Performance of LibRe protocol inside Cassandra workflow

January, 2015
Paris

From

Sathiya Prabhu Kumar
Ph.D student, CNAM

Under the supervision of

Raja Chiky, Sylvain Lefebvre
LISITE - ISEP
28 Rue Notre Dame des Champs
Paris, France.

Eric Gressier Soudan
CEDRIC - CNAM
292 Rue Saint-Martin
Paris, France.
Contents

1 Introduction 3

2 LibRe 3
   2.1 Algorithm Description ........................................ 4
   2.1.1 Update Operation ........................................... 5
   2.1.2 Read Operation ............................................. 6

3 CaLibRe: Cassandra with LibRe 6
   3.1 LibRe implementation inside Cassandra ....................... 6

4 YCSB: Yahoo Cloud Services Benchmark 7
   4.1 Evaluating Data Consistency using YCSB ....................... 7
   4.2 CaLibRe performance evaluation using YCSB ..................... 8
       4.2.1 Test Setup ............................................... 8
       4.2.2 Test Evaluation ......................................... 8

5 Conclusion 9

6 Future Works 9
1 Introduction

The stronger consistency options offered by most of the modern storage systems aka NoSql databases [10] can take one of the two possible variants. First one is reading or writing to all the replicas: Consistency-level ALL. Second is reading and writing to a majority of replicas: Consistency-level QUORUM. Some of the limitations of these models include extra cost on request latency, high number of messages and limits the system availability if one or few of the data replicas are unable to contact. In large distributed storage systems, one or few of the nodes may be down at some point in time. Hence, in case of reading or writing to all the replicas, there is always a threat on system availability for a set of data. Most of the modern storage systems are designed to be write optimized, which are meant to be dominant for writes, letting the system to always succeed the writes and as fast as possible. For example, in Cassandra, consistency option ANY and HintedHandoff confirm this behavior [7]. So in case of system that needs higher write possibility and minimum read guarantee, the desired consistency level is ONE [7]. If a data is written with consistency level ONE, the only mode to preserve consistency of this data during read time is reading from all the replicas. But as mentioned earlier, this could be a threat for the system availability. The consistency option QUORUM [7] can tolerate if few replicas are down or couldn’t be contacted, but the pass condition for the writes goes stronger with a considerable cost on both read and write operations.

The key performance of the modern storage systems rely on handling most of the data in-memory. The system is designed in a way to cache most of the recent data in order to handle the read requests faster. In case of forwarding a request to all or majority of replicas to retrieve a data, the possibility that all replicas get the right data in their cache would be improbable. Hence, the request latency will always be elevated by the slowest replica responding the request. Moreover, since stronger consistency options always come with a cost on request latency, applying stronger consistency during a peak point where the number of users trying to access the same data are high could exacerbate the system.

Another probable problem exposed by these consistency options is on the system elasticity. In the recent growth of cloud computing, the need for scalability evolved to elasticity. Scalability is the ability of the platform or architecture to handle the increasing system demand by adding additional resources. Elasticity is the system tactics to handle the varying system demand efficiently with the available system resources based on the current situation in an automatic way. Scalability is a long-term resolution, whereas Elasticity is a short-term resolution. In the recent trend of data computing, to achieve elasticity, the replica count of a data needs to be altered dynamically as in case of CDNs (Content Delivery Networks) [17]. Holding varied number of replicas instead of static replica number for a data enables elasticity, allowing the system to provision and de-provision the resources automatically. In case of varying number of replicas, the stronger consistency options: Consistency-level ALL or QUORUM might be inefficient as the replicas set for the given data is continuously changing.

2 LibRe

Analyzing the pros and cons of existing consistency options, we propose a new consistency protocol named LibRe: Library for Replications. The main goal of the policy is to achieve stronger consistency while reading from ONE replica instead of ALL or a QUORUM of replicas irrespective of the number of replicas disconnected during reads and/or writes. From the insight of the Eventual Consistency systems, at least one of the replicas contain the recent version of the needed data at any point in time. Thus, forwarding the read requests to the right node: the node that contains the recent version of the data or in other words, by restrict forwarding the read requests to the Stale Node: the node that is not updated, the consistency of the data would be preserved.

Hence, the design considerations of LibRe are:

- Ensure Read requests are forwarded to the node holding the recent version of the needed data.
- Ensure system Availability and Latency constraints are not violated.

One of the main challenges of the model is ‘How to claim a node is stale ?’. A stale node is the one where the recent update for the needed data is not yet applied. We can’t assume a node is stale if it contains one or few stale data among huge datasets. The node is stale only for a particular data items, but still consistent for other data items. Hence a registry is needed to monitor each write and update operations and to provide information in accordance to each data item. So that, the same node can be classified to be stale for particular data items and consistent for remaining data items. For example, let’s
assume a topology where servers are connected to a common high-speed network. If a particular node in the topology is down or separated from the network for a period of few minutes, the node will be inconsistent only for the operations that happened during this period. However, the node will be consistent for the remaining data items. Hence, the core idea of our approach is to identify the messages lost by a node. In other words, the model tries to identify the updates missed by a particular node. It enables the system to stop forwarding the requests to the stale node until it is consistent again. A node is considered to be stale if it contains stale data for the incoming request. This restriction can be freed once the lost updates are reapplied on the node. Hence, in order to achieve so, during update of specific data items on a node, the node has to make an announcement to LibRe Registry. The announcement should be alike: the particular data with corresponding resource identifier is added/modified in this node. So that, with the Registry of nodes announcements, the Read requests will be forwarded to the node that contains the recent version of the needed data.

The next obvious question would be ’Where to keep the Registry?’ An entity that strikes the mind immediately would be the ’NameNode’ of the Hadoop Distributed File System (HDFS) [14, 16]. The NameNode of the HDFS that manages add/copy/move/delete of a file in the file system and helps to locate the DataNodes where the needed data lives does more or less what we wanted to do with LibRe. The NameNode of HDFS manages the metadata about the larger file blocks, but in case of LibRe, the Registry has to be maintained for each individual data item, which might be bit trickier than what NameNode actually manages. On the other hand, NameNode of HDFS is a single point of failure and could be the bottleneck for the real-time OLTP requests. The inspired system for the course of LibRe registry is the Zookeeper [8]. The initial implementation of LibRe protocol tried using Zookeeper for managing the registry information. However, for better performance, in the recent version of LibRe protocol, Zookeeper is replaced by Distributed Hash Table (DHT) [18]. With the DHT design, each node manages the registry information about a set of data.

2.1 Algorithm Description

Figure 1 shows the position of LibRe in the system architecture. The Frontend is a node in the system where the client connects to. In a Multi-Reader, Muti-Writer architecture, each node in the system can take the role of Frontend. When the frontend receives a read request, the target node to query the request will be retrieved via the LibRe protocol.

As showed in the figure 1, the LibRe protocol consists of three components namely Registry, Availability Manager and Advertisement Manager. The Registry is a key-value store in which for a given data-key, the set of replicas containing the recent version of the data is stored. Availability Manager is contacted during read operations, which is responsible for forwarding the read request to an appropriate node that holds the recent version of the data. During Write operations, the replica notifies the Advertisement Manager about the current update. Availability Manager in turn responsible for updating the Registry recognising the recentness of the data. The whole setup is a Distributed Hash Table (DHT) [18]. While retrieving the set of replicas that is responsible for storing the given data-key, one among those replicas holds the three components for that particular data-key.

![Figure 1: LibRe Architecture Diagram](image)

Figure 2a and 2b shows the sequence diagram of a distributed data storage system with the LibRe contribution during the Write and Read requests respectively. From the diagram, when a client issues a Write/Update request, the frontend forwards it to the appropriate replicas based on the chosen consistency.
level. When the Write operation is successfully accomplished by the replica, the replica checks whether the update has to be notified to the LibRe node. If yes, the update will be notified to the Advertisement Manager of the corresponding LibRe node. If not, no action will be taken. During read operations, the frontend checks whether the information about the corresponding data-key is managed by the LibRe. If yes, the frontend forwards the request to the LibRe node. The LibRe node in turn finds an appropriate replica node to query the read request via the Availability Manager and forwards the request to that node. Finally, after querying the needed data, the replica node forwards the read response (result) directly to the client.

2.1.1 Update Operation

Algorithm 1 describes the contribution of LibRe’s Advertisement Manager during an update operation. In Update Operation, there are actually two cases:

- Insert: When a data is written to the storage system for the first time.
- Update: When an existing data is modified. The update could be either issued by a client or gossiped from another replica.

### Algorithm 1 Update Operation

```python
1: function log(dataKey, versionId, nodeIP)
2:     if Reg.exists(dataKey) then
3:         RegVersionId ← Reg.getVersionId(dataKey)
4:         if versionId = RegVersionId then
5:             replicas ← Reg.getReplicas(dataKey)
6:             replicas ← appendEP(replicas, nodeIP)
7:             Reg.updateReplicas(dataKey, replicas)
8:         else if versionId > RegVersionId then
9:             replicas ← reinitialize(nodeIP)
10:            Reg.updateReplicas(dataKey, replicas)
11:        Reg.updateVersionId(versionId)
12:    end if
13:  else
14:      replicas ← nodeIP
15:        Reg.createEntry(dataKey, replicas)
16:      Reg.updateVersionId(versionId)
17:  end if
18: end function
```

When a replica node sends an advertisement message regarding an update, the Availability Manager of the LibRe node follows the following actions. First the protocol checks whether the data-key already exists in the Registry: line 2. If the data-key exists in the Registry (update operation), line 3: the version-id logged in the Registry for the respective data-key will be taken. Version-id is nothing but a monotonically increasing number representing the recentness of the update, for instance say: timestamp of the operation. Line 4: If the version-id logged in the Registry matches with the version-id of the operation (gossiped update), then line 7: the node’s IP-address will be appended along with the existing replica list. Line 8: If both the version-id’s don’t match and if the version-id of the operation is greater than the version-id exists in the Registry (which means, the update is new), the line 9-10: replica list for the data-key will be reinitialized to the node’s IP-address. Also line 11: the version-id will be updated to
the version-id of the operation. In other case, line 13: if the data-key doesn’t exist in the Registry (write operation), a new entry will be created for the data-key with the node’s IP-address and the version-id of the operation (line 14 to 16). This setup helps to achieve the Last Writer Wins policy [13].

2.1.2 Read Operation

Algorithm 2 describes the LibRe policy during the read operation. According to the algorithm, since the Registry holds information about the replicas holding the recent version of the needed data, the replicas information will be retrieved from the registry (line 2). Then, line 3: a target node will be chosen from the replicas to forward the request: line 6. However, if the Registry doesn’t contain the information about the needed data-key, for the favor of system Availability over Consistency, the protocol follows the default system behavior to retrieve a target node.

Algorithm 2 LibRe Read

1: function getTargetNode(dataKey)
2: replicas ← Reg.getReplicas(dataKey)
3: targetNode ← getTargetNode(replicas)
4: if targetNode is NULL then
5: targetNode ← useDefaultMethod(dataKey)
6: end if
7: forwardRequestTo(targetNode)
8: end function

3 CaLibRe: Cassandra with LibRe

With the encouraging results obtained by simulation, we decided to implement LibRe inside a Distributed Storage System: Cassandra. Cassandra[9] is one of the most popular NoSql systems available open-Source and continuously refined and enhanced by a big open-source community. Cassandra offers different consistency options for reading and writing, some of the most used options are Consistency-level ONE, QUORUM and ALL [7]. The reason for choosing Cassandra for studying LibRe performance is its open-source nature and options to switch between different consistency-levels. The motivation is to incorporate the LibRe protocol for accomplishing reads and writes and compare its performance with Cassandra’s native consistency options: ONE, QUORUM, ALL.

3.1 LibRe implementation inside Cassandra

LibRe protocol is implemented inside Cassandra release version 2.0.0. In the native workflow, while querying a data, the endpoints (replicas) addresses are retrieved locally via matching the token number of the data with the token number of the nodes [18, 15, 6]. Each operation, Read/Write is sent to all the endpoints and waits for at least one successful response to succeed the operation if the chosen Consistency-level is ONE. In case of Consistency-level ALL, the successful response is expected from all the endpoints. Whereas Consistency-level QUORUM, expects success response from the majority of the endpoints.

While retrieving the set of endpoints addresses via matching the token number of the data against the token number of the nodes, the first endpoint address before sorting it based on the proximity, becomes the LibRe node for that data. A separate thread pool for the LibRe messaging service is designed for handling the LibRe messages effectively. In the LibRe implementation, the write operations follow the native workflow of Cassandra, except the node sends a ‘LibRe-Announcement’ message to update the LibRe registry at the end of the successful write (if the data has to be managed by LibRe). In order to verify whether the data has to be managed by LibRe or not a Bloom Filter is used [4]. The ‘LibRe-Announcement’ message contains the metadata about the update such as the data-key associated to the update, IP-address of the node, and the version-id. In the initial implementation of LibRe inside Cassandra, the hash of the modifying value is used as the version-id, although the hash value is not monotonically increasing and couldn’t be compared directly to an existing version-id. When the LibRe node receives a ‘LibRe-Announcement’ message, the LibRe Announcement manager will take control of it and handle the message. The messaging service will un-marshal the needed data and transfer it to the LibRe announcement manager to update the LibRe registry. The registry is updated matching the version-id of the operation with the version-id of the data already registered in the registry as described in the section 2.1.1, except the comparison of the two version id’s. In this case, assuming there is no delay
in the update messages, the version-id that doesn’t match with the version-id of the registry is always considered to be the new version. However, in the future implementation, the hash of the modifying value will be replaced by the timestamp of the operation, which would increase monotonically and facilitate the comparison.

During read operation, if the data-key (Row-key in Cassandra vocabulary) exists in the Bloom Filter, instead of following the default path in querying the request, the node will send a 'LibRe-Availability' message to the LibRe node. The LibRe-Availability message is prepared as the normal 'Read Message' for the needed data with the data-key on top of the message. When the LibRe node receives a 'LibRe-Availability' message, the LibRe availability manager finds an appropriate replica node to query the data. The data-key added on top of the message helps to find the target node without un-marshaling the whole message. After finding a target node to query the data, the read message will be sent to the appropriate node extracting the 'Read Message' from the 'LibRe-Availability' message. Handling the 'LibRe-Availability' message in this way profits time in un-marshalling and marshalling the whole message; also speeds the read response process.

4 YCSB: Yahoo Cloud Services Benchmark

YCSB: Yahoo Cloud Services Benchmark is one of the open source project written in JAVA for benchmarking the cloud databases. The main goal of YCSB is to measure the performance of the system by means of Throughput, Request Latency, Saturation Point and the Scalability of the database under common workload patterns. Unlike simulation tools, YCSB facilitates to examine the physical features and behaviors of the system with live workload passing it to the real-time system. The reason for YCSB implementation over the existence of traditional (e.g., TPC-C) benchmarking is the generation gap in the data storage systems. The current generation data storage systems introduced the facts like BASE [11] compliance in contrast to ACID [12], incorporations of tradeoffs and adaptations to diverged workload patterns. The TPC-C kind of benchmarking tools generates workloads which are more suitable for traditional RDBMS storage systems that includes transaction. Whereas, the modern storage systems are not meant for these kind of workload patterns. YCSB generates different types of workloads in compliance with the patterns suitable for the modern storage systems. The different workload patterns generated by YCSB includes Update Heavy, Read Mostly, Read Only, Read Latest, Scan-Insert and Read-Modify-Write.

4.1 Evaluating Data Consistency using YCSB

In distributed data store, a data item is logically one and the same irrespective of the physical copies. A read for a data item arrived after an update should see the last updated value. The read that don’t see the last updated value is called stale read. Till date, there is no proper tool or library in the literature for evaluating data consistency (number of stale reads) both in real time and via simulations. YCSB provides invoice about the performance of the systems for the current workload via Minimum Latency, Maximum Latency, Average Latency, 95th Percentile, 99th Percentile, but don’t count on the number of stale reads.

In YCSB, the workloads will be generated with the number of client threads defined by the user. As each data store system is developed for different use case models, the querying interface for each data store technically differ from one another. Hence in order to query different data stores, YCSB offers additional bindings called Database Interface Layer/Db binding that are written exclusively for the appropriate data stores to work along with YCSB. The role of the Database Interface Layer is to translate the workloads generated by the workload generator into system appropriate format to be queried. One issue with YCSB in extending it to measure a new metric along with the offered metrics is that, after each query, the database interface layer returns the needed logistics to the status evaluation module instantly. After all the operations are completed, the status evaluation module computes the needed metrics and prints the invoice of the test case to an output file. So a pact between the two layers should be established safely without affecting the existing computations in order to compute a new metric. In our case, in the database interface layer of Cassandra, we introduced a Hash-Map data structure to account each I/O operation. So during write/update operation, the concatenation of the row-key and the column-name forms the key of the Hash-Map and the column timestamp as the value. During read operation, the timestamp of the retrieved data will be compared along with the timestamp registered for the particular data in the Hash-Map. If both the values are compliant, then the read will be considered as consistent, else the stale read count will be incremented by one. After each operation, along with other logistics of
the operation, the stale read count is returned to the status executor module. By this way, the status executor module prints the number of stale reads encountered during the test suit in the output file along with the default statistics.

4.2 CaLibRe performance evaluation using YCSB

4.2.1 Test Setup

The experiment was conducted on a cluster of 19 Cassandra/CaLibRe instances, which includes 4 medium, 4 small and 11 micro instances of Amazon EC2 [1] and 1 large instance for the YCSB test suite. All instances were running on the platform Ubuntu Server 14.04 LTS - 64 bit. The workload pattern used for the test suite is workload-a: Update-Heavy with Record Count of 100000, Operation Count of 100000, Thread Count of 10 and Replication-Factor as 3 for the test case. The test case compares the performance of 19 Cassandra instances with different consistency option: ONE, ALL and QUORUM against 19 CaLibRe instances with consistency-level: ONE. Since encountering number of stale reads in a small test setup is minimum, partial update propagation is injected into the Cassandra and CaLibRe cluster to account for the system performance under this scenario. Hence, during update operation, instead of propagating the update to all the replicas: 3, the update will not be propagated to one of the replicas. In our case, since the Replication-Factor is chosen as 3, instead of propagating the update to 3 replicas, the update will be propagated to only 2 replicas.

4.2.2 Test Evaluation

The figures 3a, 3b and 3c respectively shows the evaluation of Read Latency, Write Latency and the number of Stale Reads of Cassandra with different consistency options against Cassandra with the LibRe protocol. In the figure 3a, the entity ONE represents the read and write operations with consistency option ONE. The read and write operations with consistency option QUORUM is implicated by the entity QUORUM. The entity ONE-ALL represents the operations with write consistency option ONE
and read consistency option ALL. The entity CALIBRE represents the performance of the proposed LibRe protocol built inside Cassandra.

From the read latency graph: figure 3a, we can see clearly that the 95th Percentile Latency of CALIBRE remains the same as that of the other consistency options of Cassandra. The 99th Percentile Latency of CALIBRE and cassandra with consistency level ONE remains same and better than the other options: ONE-ALL and QUORUM. The minimum and average latencies of CALIBRE is slightly elevated when comparing to Cassandra with consistency option ONE but better than the consistency option QUORUM and ALL. On the other hand, while looking at the write latency graph: figure 3b, the consistency options both ONE and ONE-ALL does the write only once. So, the main comparator for the CALIBRE during Write’s is the consistency option QUORUM. From the graph, it is clear that CALIBRE is better than the consistency option QUORUM in all the points: 99th Percentile, Minimum and Average latencies. However, when comparing to the entities ONE and ONE-ALL, there were very minimum elevation seen in CALIBRE but seems not much significant. On looking at the last graph 3c: Stale Reads, the number of stale reads recorded with the consistency option ONE is higher. There were few stale reads found in other consistency options too but are negligible when comparing to the number of requests: 100000. Hence it can be taken as, except consistency option ONE the consistency options ONE-ALL, QUORUM and CaLibRe confirms tight consistency. On conclusion, the performance of CALIBRE proved tight consistency with better read and write latencies than the popular combinations of the Cassandra’s native consistency options.

5 Conclusion

The work described in this document aims at studying tradeoff between Consistency, Latency and Availability of modern storage systems. The idea was to track the updates of some of the data items in order to offer better tradeoffs in the system. The performance of the system with the proposed idea: LibRe was evaluated in a real world distributed data storage system: Cassandra. Since encountering number of stale reads in a small test setup is minimum, partial update propagation is injected into the test setup to account for the system performance under this scenario. The performance results lead to several conclusions regarding read & write latencies. The results proves that LibRe protocol offers a better tradeoff between Consistency, Latency and Availability of distributed data storage system. In other words, if we can track the updates of few vulnerable data items, the distributed data storage system would maintain better tradeoff between Consistency, Latency and Availability. Some of the additional facts unveiled in our research includes: restrict forwarding read requests to stale nodes hold strong consistency, possibility of achieving strong consistency irrespective of number of replicas up/down, possibility of achieving strong consistency with minimal read and write guarantees.

Measuring the consistency of a distributed data storage system remains challenging both by simulations and benchmarking the system with live workloads. Hence, in order to benchmark the system with live workloads, we have extended one of the popular benchmarking tool YCSB to account on the consistency aspect of the system along with the existing system metrics offered by YCSB.

6 Future Works

Most of the popular distributed data storage systems offer different levels of consistency options and let the user/application to choose the desired consistency-level based on the severity of the request. But as the user and the system behaviors change dynamically over time, deciding a consistency-level in advance becomes tricky for the application developers. Adding intelligence to the system to predict the needed consistency-level based on the user’s and/or system behavior is a broad area of research. With the help of some statistical information, it is possible to detect the user’s behavior in the future. For example in case of time series data, the data that is accessed most over time follow similar behavior in future with respect to some influential factors. Some use cases of this type demand strong consistency during peak point and reduced consistency during non-peak point. In this type of use cases, in order to avoid exacerbating the system by applying strong consistency during the peak point where number of users try to access a particular data are high, LibRe protocol can be used to ensure the consistency of the data with lesser load to the system. A sample use case that we are currently experimenting is the Velib use case [3, 2], an application that gives information about the number of bikes and/or bike-slots available before taking/parking the bike at a velib station. The article [5] gives background about the use case and sorted the velib stations into three clusters: Balanced, Overloaded and Underloaded. The idea is
to use the LibRe protocol if the request is during the peak time and the needed information about the station belongs to the Overloaded or Underloaded cluster.

References