zk-rcu : A RCU recipe for Zookeeper

Tejaswi Tanikella Rohit Kumar

ABSTRACT

We present a recipe for acheiving RCU like wait free reader writer synchronization mechanism over Zookeeper. The recipe builds over the raw API provided by Zookeeper to allow readers to access data with very little overhead, while allowing writers to write without having to wait for the readers to complete. The recipe has been then used to build a filesystem which exposes RCU-like API to users to read and write to files. To compare the performance, another recipe to implement Readers-Writers Lock has been presented. Both the recipes and the filesystem have been evaluated by building a Text Processing Application and the results have been presented.

1 INTRODUCTION

Readers and Writers synchronizing a shared resource is classical problem that has been well studied and numerous solutions have been proposed. For a special case of reader writers where the number of readers is much larger than the writers, Read-Copy-Update (RCU) has been shown to perform extremely well. RCU allows readers to read the data without significant synchronization overhead and allows the writers to write data without having to wait for the readers.

Zookeeper[5] provides APIs to build solutions for distributed synchronization which can be used to enforce the requirements of an application. In this project, we aim to build a RCU-like recipe through Zookeeper. The next section discusses the design of the RCU recipe and the Reader Writer Lock recipe that has been used to compare the RCU recipe's effectiveness. Next the recipe has been evaluated by building a Text processing application on top of a filesystem that uses the RCU recipe.

1.1 RCU

RCU [7][4] as a synchronization mechanism has been well studied and implemented both in kernel and in userspace. In the current section we re-iterate some of the basic tenets of this mechanism and describe how similar semantics would help build a distributed application.

Theoretically RCU divides variables into Live and Dead Variables. Live variables contain values that will be used later and hence influence the future. Dead variables contain values that will no longer be used and hence have no influence on the future. Any Live variable over the course of time becomes dead once the program stop using it. Variables can then also be classified into Temporary and Permanent Variables. Temporary variables are variables that are live only inside a critical section. A Permanent variable on the other hand is live outside a critical section. Essentially the key idea is that after a grace period, once all Temporary variables are dead, references to an object no longer exist. At this point in time, called the Quiescent state, the variable can safely be modified. When implemented in a kernel, the quiescent state occurs during a softirq, when the new references are made and old references are deleted. A slightly different version can be built where an all knowing entity can track the versions that are being used and can atomically add and delete references. The entity can delete the reference once the the last reader's reference becomes dead. The entity can add a reference under CAS like conditions. This is the version which we have chosen to build as an RCU recipe.

RCU semantics promise wait free access to read operations. This lets read heavy applications have significant efficiency with little overhead when compared other synchronization solutions. For write operations, RCU semantics let the user apply the changes in one operation. RCU is ideal for read heavy scenarios which need fast reads.

2 RELATED WORK

There are multiple synchronization primitives used for distributed systems, a brief survey of which we have provided here.

For the application targeted in our project, we have looked at the various locking mechanisms and their use cases. Zookeeper recipes for building distributed locks for global synchronization already exist as well known libraries, which have the cost of acquiring both read and write locks.

Other distributed locking mechanisms such as the "Chubby lock service"[2] for loosely coupled distributed systems. The "Scalable reader-writer locks" is another instance of locking mechanism designed for mostly read-only operations in a distributed environment. A similar workflow with multiple instances of data being present in the system is that of read-only snapshots implemented in databases. The main tenet for having this functionality is more about redundancy and fault-tolerance rather than the concurrent read-write problem in our intended application. The read-only snapshots as the name suggest are captures of the data in the system at certain fixed intervals wich can be restored if needed so. There is not any accompanying requirement of keeping track of which instance of data should be read from or written to as they always deal with the latest instance.

The distributed shared memory as implemented in the IVY[6] system provides a ordering of the operations based upon the write operations. This behavior provides to us an idea of using events to achieve sequential consistency. The idea behind a total ordering of events given by the write operations could be extended into the RCU-recipe wherein a bunch of read operations interspersed between the write operations could be handled similarly.

3 DESIGN

In this section we describe the design of the components built for the project. The first subsection briefly describes the Zookeeper APIs and their behaviour which was critical for building the RCU recipe. It is followed by descriptions of the RCU recipe and the Readers-Writer Lock recipe.

3.1 Zookeeper

Zookeeper is a distributed coordination serivice that can be used as a basic block to build higher forms of synchronization. The service has been used to build the recipes described in later subsections. The calls that Zookeeper provides are Create, Delete, Children, Exists, Set and Get. We discuss a few functionalities that Zookeeper provides that allows building synchronization recipes. It does not discuss the internal working of Zookeeper which facilitate these functionalities.

The basic building block of any synchronization mechanism is an Atomic compare and swap command. In CPU hardware, the CAS instruction which atomically compares the given value with the current value, and only on match sets it to a new value, provides the basis on top of which all sychronization primitives can be built. Zookeeper provides the Set() API which along with file versioning becomes the basic block over which distributed synchronization solutions can be built.

The Set call takes the new data and a version which the user expects the data to be at. If the version provided by the user does not match the version Zookeeper has, the change is rejected. The version number thus provides the atomicity guarantees similar to those that CAS provides. Zookeeper also allows users to override this check by using the version number -1.

The Delete call, similar to Set call takes a version, which should match for Zookeeper to delete the file. In addition to the version check, Zookeeper also enforces that the file cannot not have children for a deletion to proceed. Only a file without any children can be deleted.

In addition to the five methods provided by Zookeeper also provides variants of the methods that notify the user on modifications. These variants allow the user to wait for changes instead of polling Zookeeper for changes. These methods allow building distributed Mutexes instead of distributed spinlocks.

The final feature that is critical to building the recipes described below are Sequential and Ephemeral files. Ephemeral files only exist till the owner of the file is in contact with the zookeeper. For all other users, disconnection of the owner and file deletion by owner both can be handled simply as the disappearance of the Ephemeral file.

Sequential files, with a monotonically increasing suffix, provide a quick way to enforce total ordering on events occurring at the same time. Ephemeral and Sequential files are used by synchronization recipes to construct a queue of users waiting for the resource, which by design can handle disconnection and failure of users.

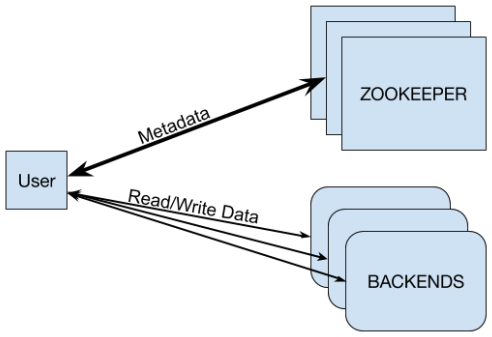

Figure 1: Design of a system built using Zookeeper. Zookeeper maintains the metadata, which provide information about the location of the data which is stored in other backend servers.

Zookeeper, like shown in Figure 1 is used to store the metadata that a user needs. The data is stored elsewhere, whose location is provided by the metadata. In the subsequent sections, which describe the recipe, the metadata needs to synchronized against parallel access.

3.2 RCU Recipe Design

The design goals behind creating a RCU Recipe are:

(1) Wait Free Reads: The reader should not have to wait for a writer to complete writing. Metadata read by the user needs to be as upto date as possible. Strict guarantees that the metadata must be the latest requires additional locking and hence waiting, which we have decided not to pursue.
(2) Low Latency Writes: Writers should be able to quickly make a change. The change might not succeed since the update is out of date, but writer should not have to wait for readers to make changes.

The design goals described above are inspired by those that classical RCU chooses and thus, the methods that the

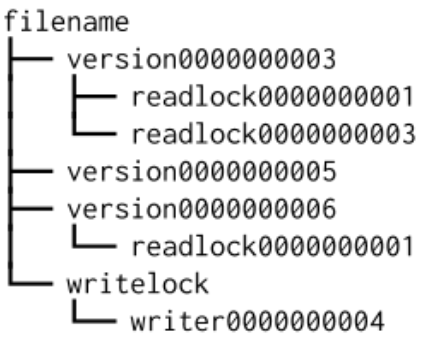

Figure 2: The above figure is a sample snapshot of the structure built by the RCU recipe. In the above structure three versions are available, versions 3, 5 and 6. Version 3 is the oldest among them, with two readers using the metadata stored in the file "/filename/versin0000000003". Version 5 has no active readers and will be deleted by the RCU Garbage Collector. Version 6 is the latest version with one reader. One writer is present in the writelock directory, i.e. currently owns the lock and might be able to add a new version.

recipe provides are also similar to those of RCU. The RCU recipe provides access to metadata stored on Zookeeper via the following APIs:

(1) `Create_RCU_resource()` : This function creates the necessary files and structures in to allow the user to Assign and Dereference metadata from Zookeeper.
(2) `Read_Lock()/Read_Unlock()` : These function calls mark the start and end of the reader's RCU section. Within these boundaries the reader is guaranteed that the metadata being read from Zookeeper would not change.
(3) `Dereference()`: This function should be called within a RCU section and is the safe way to read metadata from Zookeeper. This call will return the same information within a RCU section.
(4) `Assign()`: This function should be called outside a RCU section and allows the user to replace Zookeeper's metadata with the one provided by the user.

The user's first call `Create_RCU_resource()` first creates a main file for the resource named based on the input string. It then creates a sub directory for write locks and creates an empty "version000000000" file which is empty.

On an Assign() call, the recipe creates newer version file and attaches the provided metadata to the newly created file. This file is created using the Sequential flag, which guarantees a total ordering of the version numbers. The Assign call also lets the user pass a version, which they expect to replace. If the version number provided by the user does not matches the latest version of the resource, the Assign() call fails. Similar to Zookeeper's Set call, Assign()'s version number checks are overridden if the version number provided by the user is -1. Since the Assign() call first needs to check the latest version and then create a newer version file, without a lock, the Assign call is not atomic. The write lock makes the check and file creation an atomic step, preventing any race between simultaneous Assign calls.

Each Read_Lock() call finds the latest version that is available and creates a Sequential and Ephimeral file under that version. This call involves calling Children call to find the latest version and then calling Create() to create a new file. This step is not atomic and the latest version that the user read could disappear by the time it calls the Create call. If the version disappears, we report an error to the user, who can retry acquiring the Read_lock. We have explicitly chosen not to acquire a lock and rather choose to optimistically try and fail. The scenarios where RCU should be used, like mentioned earlier, are ones with greater number of readers than writers, hence the number of such failures are low. This has been discussed more in the subsection where we describe the design of the RCU garbage collector. The Read_Unlock() simply deletes the file created by Read_lock().

Within a RCU section, the user calls Dereference() which reads the metadata of the version which the user has locked. This is simply a single call reading the metadata from Zookeeper. Since Zookeeper stores metadata and not the actual data. The actual data might be stored elsewhere on backend servers. Events in the backend servers, like failures and disconnections, can cause the metadata to change. But the actual data that the metadata points to remains the same. Hence though multiple Dereference calls wont return the same metadata, the information that the metadata points to remains the same across Dereference calls.

3.2.1 RCU recipe - Garbage Collection

In the original RCU literature, resources can be freed when the system reaches a quiescent state. In our case, a seperate method is provided by the RCU recipe, `Rcu_gc()`, which identifies directories which have a structure similar to that of those created by the recipe, finds versions which do not have children and deletes them.

It is very obvious that a reader creating and the GC deleting the version could run into race conditions. But here the safety provided by Zookeeper has been used to ensure safety without resorting to locks. Like mentioned earlier the delete call deletes a directory only if the file does not have any children. If the GC finds the version empty and a reader creates a reader file inside immediately after, the delete operation would fail. On the other hand if the reader, while trying to

acquire a read lock identifies a file as the latest version and the GC deletes the file, the create call called by the reader would fail. The reader can retry if the readlock fails in this way.

The method provided by the recipe only runs one cycle of cleanup. This method can be called with periodicity that the application developer chooses. The periodicity can be chosen so the overhead of garbage collection and the overhead of old version files are balanced.

3.2.2 *RCU recipe - Use Cases*

The recipe described above should only be used in the following scenarios:

(1) Number of writers should be much smaller than the number of readers. Once the number of writers increase, the Assign() calls would start failing, causing writers to retry. In case the number of writers are greater, using Simple Distributed Locks is advisable.
(2) The data on which the user needs to operate on is large. If the data on which the reader operates is small, the reader can simply fetch the metadata from Zookeeper, pull the data and cache it locally. Multiple read calls and guarantees of data not changing are not necessary since the local cache provides all such guarantees.
(3) Out of date data should be valid till all the users complete reading the data. Thus the garbage collection needed on the backends depends on the garbage collection that the recipe does.

In scenarios where these do not hold, the performance of RCU, as shown in subsequent sections, will begin to degrade.

3.3 Readers-Writer Lock (RW Lock) Recipe Design

The RWLock recipe is an implementation which provides special locks to readers and writer which allows multiple readers to read the data and ensures only on writer makes modifications. The Readers-Writer Lock Recipe synchronizes access to metadata stored on Zookeeper via the following APIs:

(1) Create_RWLock(): This function only initializes the data structures, and creates a single directory to enqueue all readers and writers.
(2) `Read_Lock()/Read_Unlock()`: These mark the region where reader is allowed to read the resource. Multiple readers are allowed to simultaneously read the metadata.
(3) `Write_Lock()/Write_Unlock()`: These mark the region where the writer can make changes to the resource. Only a single writer is allowed to write.

The recipe is inspired by way IVY handles reads and writes. IVY defines views where multiple readers can read data. A write marks the end of an old view and the start of a new view. Each user reader or writer creates a sequential file in the directory with the corresponding prefix. A reader creates a reader file and a writer creates a writer file, but the sequence numbers assigned by Zookeeper enforces a total ordering of the creations. Once the files are created, the resource can be read or written to. If the file with lowest version number is a reader file, the owner of the file and all subsequent reader files are allowed to read the file. But if the file with the lowest version number is a writer file, only the owner of the file gains access and can modify the resource data.

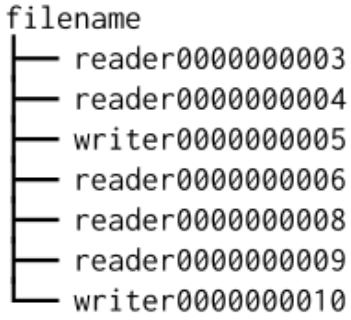

Figure 3: The above figure is a sample snapshot of the structure built by the Readers Writers Lock recipe. The queue has 7 nodes. Readers 3 and 4 form a view and have read access to the metadata stored in "/filename". The next view is formed by readers 6, 8 and 9 seperated by writers 6 and 10. All nodes except for reader3 and reader 4 are waiting to gain access to the metadata.

An advantage of this design is that both readers and writers are given equal chance to access the resource. A user that is added to the queue is guaranteed access to the resource eventually, given all participants access the resource for a finite time. An alternate design that gives higher priority to writers can cause starvation to the readers in certain scenarios. But the disadvantage with dividing into views is the delay that writers incur since they have to wait for all readers ahead before making the change.

The original paper presenting Zookeeper describes problems with naive lock implementations. To avoid the herd effect described there, a reader waits on the writer a version number just lower than it's number to complete. Similarly, a writer waits for all readers in the view just ahead of the writer to complete.

4 EVALUATION

To demonstrate an application that uses the RCU recipe, a sample text processing application was built which was built on top of a File system that provided RCU like bindings to read and write files. The design of the fileystem and the application have been described in the subsections below. A subsection also describes the metrics that were used to measure performance of the recipes and the applications built using the recipes.

4.1 File system Design

The design goals of the filesystem were:

(1) Allows Readers and Writers to run simultaneously using underlying synchronization to guarantee correctness.
(2) Operations on large files are facilitated. Similar to the Map Reduce [3][1] paradigm, the operations are run on remote servers and the results are aggregated at the user.

The filesystem was built to expose RCU-like bindings which allow the user to run multiple reads while guaranteeing that the data the user views does not change while the user holds the read lock. The filesystem exposes the following interface:

(1) `Open()`: This method establishes a connection to Zookeeper and creates a file within which the synchronization primitives can create files.
(2) `Read_lock()/Read_unlock()`: These calls mark the region where the user is guaranteed the metadata does not change.
(3) `Read_op()`: This method runs a read call on the resource. The filename and the operation need to be provided, the operation is run on the backend servers and the results are returned to the client.
(4) `Write()`: This method writes the metadata to the backend servers and returns metadata. The call does not involve the Zookeeper.
(5) `Write_meta()`: The method is called after a write call, to push the metadata to the Zookeeper.
(6) `Close()`: This method releases the resources and closes the connection to the Zookeeper.

The metadata is stored on Zookeeper and the file data is divided into multiple shards and is stored in backend servers. Eight AWS instances were used to run the backend servers. Each shard is identified by the SHA256 hash over the shard. The metadata contains the hash and the backend server's address, that is encoded into JSON and stored in Zookeeper. Three zookeepeer nodes were run on AWS.

The Read calls run operations on the shard at the backend server and the results are returned and are aggregated by filesystem on the user's machine. This lets the user work on large files by distributing the computation. In addition to Read_op and Write an additional method Write_op is provided which runs an operation and writes the result to a local shard. This method lets the user transform data without having to pull the data.

The filesystem's methods have been designed for resources with multiple readers and writers trying to access the underlying resources. The Open call takes the synchronization mechanism as an argument and initializes it. The filesystem can be configured to take one of the following synchronization mechanisms:

(1) RCU : The recipe the paper presents.
(2) RWLock : The recipe described in an earlier section.
(3) Distributed Mutex: A single Distributed Lock which both the reader and writer share.
(4) Unprotected: This solution does not provide any synchronization, but has been used to evaluate the filesystem.

4.2 Sample Text Processing Application

A Sample Application was built over the filesystem described above. The application's objectives were:

(1) Given a file, find and replace instances of blacklisted words. Readers read, find errors and make corrections to the text file.
(2) Allow processing large text files. This is achieved by building on top of the filesystem described earlier.

Before measuring the application's performance, we measured the performance of the individual components, which is presented in the following sections.

4.3 Zookeeper Method Microbenchmarks

The throughput of four methods that are used in the recipes have been benchmarked and have been presented in Table 1. We measured these numbers to understand the relative cost of each of the calls and accordingly make choices to build the recipe.

Operation	Time (ms)
Create	1233.998
Get	341.782
Set	993.520
Delete	1046.106

Table 1: Zookeper Micro Benchmarks

The results show that all the calls have nearly equal throughputs, except for the Get method. Since zookeeper can provide Get results directly without having to contact the other

zookeeper instances, it is expected that the Get throughput is much higher, and the results show the same.

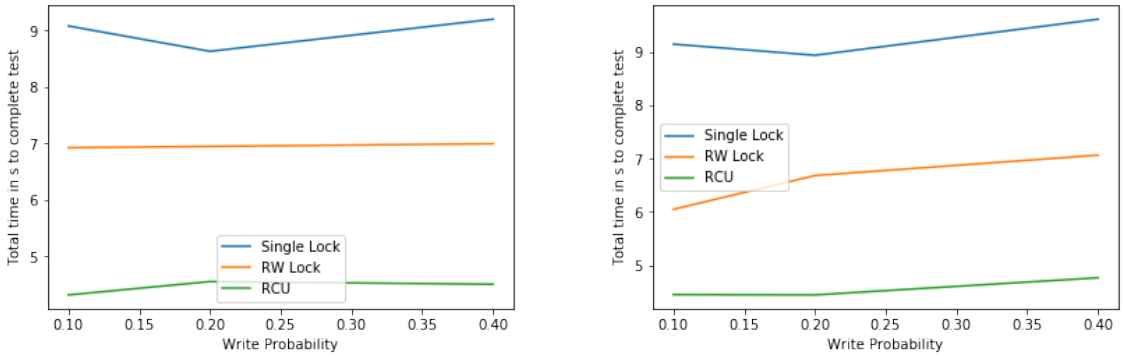

Figure 4: Random Read Write Test. The graphs show the time each recipe takes to complete 500 calls, as the write probability changes from 0.1 to 0.4. The plots present the behaviour with 0 and 0.25 lock hold probability respectively.

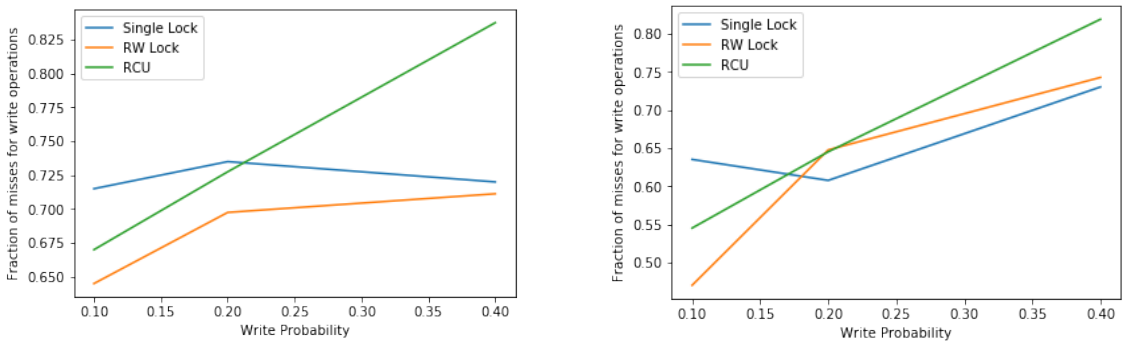

Figure 5: Random Read Write Test. The graphs show the fraction of the writes that fail due to mismatching versions, as the write probability changes from 0.1 to 0.4 The plots present the behaviour with 0 and 0.25 lock hold probability respectively.

4.4 Random Read Write testing

The filesystem was evaluated by building a test application that called the Read_op and Write methods of the filesystem. The evaluation also provided insight into the relative performance of the synchronization recipes. The test application takes in three value as inputs:

(1) Write Probability: The probability of the application calling the write method. It determines the number of write calls the application makes during testing.
(2) Read Hold Probability: The probability that the application would continue holding a Read lock and run multiple read transactions within a read section.
(3) Number of Method calls: The total number of calls to Read or Write.

The test application would use the above parameters to build a chain of transactions, which it would begin running. This application was used to measure the time it takes to complete 500 method calls. Four instances of the test application were used in parallel and the results have been presented in table 2.

Lock Mechanism	Time (s)
RCU	4.405707
Single Lock	8.868124
RW Lock	4.517152
No Lock	4.43437s

Table 2: Time to complete 2000 read calls without any writes. RCU and RWLock perform close to Reading without synchronization, validating their design.

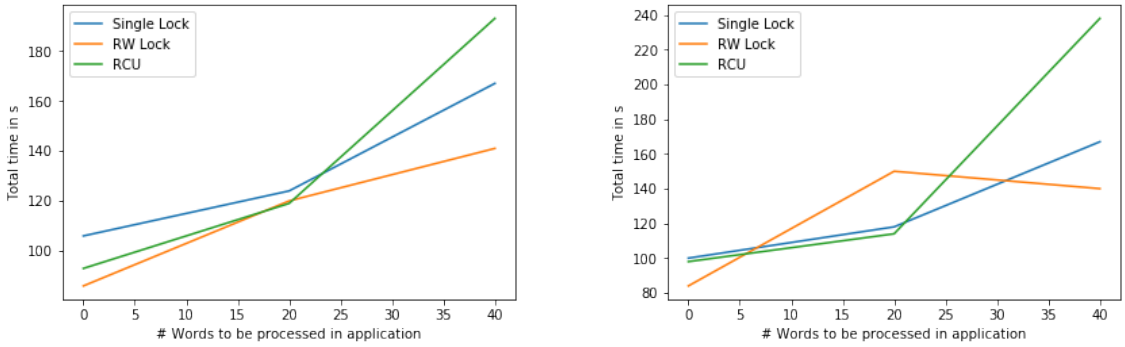

Figure 6: Run to make corrections using a filesystem that used Single Lock, RW Lock and RCU to synchronize data. The plot show the time it takes as the number of corrections are increased from 0 to 40. The plots are for 10 and 20 parallel clients respectively.

We can make four observations from the results presented in Figures 4 and 5. Firstly for pure reads i.e. write probability of 0, the performance of RCU and RWLock match the results obtained without any Locks. This is encouraging, since it implies that the implementations have very low overhead for read calls. On the other hand, as expected the since the Single Distributed Mutex serializes the requests, takes longer to complete.

Secondly, with write probability increased to 0.1, the RCU recipe performs better when compared to RWLock. This is because while the RCU recipe allows multiple readers to read, the RWLock recipe divides the readers into views and makes them wait for writers to complete. Thus, this result is proof that the RCU recipe performs better for cases with low reads.

Thirdly, as the write probability increases we see increasing failures for all recipes due to version conflicts. This is again expected since, as the writes increase the chance of a reader reading an old version increases.

Finally, as the write probability increases the number of successful writes that RCU has completed decreases. This is due to RCU's optimisic try and fail behaviour that makes it prone to higher number of failures as the number of writes increase. It can also observed that the SingleLock performs greater successful writes but takes longer to complete the transactions. Finally the RWLock takes the middle path, taking longer than RCU but completes the run with fewer conflicts.

4.5 Application Benchmarks

A large text file of size 238MiB, with 4,764,563 lines, was generated by downloading files from Project Gutenberg. A list of words that are spelled differently in US and UK English was chosen as the data that will be corrected in the file. After all the corrections are made, the final file should not have words that appear in the list. To regulate the ratio of writes to reads, out of the 1000 proposed changes, the number of words that actually were present in the file was varied from zero to 40. That is, out of the 1000 words only P words would be found in the file, and these instances would be corrected. Each client, chooses a disjoint set of words from the list and starts searching for occurrences of the word, and makes the correction if found. The time it takes N such clients to go through the file and make edits has been chosen as metric for evaluating the application.

Firstly achieving the same results on a single machine by a single user on a single AWS instance using grep and sed takes around 4 minutes to complete. The results show some form of improvement in the numbers, which is encouraging.

We can make two observations from the results shown in Figure 6. Firstly, the RCU recipe takes longer than the Distributed Mutex once the total number of words that need to modified is more than 20 out of the 1000 words. This once again shows the poor performance that RCU provides once the number of writes increases.

The second observation is that RWLock provides better performance if more than 20 words need to corrected. The RWLock recipe completes the task faster than both the RCU and Mutex recipes in scenarios with 10 and 20 threads.

5 CONCLUSION AND FUTURE WORK

From the results and discussions in the previous section we can draw the following conclusions:

(1) The RCU recipe works in scenarios where the number of write operations are very few as compared to the number of write operations.
(2) As the number of write operations increase, the RCU recipe's performance degrades. Its optimistic approach of reading an older version becomes the bottleneck. As the number of write operations increase, the number

failed writes increase, thus caused increased number of transactions, degrading the performance.

(3) The Reader Writer Lock, built to compare RCU's performance performance better than the RCU recipe for higher number of writes.

Two major defeciencies emerged while we built the system. Firstly the performance of the backends was not optimal. Instead of choosing to build functionality within the backend servers, we chose to let the backend run applications like grep and sed to process data. This was done as an experiment and also due to lack of time. As future work we need to fine tune the filesystem performance and then evaluate the sample application's performance.

Secondly, to make sure code is common between the recipes, we chose to build the filesystem with RCU like bindings that uses RWLock underneath to synchronize. Providing the RWLock like bindings itself might have allowed better performance. This is something we need to tackle as well.

REFERENCES

[1] Dhruba Borthakur et al. 2008. HDFS architecture guide. *Hadoop Apache Project* 53, 1-13 (2008), 2.

[2] Mike Burrows. 2006. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*. 335–350.

[3] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[4] Mathieu Desnoyers, Paul E. McKenney, Alan Stern, Michel R. Dagenais, and Jonathan Walpole. 2012. User-Level Implementations of Read-Copy Update. *IEEE Transactions on Parallel and Distributed Systems* 23 (2012), 375–382. https://doi.org/10.1109/TPDS.2011.159

[5] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free Coordination for Internet-scale Systems.. In *USENIX annual technical conference*, Vol. 8.

[6] Kai Li and Paul Hudak. 1989. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems (TOCS)* 7, 4 (1989), 321–359.

[7] Dipankar Sarma Paul E. McKenney. 2002. Read Copy Update. *Proceedings of the Ottawa Linux Symposium* 31, 4 (2002), 338–367.