Project Landslide: RocksDB + RAFT

Parvinderjit Singh, Angelo Kastroulis
{psingh,akastroulis}@carrera.io
Carrera Group

ABSTRACT

RocksDB is a LSM key-value store capable of harnessing multiple cores for processing transactions and provides high performance on I/O bound operations. The aim of this paper is to present the design, implementation and performance characteristics of a distributed replication solution based loosely upon the RAFT algorithm built within RocksDB. This particular implementation is currently restricted to storing and fetching integer keys and values.

1 INTRODUCTION

Distributed computing has become the de facto computing paradigm underlying the architecture of most new large-scale high performance applications. Any non-trivial application will consist of a data systems layer that needs to be robust, efficient and fault tolerant. In addition to these requirements, a distributed data system must also be able to replicate transactions across multiple instances spread across multiple computing nodes. This last requirement enables the distributed application to remain highly available and consistent across vast geographic distances.

RocksDB is an embedded database solution that checks off most of these requirements in the form of an open source library that can be integrated into any application. The missing element within rocksdb is the replication of transactions across multiple instances. This lack of replication limits the applications into which rocksdb can be integrated. Existing applications have relied on incorporating third party libraries or tools in order to simulate the functionality of a replicated data system with rocksdb.

The aim of this paper is to reduce the complexity of a distributed data system by adding a replication component to RocksDB. In order to accomplish this, pieces of the RAFT algorithm particular to replication have been implemented within RocksDB, which allows for sharing of resources between the replication component and the rest of the database. The RAFT algorithm describes a replicated state machine operating across multiple servers to keep a distributed data system consistent.

2 BACKGROUND AND RELATED WORK

The Raft algorithm is a consensus algorithm for distributed systems built primarily for ease of comprehension and implementation. The Raft algorithm details two key pieces of functionality that are core to distributed data systems: cluster management and transaction replication. Cluster management is the process of managing cluster membership and ensuring the availability and reliability of a cluster. Transaction replication is the process of replicating a transaction from a source system to the rest of the nodes in the cluster to ensure durability and consistency.

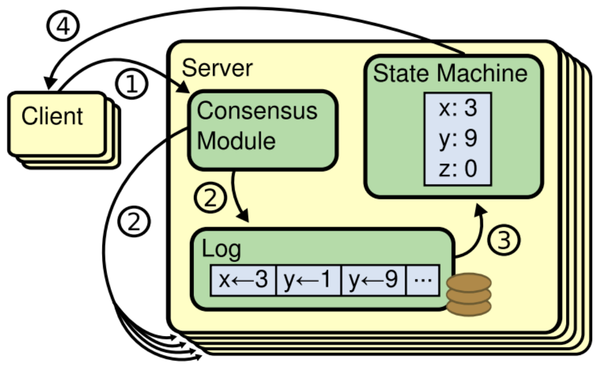

Figure 1: Replicated State Machine

2.1 Transaction Replication

Transaction replication is the process of replicating transactions or requests that modify the state of the data system to other nodes in the cluster. The transaction replication aspect of Raft is implemented using a replicated state machine architecture. The replicated state machine consists of three distinct modules as depicted in Figure 1.

Each server in the cluster contains it's own replicated state machine with these three modules. The consensus module is responsible for ingesting client requests, writing them to the local log and passing the request on to the consensus modules in the other server's to process. Raft describes a master-slave architecture for replication where the master is denoted as the leader and slaves are denoted as followers. The consensus module on the leader is the only one that can accept external client requests and the consensus modules on the slave nodes can only accept requests from the leader.

The replicated log in an indexed log of all requests that have been received by the cluster that are in either a committed or uncommitted state. Only the leader in the cluster can add new requests to the log and then replicate to the other nodes in the cluster. A new request is added to the log in an uncommitted state and can only be moved to a committed state once the request has been replicated to a majority of the other servers in the cluster. A committed request is then available for the state machine to process.

All requests in all the logs occur at the same index across the cluster and all the logs are ordered in the same way. This ensures that when the state machine processes requests, it processes them in the same order and therefore the state of the data system is consistent across all of the nodes in the cluster.

2.2 Cluster Management

Raft lays out an elegant cluster management algorithm that ensures there is always one strong leader present to manage the cluster and replicate transactions. The leader is selected through an elections process that ensures only a candidate server with the most up-to-date log can be elected as the leader.

Each server within a Raft cluster starts off as a follower with a randomized timer. If the timer reaches zero before the server receives a request from a leader, the server will kick of an election by converting to a candidate and sending out a request for votes to all the other servers in the cluster. Once a candidate receives a vote from a majority of the nodes in the cluster, it converts to a leader and starts sending heartbeat requests to all nodes in the cluster to prevent another from starting a new election.

If the leader node for some reason subsequently goes down, another server will start the election process once it's randomized timer runs down without having received any requests from the leader. A new server can be added to an existing cluster at any time and it will stay in a follower state once it starts receiving heartbeat requests from the leader of the cluster. The leader of the cluster receives the current index of the logs for all followers in their response back to the heartbeat and therefore can replicate data as needed to bring the servers up-to-date.

3 METHODOLOGY/DESIGN

The implementation completed as part of this paper involved only the replicated transaction process of Raft. The Raft paper lays out detailed experimental results for the cluster management process. Therefore in this paper, the focus is on the performance results for the replicated log and the consensus module.

3.1 Cluster Definition

The implementation of Raft done within RocksDb only included the replicated transactions, therefore a simple configuration file was defined for identifying the nodes in the cluster. An example configuration file is shown in listing 1. This file would be provided as input to a rocksdb instance, along with a number identifying the configuration line to applied to the current instance.

Listing 1: "Cluster Configuration File"

```
leader 5005 5006
slave1 5007
slave2 5008
slave3 5009
slave4 5010
slave5 5011
```

3.2 Consensus Module

The consensus module on the leader is implemented as a server that accepts requests via existing rocksDb interfaces such as Write and transmits these requests to other nodes using TCP. Using this architecture minimizes the changes from the client's perspective. When a client invokes a write call on rocksDb, the call is first passed to the consensus module, which copies the request into the local replicated log and then allows rocksDb to commit the transatcion to the database.

Every 100 milliseconds a thread copies up to ten thousand entries from the log and transmits them to the follower nodes, along with the last commit index on the leader. The protocol used to transmit this data is TCP. Once a follower receives this request with the data, it copies the data into its local log and passes the requests to the local db. Then the follower will return a positive acknowledgement to the leader. The leader waits for a positive acknowledgement from all the followers before sleeping again for 100 milliseconds and starting the process all over again.

3.3 Replicated Log

The replicated log is implemented as a vector that maintains the order of the requests across all the nodes. This is ensured by the serial nature of the requests being sent by the leader to all of the followers. The leader maintains the highest know copied request for each follower and will transmit up to ten thousand transactions in each copy entries call from the highest known index. The fact that the data is immediately committed as soon as it's received on both the followers and the leader eliminates the need for a state machine to process the transactions. This does violate the safety guarantee of Raft because a transaction that is not durable is committed and a subsequent crash of the leader before the transaction is replicated to the other nodes means that only the leader will have this transaction.

4 EVALUATION

4.1 Test Setup

In order to test the performance of this implementation, two experiments were performed with 6 nodes, all running on a single machine and several timing metrics were captured and evaluated. The first test setup consisted of one thread running from 1,000 to 100,000 write operations as quickly as possible with no wait between writes and a second thread reading every ten microseconds (μs). The second experiment had a thread on each slave node reading every millisecond (ms) and the same setup as experiment one for the leader, except the number of writes went from 1,000 to 29,000.

4.2 Test Environment

Both experiments were performed on a Macbook Pro with 2.6 GHz 6-Core Intel Core i7 processors, 16 GB memory and 500 GB SSD. Raft implementation was done within RocksDb library code in C++ and the performance tests were also written in C++, making direct use of RocksDb interfaces.

4.3 Results

The chart in Figure 2 details the results of the first experiment. This experiment attempted to measure the write performance that a client would experience on the leader node. The chart shows that there is very little impact experienced by the client by the addition of the replication functionality within RocksDb. The leader is capable of ingesting over a 100,000 transactions in under a second. There is a steady rise in as the number of transactions increases until about 87,000 transactions, after which there is a dramatic drop-off and then a steep rise again as the number of transactions continue increasing.

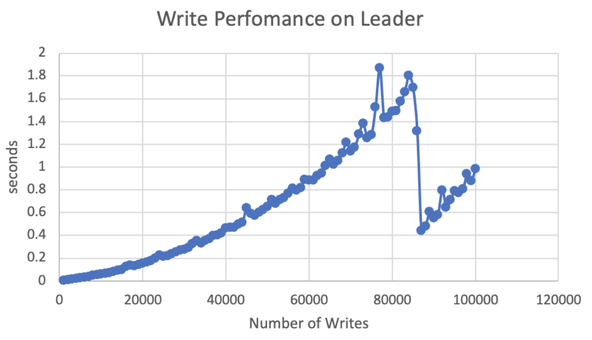

Figure 2: Write ingestion performance on leader node

The chart in Figure 3 shows the number of seconds it took to propagate transactions to the slave servers as a function of the number of transactions. The curves for each of the slaves are right on top of each other, showing that the slave servers each receive and process the transactions concurrently and within a relatively short time of each other. The time required to propagate all transactions appears to rise exponentially as the number of transactions increases.

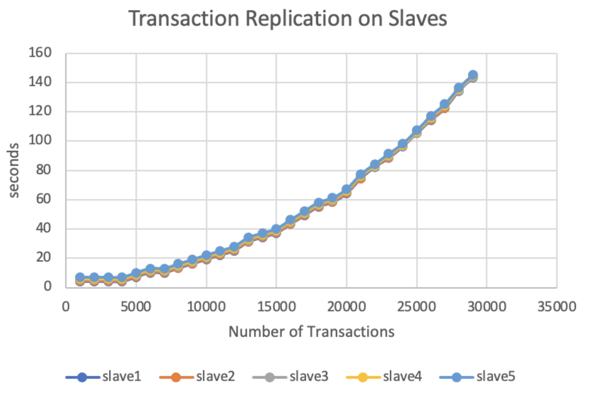

Figure 3: Transaction replication performance to slaves.

The following three figures show the difference in processing times between the different slave nodes at different transaction loads. The curves for all three graphs are pretty similar to each other, revealing that the performance characteristics of the slave nodes stay relatively static as compared to each other irrelevant of the transaction load applied to them.

5 CONCLUSION

This paper demonstrates the viability of the transaction replication aspect of Raft being implemented in RocksDb. The performance metrics presented in the previous sections show that adding Raft to RocksDb does not negatively impact the performance of Raft from the clients perspective, if the client is limited to writing to the leader only. Raft can be implemented to provide eventual consistency in the background while minimally impacting client workflows. In

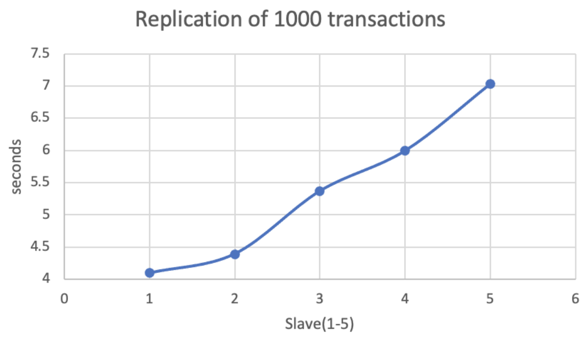

Figure 4: Comparison on replication performance between slaves for 1000 transactions.

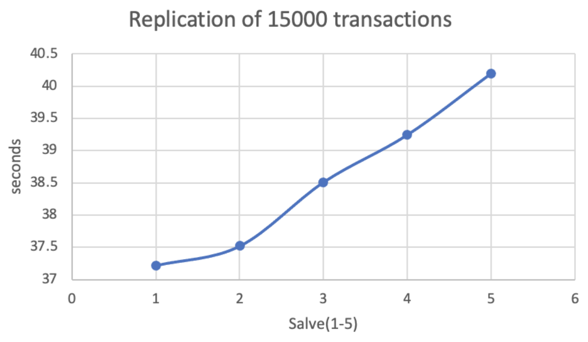

Figure 5: Comparison on replication performance between slaves for 15000 transactions.

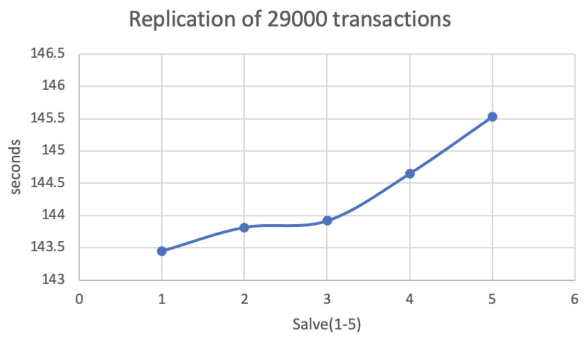

Figure 6: Comparison on replication performance between slaves for 29000 transactions.

addition, this approach to setting up a distributed data system hides the complexity of the distributive system to the user behind the familiar interfaces of RocksDb.

5.1 Subsequent Work

The work as presented in here is incomplete, since the Raft implementation does not adequately handle cluster management and is not therefore durable to failures. Therefore, future work should endeavor to fill this hole by implementing the rest of Raft or using a third party solution to fulfill this role. In addition, there are several performance improvements that can be pursued within the code to aid in propagation. One of these is asynchronously processing transactions on the slave after receiving them from the leader, rather that processing them synchronously and tying up the leader in the meantime.

REFERENCES

A APPENDIX