Viewstamped Replication Revisited and Implemented

Li Li (lili1008@stanford.edu), Sidharth Goel (sgoel2@stanford.edu)

1 Introduction

This project is an implementation of the paper *Viewstamped Replication Revisited* (Barbara Liskov and James Cowling). The focus of the project is the naive implementation (Section 4 in the Liskov paper) without optimizations (Section 5 and 6) in order to explore the limitations of the algorithm and to understand the design choices for the proposed optimizations.

On a high level, the system comprises of a primary that accepts and executes client requests, and backups that replicate the primary states. The algorithm ensures consistency by guaranteeing that the state of the service is always replicated on at least f+1 replicas, and the service tolerates up to f byzantine failures. Even though Viewstamped Replication does not guarantee availability, the system performs view changes to make sure the system proceeds upon failure.

To keep the implementation simple and extensible for future development, we implemented the “service code” (Figure 1) as an echo service that simply outputs the same string in the client request. However, we designed an extensible RPC upon which more complicated service code can be added. Our language of choice was Go because it is simple and non-verbose, provides many useful libraries, and, most importantly, has an excellent multithreading model.

The remainder of the report is organized as follows. Section 2 lays out the architecture of each replica and explains the reasoning behind our decisions. Sections 3 and 4 describe how the View Change and Recovery protocols are implemented, respectively. Section 5 adds further details about code design choices that do not fit neatly into prior sections. Section 6 discusses some interesting results we observed from our system, and we conclude in Section 7.

2 Architecture

The two main components are the primary and backup modules. The primary (Figure 1) exposes an RPC service to the client and executes client requests, and backups (Figure 2) synchronize with the primary and initiate view changes when necessary. In the figures, circles represent ongoing thread(s), rectangles represent communication and synchronization across threads (implemented using Go channels), triangles represent in-memory storage, and arrows represent dataflows.

Primary

When the primary isn’t faulty and all participating replicas are in the same view, the system runs in normal phase. Each step of Section 4.1 in the paper is followed exactly, but there are certain ideas we added ourselves. In the normal phase, the primary node exposes an RPC server that listens on client requests, which are pushed to the Incoming Requests Queue (IRQ) - a buffered Go channel, to decouple the implementation of the RPC service and the primary logic and prevent blocking. The design allows for both linearizable operations and FIFO client order by implementing a per-client IRQ (each client's requests will be executed in order, but requests from different clients are executed in parallel). This makes sure clients do not experience much delay. However, for simplicity, we only implemented linearizable writes for all the incoming requests (all requests from different clients will be pushed to the same IRQ).

The "primary" service processes requests from the IRQ, writes to log and client table, and sends Prepare messages to backups. After a primary receives at least f PrepareOk messages from backups, it executes the service code and updates the client table with the result of the operation, before it sends the reply to the client. The underlying service code currently only returns the same message as the message in the request, but it was made extensible so that more meaningful operations can be added later.

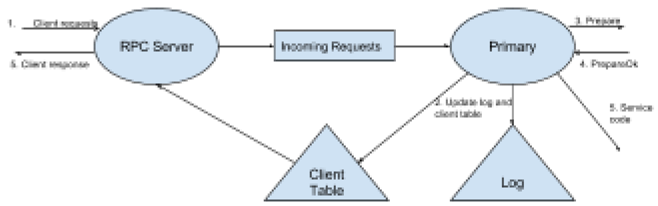

Figure 1: Primary server layout.

Backup

The Backup has a very similar design to the primary, with the main difference being the RPCs they send. After a backup receives Prepare messages from the current primary, it processes the request locally by writing to its operation log and client table, and upon success, replies to the primary with a PrepareOk message.

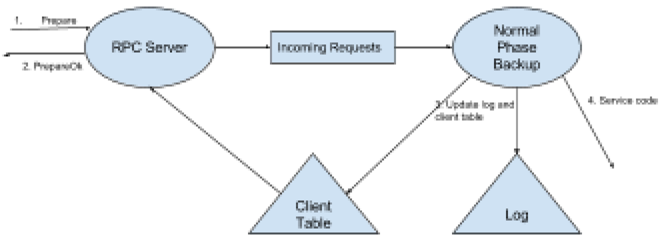

Figure 2: Backup server layout.

Client Table

This is an in-memory key-value store that records the request number of the most recent request for each client, if the request has been executed, and the result of the request. As for implementation, we used go-cache (1), an in-memory key value store for Go.

Operation Log

This is in-memory append-only array for keeping track of operations. For simplicity, we implemented the operation log using a Go array; however, improvements can be done to make data compressible or serializable so that it's easy to send them other machines via RPCs.

3 View Change Protocol

The design of the view changes follows the paper as closely possible; where details are left unspecified, we creatively figured out ways to make the system efficient. As shown in Figure 3, we implemented the protocol as a state machine (monitor module in the source code) in Go.

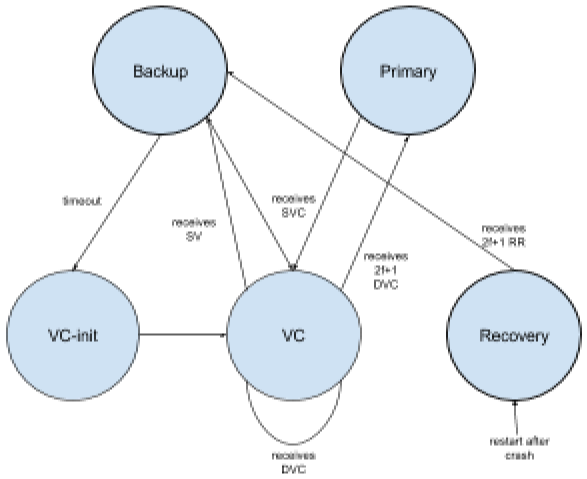

Figure 3: State machine of transitions between phases

View changes are initiated by backups. They monitor the primary and expect to hear from it regularly; when they do not, they time out and initiate a view change. The monitor module running on backups listens for messages on two Go channels: one channel for timeouts after no heartbeat messages from the primary, and another for view change messages (“StartViewChange”, “DoViewChange”, and “StartView”). If a backup times out waiting for heartbeat messages from the primary, it switches to “view-change-init” mode and starts sending “StartViewChange” messages to other replicas, and then switches to “view-change” mode. If a replica receives “StartViewChange” messages with proposed view numbers larger than the highest number it has received so far, it will switch to “view-change” mode and start advocating “StartViewChange” messages to other replicas with the highest view number. A replica starts sending “DoViewChange” messages to the new primary after it has received at least f “StartViewChange” messages from other replicas. If the new primary receives at least f “DoViewChange” messages, it will become the new primary and start sending “StartView” messages to other replicas. When a backup receives “StartView” messages from the new primary, it will synchronize with the new primary and transition into a new view.

Last but not the least, when a replica restarts after a crash, it enters the “recovery” mode, in which it sends “Recovery” messages to all other replicas and wait for f+1 “RecoveryResponse” messages, including one from the current primary. After it synchronizes with the new primary based on the information in the responses, it switches to backup mode and joins the new view.

To allow a single binary to switch between its role as a primary or backup, we needed a way to cancel the current execution of operations on a given binary at any time. For this, we passed cancellable Go contexts to all the relevant functions and cancelled the contexts when a view change had begun.

View Changes are particularly tricky because there are many timing issues with replicas talking to many other replicas, each sending different types of messages and keeping track of a lot of states.

Because of network latency, messages from different views can be sent to replicas at the same time, increasing the complexity of the system. Additionally, the various threads updating the same counters and flags required locking to prevent race conditions. This led to many subtle bugs that we noticed and fixed, often requiring small design changes. For example, currently there is a bug in which locking on a global variable can result in a buffered Go channel not being flushed, which blocks the primary from sending any messages to other replicas. Better test coverage for the system is required to find out more subtle race conditions they system can experience.

We also made sure that if node cannot connect to another node, it will retry and then stop trying rather than failing itself and causing a domino effect.

It is also important for the client to know who the primary is at all times (not just at startup) so that it knows who to connect to. If a view change occurs, the old primary, who the client is still communicating with, responds to the RPC indicating who the new primary is and the client can seamlessly dial the new primary. Additionally, if communication with the primary fails, the client will keep retrying.

4 Recovery

When a replica recovers after a crash it must first learn about the state of the world before it participates in the request processing and view changes.

To distinguish between a replica starting up for the first time and a recovering replica, we write a file to disk every time a node starts up. If the node exits normally, it deletes the file, and when a node restarts after a crash, it sees the file previously written that is not deleted yet, an indicator as to whether the replica has crashed before.

When a replica starts up after a crash, it sets itself to recovery mode and calls the Recovery RPC. Once it receives f+1 RecoveryResponses, including one from the primary, it updates it log using the state from the primary and returns to normal phase.

5 Code Design

The code was laid out in a way that logically follows from the architecture and follows Go conventions.

The entry point to any Go program is **main.go**. This runs **monitor.go**, which starts the binary in a given mode (primary, backup, or client) and switches to other modes on certain conditions as described in Section 3. The modes, ids and ports of each replica is specified in **replicas.csv**, which can be changed to make it easy to scale the number of replicas.

The primary code is implemented in **primary.go**. As a primary, the binary initializes an HTTPServer that the client will connect to and also connects to all the backup replicas. It processes incoming requests from the IRQ and synchronize with the backups. The backup code is implemented in **backup.go**, which implements an HTTPServer to communicate with the primary. The remaining logic is self-explanatory, including **oplog.go**, which implements the operation log, **table.go**, which implements the client table, the **rpc** module, which describes the RPC protocols, **recovery.go**, which performs a recovery, and **globals.go** and **flags.go**, which define the information shared between modes on a replica.

6 Experiments and Results

We measured the performance of the system under various conditions. Those experiments were performed on a Macbook Air 13-inch, with nodes running in separate processes.

The first experiment we performed was to see how transaction time changed as a function of the number of nodes running VR (Figure 4). The conditions were

- There is only one client and the client sends requests at a rate at 2 QPS
- The size of the primary Incoming Request Queue (IRQ) is 5, which means that the client can queue up to 5 incoming requests until it does not accept client requests anymore

We saw an increase in the amount of time it took a client to get a response as the number of nodes grew, likely because the number of nodes the primary had to wait on for quorum increased. Additionally, as these experiments were running on a single machine, there was likely a performance hit for increasing the number of running processes. Also note that over time, we do not see any significant increase in transaction time. This is likely due to the fact that the number of messages and size of messages do not change along with time. More experiments can be done to explore the maximum size of messages and the minimum size of the primary IRQ before the system becomes unavailable.

We also considered how the time for a view change varied as the number of client requests grew. The conditions of this experiment were

- A view times out after 5s
- The client sends 50 requests per view at a rate of 10 QPS

Over time, as the client sends more messages, the size of the operation logs maintained by each replicas grows. This means that when a view change or recovery happen, there is a constantly growing amount of data to exchange, which increases latency of the system. Despite our concurrency optimizations, there are places in the data path that require blocking until processing is complete, and so this behavior is unavoidable without additional optimizations (mentioned at the end of the Liskov paper). More experiments can be done to explore the tradeoffs between the size of the operation logs and the number of additional messages to be exchanged if the optimizations mentioned in the Liskov paper were implemented.

Last but not the least, the experiments mentioned above can be performed on multiple physical machines in a network to produce more realistic results.

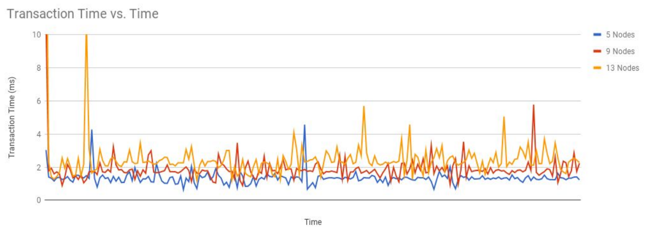

Figure 4: Transaction time as # of replicas change.

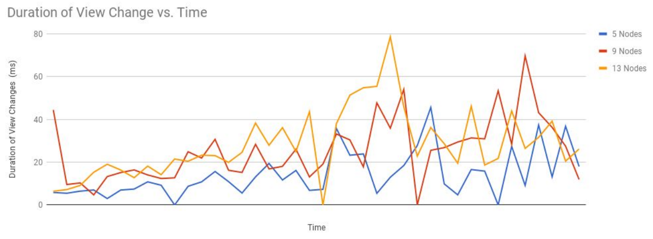

Figure 5: View change time as client requests increase.

7 Conclusions

There are many ideas the paper proposes that would useful features or improvements to the protocol. For example, an important feature would be to add the reconfiguration protocol that specifies how the replica group can change through addition or removal of nodes (Section 7). There are also many optimizations the paper proposes in Section 6, such as using witnesses to supplement the backups and batching requests to increase throughput during heavy load.

Beyond the ideas suggested, it would be very interesting to implement a service such as Chubby on top of this system. Ideally, we could run this VR implementation across multiple replicas, create a higher-level application running on top, and perform experiments measuring its performance more rigorously.

While the paper provided a clear and simple framework to follow, there are many implementation choices we came up with (e.g. how to indicate to client who the new primary was after a view change). We found that focused pair programming was the most effective way to avoid the many of the traps that are part of building such a complex protocol. Specifically, there were many edge cases to consider (e.g. what conditions may cause our view change to malfunction) and multi-threading pitfalls to be aware of (e.g. locking functions that could disrupt another thread, without causing a deadlock).

This project required a thorough understanding of the VR protocol and a meticulously thought-out design. By carefully reading the paper, taking advantage of Go’s strengths, and preferring simplicity to complexity, we were able to build a working, robust implementation of the VR protocol that is extensible to support new applications. Overall, working on this project was an excellent learning experience that allowed us to deeply understand the viewstamped replication protocol.

8 References

1. Liskov, B., and Cowling, J. Viewstamped replication revisited. Tech. Rep. MIT-CSAIL-TR-2012-021, MIT, July 2012.
2. "Viewstamped Replication Revisited | the morning paper." 6 Mar. 2015, https://blog.acolyer.org/2015/03/06/viewstamped-replication-revisited/. Accessed 13 Dec. 2017.