes it against the store.

To implement the VR protocol, I've defined the following RPCs:

Request(*ClientRequest, *ClientResponse) error	Clients will invoke this RPC when they want to execute a command against the distributed key-value store.
Reply(*ClientReply, *EmptyResponse) error	The replicas will invoke this RPC to tell the clients the result of the requested command execution. (The clients will need to define this RPC)
Prepare(*PrepareRequest, *EmptyResponse) error	Once the primary is ready to replicate the client request, it will invoke this RPC against the backups.
PrepareOk(*PrepareOkRequest, *EmptyResponse) error	Backups that receive a valid Prepare request from the primary will invoke this RPC if the request is valid.
Commit(*CommitRequest, *EmptyResponse) error	The uses of this RPC are two-fold. The primary will invoke this RPC to tell the backups about the last committed request from the primary so they can commit them. The primary also sends them as heartbeats during periods of inactivity.
StartViewChange(*StartViewChangeRequest, *EmptyResponse) error	Backups that do not receive any heartbeats from the primary within the timeout will invoke this RPC to initiate a view change.
DoViewChange(*DoViewChangeRequest, *EmptyResponse) error	Replicas that have received enough view change requests will invoke this RPC against the replica that's next in line to be the new primary.
StartView(*StartViewRequest, *EmptyResponse) error	The new primary chosen from the view-change will invoke this RPC to resume normal operation in the new view in the other replicas.
GetState(*GetStateRequest, *NewStateResponse) error	A replica that is lagging behind the primary will invoke this RPC to get in sync.
Recovery(*RecoveryRequest, *RecoveryResponse) error	A crashed replica that restarts will invoke this RPC to update itself with the current state of the cluster.

5.1 STARTUP

The server takes an ID as a command-line argument to startup with. This ID is used to load its configuration from the config file. It also optionally takes a boolean value as the second argument to indicate if it should startup in recovery mode. During startup of the cluster for the first time, the primary is fixed to be replica 0 and it continuously polls the other replicas with a dummy commit request until the other replicas are online and the cluster is formed. Similarly, the replicas wait for a commit request from replica 0 (primary) before they resume normal operation.

5.2 NORMAL OPERATION

During normal operation, the replicas are ready to handle client requests. The backups will reject them by returning an error. Only the primary will process client requests. During periods of inactivity i.e., when there are no client requests the primary will periodically send commit requests as heartbeats.

Client requests are handled as described in the protocol:

1. The primary accepts the client request if the request ID is valid. If it's the last served request, then it quickly responds with the result stored in the client cache.
2. Once quorum is achieved for the client request, the primary executes the requested command against the key-value store and replies back to the client with the result of the command execution.
3. The result holds a value and an error. If the command execution failed, then the error is set. The value is set only for READ commands.
4. Although no disk writes are needed, once commands are committed, they are written to op-<replica-id>.log, purely for studying the protocol and debuggability.

Golang's channel buffering is a key component in ensuring that the requests are only processed one at a time. This helps prevent race conditions where multiple logs get assigned the same op-id due to client requests arriving simultaneously.

5.3 VIEW CHANGE

When replicas stop receiving heartbeats from the primary, they initiate the view-change protocol as described in the paper[3]. View change happens in three phases:

1. First, all replicas broadcast StartViewChange requests to each other.
2. Once a replica receives *f* such requests, it sends a DoViewChange request to the new primary chosen in a round-robin manner. For example, if replica 1 was the primary in view 1, then during the view change to view 2, the new primary will be replica 2.

Replicas keep track of whether they've already sent out StartViewChange requests. This is to ensure that replicas broadcast them only once.

3. After the new primary receives *f* + 1 DoViewChange requests, it changes state, it switches to the new view and sends StartView messages to the other replicas to resume normal operation of the cluster.

One of the drawbacks of using round-robin to select the new primary is that, if the new primary is down, then the view change protocol will need to be restarted i.e., it's possible for a replica to be chosen as the new primary even though it hasn't participated in the view change protocol. This is one of the key differences from Raft[4] where a replica has to have participated in the leader election to be selected as the new leader.

5.4 Other Features

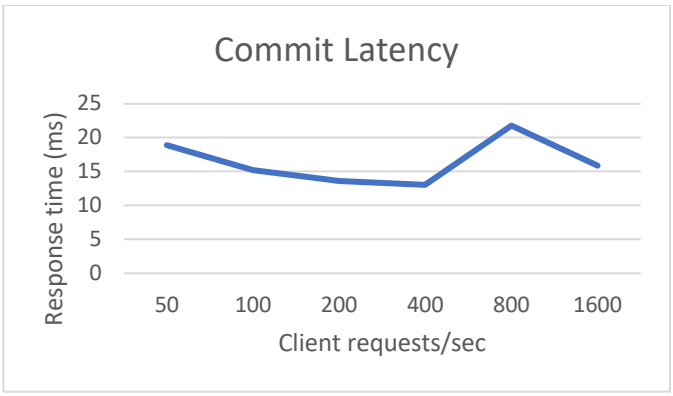
When a replica detects that it is not in the current view or does not have all the logs (at an older op-id), then it initiates the state transfer protocol to sync with the cluster. To do this, it sends a GetState request to a replica other than itself to get the updated logs and view information. It repeats this request until it hears back from a replica that is in the view requested. While the paper doesn't talk about it, I've implemented the state transfer protocol to also let the requesting replica know who the primary in the view is. This is to ensure the replica resumes normal operation as it would otherwise reject requests from replicas that it thinks isn't the primary.

When a replica crashes and restarts in recovery mode, it initiates a Recovery request to all the replicas in the cluster. While all other replicas will respond to the request, only the primary will provide the full state of the cluster. I generate a unique nonce using Golang's math/rand library and send it along with the recovery request. Responses containing a different nonce to the one sent are rejected. This isn't required since I've designed the RPC call to be synchronous, but I've implemented it anyway as the paper talks about it.

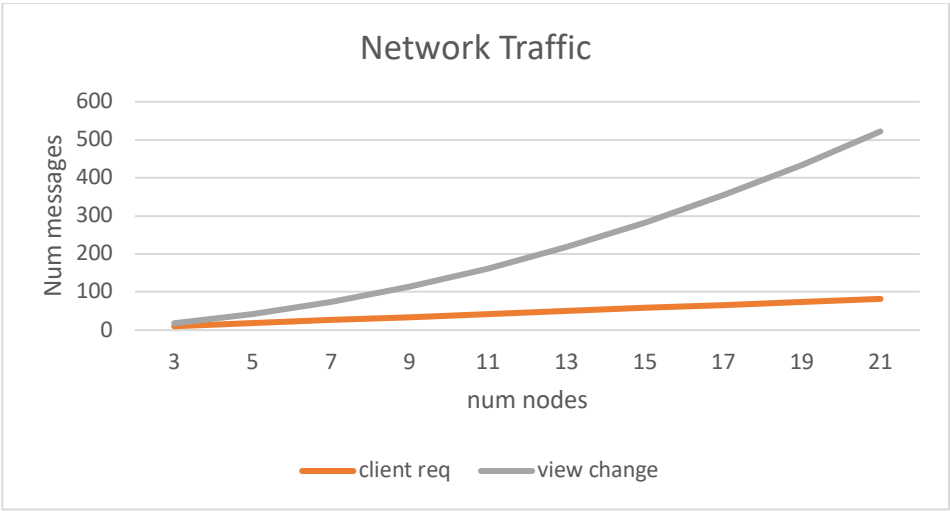
6 Performance

To evaluate VR, we need to consider 3 important questions:

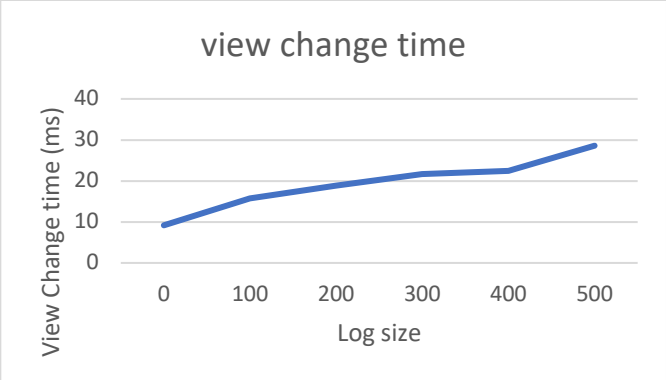
1. How long does it take for client requests to be committed?
2. How does VR scale as the size of the cluster increases?
3. How is the view change time affected as the uptime of the cluster increases?



To answer question 1, I ran a client to dispatch requests against a 5-node cluster at different rates for 20 seconds. Scaling the client request rate from 50 requests/second to 1600 requests/second, there was marginal difference in the time taken for committing requests. At 50 requests/second, each commit took about 20ms. This rate of commit did not change even as the no. of requests were increased 3-fold. This can be mostly attributed to the fact that requests are processed one at a time.



To answer question 2, I analyzed the network traffic as I ran client requests and triggered manual view changes against different sizes of the cluster. Number of messages exchanged to commit a client requests increased linearly while the number of messages exchanged during view changes increased expontentially. This can prove to be a big bottleneck to scale the cluster as network bandwidth will need to be increased as well. It is crucial that the size of the messages be kept small. The paper[3] suggests the size of the messages can be kept small by limiting the the number of log entries sent during view change, as the new primary should not be lagging behind too much under normal circumstances.



To answer question 3, I ran a client to periodically send requests to a 5-node cluster and at various points of time, triggered a view change by bringing down the primary. The time taken to complete a view change increased linearly as the size of the logs increased. This is attributed to the size of the messages being transmitted during a view change and only goes to show that it is crucial to limit the size of the messages for VR to remain performant over time.

7 Conclusion

Overall, the VR protocol is easy to understand once it was decoupled from the application-specific details that it was designed with. While the protocol eliminates the need for disk I/O, it quickly becomes apparent that, for the protocol to remain performant, writing logs to disk is essential to optimize the size of the messages exchanged during the protocol. Also, one of the hidden assumptions in the paper[3] was that the external service upcalls are always successful. This needs to be kept in mind when designing a distributed application using VR state replication.

8 References

[1] Etcd, https://etcd.io/

[2] OKI, B., AND LISKOV, B. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *Proc. of ACM Symposium on Principles of Distributed Computing* (1988), pp. 8–17.

[3] Liskov, B., & Cowling, J. (2012). Viewstamped replication revisited.

[4] Ongaro, D., & Ousterhout, J. (2015). Raft consensus algorithm.