CS 244B - Distributed Systems Project Report

Mingwei Tian, Yuchen Li and Gerald Hng

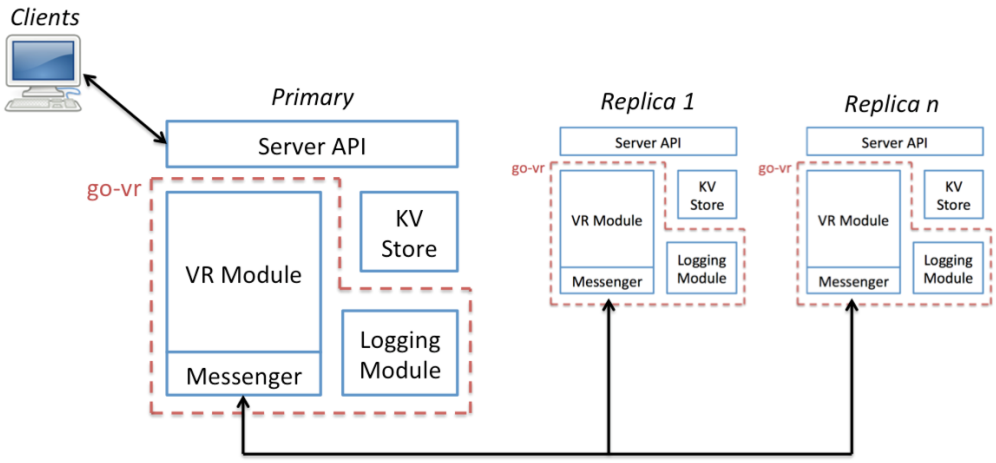
Introduction

In this report, we present Go-VR, a Viewstamped Replication (VR) library that can be easily used in distributed systems and provides visualization of the system state. The three main contributions in this project were (1) the main Go-VR implementation, (2) Go-VR-view, a visualization tool to help users track the state of the system and (3) a distributed key-value store that demonstrates our VR implementation.

Go-VR is implemented in the Go language and uses REST API to allow ease of interfacing and communication. We designed Go-VR to be imported as a library in other distributed systems with minimal changes and it was implemented based on the VR protocol described in [1] to replicate client requests and ensure consensus among nodes. Go-VR is also able to handle both primary and replica failures through the view change and recovery protocols. The codes will be put on Github and we hope to see some downloads.

To improve the system's understandability and ease of debugging, Go-VR includes a visualization tool that centrally displays the states of the protocol. We hope that this tool will be useful in future as a teaching and learning aid, as well as a debugging tool to isolate problems in systems using Go-VR.

Design Overview



Go-VR was designed to be easily integrated with other distributed system and it mainly comprises of a VR module, an inter-node messaging module and a logging module, as shown in figure 1. Through a set of provided APIs, services that need replicated operations communicate with Go-VR by registering a callback function and passing in serialized messages for replication.

Upon successful replication, Go-VR will use the callback function to inform the service that the operation is safe to execute and externalize. Our VR implementation is unaware of the implementation details of upstream services. For the purpose of demonstrating Go-VR in this project, we implemented a distributed key-value store to show that the replication of client requests and processing of view changes result in consistent states in all nodes.

For ease of ad-hoc testing and debugging, we are currently using Json over HTTP through REST API for communications between nodes. This can easily be switched to other more efficient wire formats like protobuf, by replacing the messaging module.

Implementation

Go-VR API

Go-VR exposes a very simple API to upstream services that integrates with it. The upstream service first has to register an upcall and a result-transfer handler with Go-VR, to handle operations that has been replicated successfully and can be externalized. The upstream service calls Go-VR by sending a request with the client ID, the request ID, and the request message. The client ID has to be distinct for all clients in the same replication group and request ID has to be monotonically increasing for each request. One difference with the VR paper is that in our implementation, clients call the service, which uses Go-VR. But in VR revisited, clients communicate with VR directly, which performs upcalls to the service.

API	Description
InitializeService(callback)	The upstream service uses this to register an upcall handler with Go-VR. Function ‘callback’ is executed for each successfully replicated message to inform the upstream service that the operation is safe to execute. The order is serializable. Function `transfer` sends the results of VR timeout, previous execution and results of calling `callback` back to the service.
RequestAsync(clientID, requestID, message)	Upstream service uses this to make a request with Go-VR to replicate a message.

VR Normal Operations

In our implementation, the normal operation scenario closely follows the outline from [1]. The protocol is implemented as an event-driven state machine: receiving each message is an event, which triggers state modifications and often additional messages being sent. The

implementation is straightforward - the most tricky part is the client-table, especially when the client-table needs to be updated during log replay/catch-up. Furthermore, our implementation asynchronously logs committed operations to disk in addition to having a copy in memory, unlike the paper, which keeps them in memory only. Persistent log enables the service to recover at least partially after catastrophic failure (e.g. all instances go down).

VR View Changes

When a primary fails or is unreachable by the rest of the nodes, a view change has to be performed to select a new primary. The view change protocol used in Go-VR is similar to the one described in [1] so we will not go through the details here. We will instead describe the additions we made to the view change protocol - having retries of view changes to improve liveness and using random timeouts to improve scalability.

It may be possible that a view change fails due to various reasons e.g. new primary is not working, message corruptions and network partitions. The protocol must avoid getting stuck in a view change mode perpetually i.e. it must make progress and terminate. Go-VR does this by having a separate viewchange timeout that is triggered when a view change does not complete within a specified interval (10s in our tests). Upon timeout, nodes will abort the failed view change, reset its states and start the next view change. This ensures that replicas can retry view changes after some time when network condition improves and make progress as long as a quorum of replicas exists.

To improve scalability, the timeouts used by the replicas to initiate a viewchange were also slightly randomized. Upon a time out, nodes will start broadcasting a StartViewChange message to every other node in the system. If the system is large and comprises of hundreds or thousands of nodes, this simultaneous flooding of messages may cause network congestion and degrade performance. The timeouts in Go-VR are randomized slightly, by adding between 0 to 10ms to a timeout of 10s, to stagger the broadcast of StartViewChange.

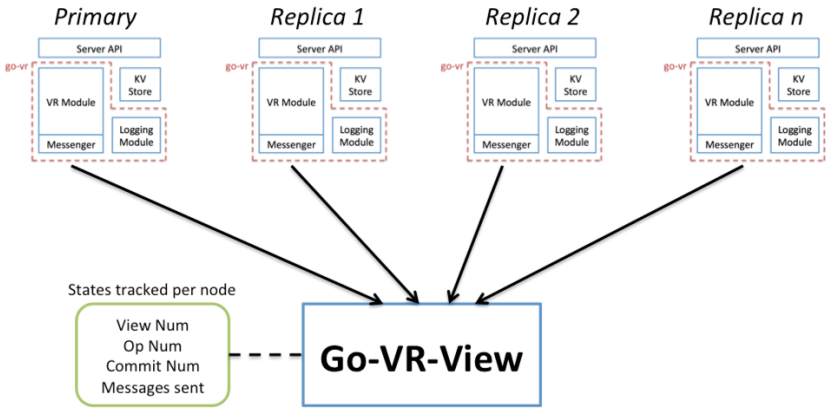
Go-VR Logs, Key-Value Store and Recovery

Go-VR has a logging module that maintains a log of all committed operations that is stored on disk. The key-value store, used mainly to demonstrate Go-VR, is located in each node's memory. Each operation that modifies the key-value pair is appended to the end of the node's log file, if it successfully commits to a quorum of replicas. Based on the records stored in the log file, Go-VR can easily reconstruct the state of the node by replaying the logged operations upon node failure.

Our log has 3 main fields: view-number, op-number, and the replicated data. The view-number and op-number are the same as described in the VR revisited paper, while the data field contains the data that the upstream service needs to replicate. For our key-value store implementation, the data field in the log contains 3 sub-fields: operation type, key and value, which record the type of operation and the key-value pair being modified by the operation.

On recovery, the node that recovers can use the log it stored on disk if it learns that its logs are still up to date i.e. the view, op and commit numbers are still valid. However, if it has an outdated log file, the recovery protocol will allow it to get just the missing entries (based on the last committed entry stored in its log) from the primary. This is more efficient than simply sending the entire primary log as described in [1]. With the updated log, Go-VR can then invoke the callback function to allow the upstream service to replay the operations and recover its state. Nodes will not participate in normal operations or view change while they are in recovery.

Visualization tool



When we first started to build a VR implementation, we realized that it would be good for future users of Go-VR to be able to easily visualize what the states of each of the replicas were and how the protocol was progressing. Thus, we decided to create a simple visualization tool, Go-VR-View, that centrally tracks the different states of the replicas and displays them to the user. The user can then see at a glance what is the latest view number, who is the primary, what are the messages that successfully committed, what messages are still preparing and the latest messages sent by each node. When visualization is enabled, nodes will send messages to a central Go-VR-View service for tracking as shown in figure 2. We think that it will be useful not just as a debugging tool, but also as an educational tool to help people understand how VR operates.

Validation Tests

We conducted tests with networks of 5 nodes, one of which is the primary and the rest are replicas. The tests were all conducted on the same machine running 5 separate terminals but Go-VR is designed to work even if the instances exist in different machines. Since Go-VR was designed as a library to be imported into other distributed system, we conducted a series of tests to ensure the correctness of the protocol. We also did some performance tests to stress the system.

Tests for normal operations

The normal operations tests mainly ensure client operations to Go-VR are handled correctly and that operations to the key-value store are replicated to a quorum before committing. The tests also ensure that all operations are written, retrieved and deleted properly by the nodes. See Appendix A for details of the tests.

Tests for view changes

We conducted various tests to ensure that the view change and recovery protocols in Go-VR works correctly. In particular, the tests focused on checking that all previously committed entries remain committed after a view change or recovery, and that nodes will not be stuck perpetually in view change or recovery mode. The details of the tests can be found in Appendix A.

Performance/stress tests

To ensure Go-VR is robust, we conducted some performance tests to stress the system. For example, we have a test that repeatedly put and delete operations of a specific key for 500 times in a short time. This is to ensure that when clients put and delete values to the nodes extensively, the final memory state and the log files still remain consistent.

Release of Codes

A quick search in Github revealed that there are few VR implementations available compared to Raft. We have released our codes in a public Github repository with the hope that Go-VR will be downloaded and used. The repository can be found at https://github.com/robertli8629/go-vr.

Appendix A - Tests Details

1	Commit with quorum	Ensures that user key-value put requests are properly committed and externalized when a quorum exist.
2	Commits without quorum	Ensures that user key-value put requests are not committed and externalized if quorum does not exist.
3	Client requests to replicas	Ensures that client requests to replicas are rejected and only primary can process requests.
4	Simple view change	Ensures that view change starts after primary fails, new primary is correctly elected and it has all previously committed entries.
5	Simple view change with recovery	Ensures that after a primary/replica fails, it is possible for it to recover and join a later view. Also ensures that all previously committed messages will be transferred to it upon recovery.
6	Consecutive primary failures	Ensures that new primaries will always be elected as long as a quorum exists. Also ensures that every replica is capable of being the primary and that all committed entries survive all view changes.
7	Aborting failed view change and starting new one	Ensures that nodes do not get stuck in failed view changes perpetually (liveness) and that view change makes progress as long as quorum exists, even in events like unresponsive new primary.
8	Recovery tests	Ensures upon failure and subsequent recovery, nodes are able to join a later view and get their entries up to date.
9	Recovery with own persistent logs	Ensures that upon recovery, a node will be able to use its own persistent logs to replay operations if the logs are up to date.
10	Recovery retries	Ensures that recovering nodes will not get stuck in a failed recovery (liveness) and that they will restart the recovery protocol periodically if it does not successfully recover.
11	Performance test	Ensures that all nodes will have the correct memory and log states after performing put and delete for 500 times.

References

[1] Liskov, Barbara, and James Cowling. "Viewstamped replication revisited." (2012).

[2] Oki, Brian M., and Barbara H. Liskov. "Viewstamped replication: A new primary copy method to support highly-available distributed systems." In Proceedings of the seventh annual ACM Symposium on Principles of distributed computing, pp. 8-17. ACM, 1988.

[3] Ongaro, Diego, and John Ousterhout. "In search of an understandable consensus algorithm." *Draft of October* 7 (2013).