CaLibRe: A better Consistency-Latency Tradeoff for Quorum based Replication systems

Sathiya Prabhu Kumar[1,2]
Sylvain Lefebvre[1]
Raja Chiky[1]
Eric Gressier-Soudan[2]

[1]LISITE Laboratory, ISEP Paris, Paris, France
[2]CEDRIC Laboratory, CNAM Paris, France

The final publication is available at Springer via
http://www.dx.doi.org/10.1007/978-3-319-22852-5_40

CaLibRe: A better Consistency-Latency Tradeoff for Quorum based Replication systems

Sathiya Prabhu Kumar[1,2], Sylvain Lefebvre[1], Raja Chiky[1], Eric Gressier-Soudan[2]

[1] LISITE Laboratory, ISEP Paris, France
[2] CEDRIC Laboratory, CNAM Paris, France

Abstract. In Multi-writer, Multi-reader systems, data consistency is ensured by the number of replica nodes contacted during read and write operations. Contacting a sufficient number of nodes in order to ensure data consistency comes with a communication cost and a risk to data availability. In this paper, we describe an enhancement of a consistency protocol called LibRe, which ensures consistency by contacting a minimum number of replica nodes. Porting the idea of achieving consistent reads with the help of a registry information from the original protocol, the enhancement integrate and distribute the registry inside the storage system in order to achieve better performance.
We propose an initial implementation of the model inside the Cassandra distributed data store and the performance of LibRe incarnation is benchmarked against Cassandra's native consistency options ONE, ALL and QUORUM. The test results prove that using LibRe protocol, an application would experience a similar number of stale reads compared to strong consistency options offered by Cassandra, while achieving lower latency and similar availability.

1 Introduction

In distributed data storage systems, data is replicated to improve the performance and availability of the system. However, ensuring data consistency with higher availability and minimum request latency is notoriously challenging [1,9]. In order to efficiently handle these challenges, the Dynamo system [7] designed by Amazon uses a quorum-based voting technique that facilitates configurable tradeoffs between Consistency, Latency and Availability. This technique inspired subsequent distributed data storage systems such as Cassandra [14], Voldemort [22] and Riak [11]. The quorum-based voting technique ensures consistency based on the math behind the intersection property of the quorum systems [18,23]. This intersection property can be expressed by the formula $R + W > N$. This formula symbolizes that the system can ensure consistency if the sum of the nodes acknowledging the Read (R) and the nodes acknowledging the Write (W) is greater than the total number of replicas (N). Since more nodes have to be contacted, ensuring consistency comes with a communication cost and a threat to the system availability. Hence, in order to provide fast response time, these

storage systems rely on eventual consistency and do not satisfy the intersection property by default. Therefore the user can configure the number of quorum members to contact during read and write time in order to ensure consistency on demand.

If the intersection property cannot be satisfied, the system will reject the operation. The three popular ways of satisfying the intersection property to ensure strong consistency are as follows: write to all and read from one node, write to one and read from all nodes, write and read to / from a majority of nodes.

Most of these storage systems are optimized for write intensive workloads, which requires the system to acknowledge writes as fast as possible and reconcile conflicts at read time. For these systems to guarantee the reads with minimum latency, the default eventual consistency option (non-overlapping quorum) is desirable. But, if a data item is written with the minimum write quorum (one node), the only mode to preserve consistency for this data during read time is reading from all the replicas. However, if one of the nodes is temporarily down or can not be contacted, the read will fail. The same risk applies when the system writes to all nodes and reads from one node. When reading and writing from / to a majority of nodes, failure of one or few replica nodes is tolerable. In that case, availability guarantees will still be affected if the system is not able to communicate to a majority of replica nodes. Besides, the latency for both reads and writes will be affected as well.

Performance of these modern storage systems rely on caching most of the recent data in order to handle the read requests faster. When a request is forwarded to all or majority of replicas to retrieve a data item, the possibility that all replicas have the right data in their cache would be improbable. Hence, the slowest replica node responding to the request will increase the request latency and the advantage of cache memory is lost.

The existing "strong" consistency options offered by these storage systems are strong enough to ensure data consistency when no partition occurs, but add some extra communication cost and a risk to data availability. To our knowledge, there is no softer option that can ensure consistency guarantees with availability and latency guarantees similar to the default level of eventual consistency.

In this paper, we discuss improvements on a consistency protocol called LibRe [12], which acts as an in-between consistency strategy between the default eventual consistency and the strong consistency options derived from the intersection property.

The original LibRe protocol used a registry, which records the list of replica nodes containing the most recent version of the data items. Hence, referring to the registry during read time helps to forward the read requests to a replica node holding the most recent version of the needed data item.

Instead of relying on a synchronization service such as Zookeeper [10], as initially proposed in [12], the enhanced protocol distributes the registry over each node in the cluster and manages the data items entries in the registry only until all the replicas converge to a consistent state. These mechanisms are detailed

in section 2.4, along with the protocol description. Since the improvements are inclined towards higher availability and minimum request latency, its consistency guarantee is slightly relaxed compared to the original LibRe design [12]. For the sake of simplicity, in the following sections, the enhanced LibRe protocol that the paper intended to describe is termed as LibRe and its initial version proposed in [12] is termed as original LibRe or simply original protocol.

The following section provides a description of the LibRe protocol. The implementation inside the Cassandra distributed data storage system [14] and its performance evaluations are discussed in section 3. We describe some of the related works at section 4 before presenting our conclusions and future works in the last section.

2 LibRe

The LibRe acronym stands for "Library for Replication". The protocol collects information about writes until they are fully propagated to all the replica nodes. If an update is not propagated to all the replica nodes, then an entry for this data item is added to an in-memory data structure called the LibRe Registry, along with a *version-id* and the list of replica nodes holding this latest version. The *version-id* is a monotonically increasing value representing the recent version of the data item, for instance it can be the timestamp of the operation or a version vector. Consulting the registry at read time helps to avoid forwarding the read requests to a replica node where the recent update is not effected.

2.1 Targeted System

Key-Value data stores that store an opaque value for a given Key seen enough popularity in the modern distributed storage systems. Some of these systems ensure strong consistency, whereas most of the systems such as Dynamo [7], Riak [11] and Voldemort [22] rely on tunable consistency. In order to tune the consistency level of the system on a per-query or per-table basis, these storage systems follow a quorum-based voting technique such as described in section 1. These systems mostly use a Distributed Hash Table (DHT) [24] for identifying the replica nodes of a data item. LibRe protocol targets the Key-Value data stores that offer tunable consistency based on quorum based voting technique using DHT. In building LibRe we assume that the underlying system provides failure detection and tolerance mechanisms, which are building blocks for the reliability of our Registry. Currently, we do not consider the use of LibRe for inter-cluster replication.

2.2 LibRe Registry

The core of the LibRe protocol is its Registry. The registry is an in-memory key-value data structure that takes the data identifier as the Key and the list of replica nodes id (IP addresses) holding the most recent *version-id* of the data

item as the Value. During write operation, each replica node tries to add its id in the list. If the number of ids in the list reaches the total number of replica nodes for the data item, then confirming the convergence of all replicas, the entry for the data item in the registry can be safely removed.

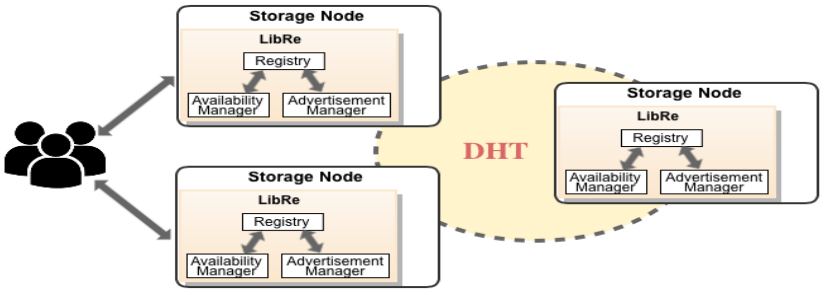


Fig. 1: LibRe Architecture Diagram

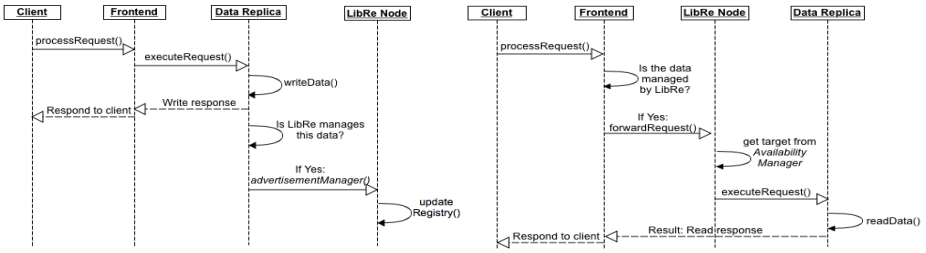
The registry is distributed over all the nodes in the cluster, but at any point in time only one copy of the entry for a data item exists. An entry for a data item d_i will be stored on only one of the available replica nodes that is responsible for storing the data item d_i.

Figure 1 shows the position of LibRe in the system architecture and the components of the LibRe protocol. Let R_i be a replica set for a data item d_i, such that $R_i = \{r_1, r_2, ..r_n\}$, where r_x is a node identifier, and n is the number of nodes in the replica set. So, one of the available replica nodes in R_i (say the first one: r_1) will hold the registry and the other two supporting components of LibRe: the Availability-Manager and the Advertisement-Manager, as shown in figure 1. The node that holds the registry and the supporting components is called the Registry Node for the particular data item. For any data item d_i, the id of the first replica node obtained via consistent hashing function is the registry node id. In other words, the replica node that has the lowest token id is considered as the registry node. The registry is distributed over each node in the cluster, so each node plays the role of replica node as well as registry node.

2.3 LibRe Messages

The LibRe protocol is based on two types of messages, namely: the Advertisement Message (figure 2a) and Availability Message (figure 2b), corresponding respectively to the Advertisement-Manager and Availability-Manager of the LibRe components shown in figure 1. The figure 2 and the corresponding algorithms 1 and 2 discussed at the section 2.4 show the improvements of the LibRe protocol over the original algorithm discussed in the paper [12].

Advertisement Message: From figure 1, a client can connect to any node in the system. Some storage systems call this node the Coordinator node [14]. In the



(a) LibRe Write/Update Operation (b) LibRe Read Operation

Fig. 2: LibRe Sequence Diagram

usual system behavior, a write request will be forwarded to all the replica nodes that are available. If the coordinator node receives back the required number of acknowledgements (or votes) from the replica nodes for the write, the coordinator issues a success response to the client. If the sufficient number of vote is not received within a timeout period, the coordinator issues a failure response to the client. LibRe protocol follows this default system behavior, but in addition, after a successful write operation, the replica node sends an *advertisement message* to the registry node asynchronously. The replica node sends advertisement message only for the data item that are configured to use LibRe. The advertisement message consists of the data key, version-id and originating node id.

Availability Message: When the coordinator node receives a read request that is configured to use LibRe protocol, the coordinator sends an *availability message* to the Registry Node of this particular data item. The availability message contains the original *read message* received from the client and the data key of the needed data item. When the registry node receives an *availability message*, it finds a replica node from the registry and forwards the original *read message* to that replica node. The replica node sends the read response directly to the coordinator node and the coordinator forwards it to the client. If an entry for a data key is not found in the registry, then the *read message* will be forwarded to one of the available replica nodes.

2.4 LibRe Protocol

Algorithm 1 describes the role of LibRe's *Advertisement Manager* during update operation. The update for a data item can be issued when its replicas are in converged state or in diverged state.

When a replica node sends an advertisement message regarding an update, the *Availability Manager* of the LibRe node takes on the following actions. First the protocol checks whether the *data-key* already exists in the *Registry*: line 3. If the *data-key* exists in the registry (replicas are in diverged state), line 3: the *version-id* logged in the registry for the respective data-key is compared to

Algorithm 1 Update Operation

$r_k = \{n_i, n_j, ...\}$: set of replica nodes holding recent version of data item k.
$e_k =< v_k, r_k >$: record where v is a version-id and r is a replica set.
$R : k \rightarrow e_k$: Map of data item keys k to corresponding entry record.
N: Number of replicas

```
1: function ADVERTISEMENTMANAGER(k, v, n)
2:     entry ← R.k
3:     if entry ≠ ∅ and entry.v = v then
4:         entry.r ← entry.r ∪ {n}
5:         if |entry.r| = N then
6:             R ← R \ {k}
7:         end if
8:     else if entry = ∅ or v > entry.v then
9:         entry.v ← v
10:        entry.r ← {n}
11:    end if
12:    R.k ← entry
13: end function
```

the one sent with the update message. If the *version-id* logged in the registry matches with the *version-id* of the operation (replica convergence), then line 4: the node id (IP-address) will be appended to the existing replica list. Line 5-6: If the number of replicas in the list is the same as the total number of replicas for data item k, then replicas are in converged state, and the entry is deleted. If the entry does not exist in the registry or the *version-id* of the operation is greater than the existing *version-id* in the registry (line 8): the entry is created or reinitialized with the operation's *version-id* and the sender node id (lines 9 and 10). Finally the entry is recorded in the registry (line 12). This setup helps to achieve Last Writer Wins policy [17,19].

Read Operation Algorithm 2 describes the LibRe policy during the read operation. According to the algorithm, since the Registry keeps information about the replica nodes holding the recent version of data item k, the nodes information will be retrieved from the registry (line 2). Line 3: If an entry for the data-key exists in the registry, line 4: one of the replica nodes from the entry will be chosen as the target node. The method *first()* (line 4 and 6) returns the closest replica node sorted via proximity. Line 5: if the Registry does not contain an entry for the needed data-key, then, line 6: one of the replica nodes that are responsible for storing the data item will be retrieved locally via DHT lookup. Finally, the read message will be forwarded to the chosen target node: line 8.

2.5 LibRe Reliability

As mentioned in section 2.1, the reliability and fault tolerance of the LibRe protocol relies on the guarantees of the targeted system. The systems that use DHT for quorum based voting actively take care of the ring membership and failure detection [7]. The underlying DHT helps to find the first available replica node. In the event of node joining and/or leaving the cluster, the Consistent Hashing technique supports minimal redistribution of the nodes keys. In such case, there

Algorithm 2 Read Operation

$r_k = \{n_i, n_j, ...\}$: set of replica nodes holding recent version of data item k.
$e_k =< v_k, r_k >$: record where v is a version-id and r is a replica set.
$R : k \rightarrow e_k$: Map of data item keys k to corresponding entry record.
$D_k =< n_i, n_j, ... >$: replica nodes for data item k that are obtained via default method.

```
1: function getTargetNode(k, M_read)
2:     replicaNodes ← R.k.r
3:     if replicaNodes ≠ ∅ then
4:         n ← replicaNodes.first()
5:     else
6:         n ← D_k.first()
7:     end if
8:     forward(M_read, n)
9: end function
```

will be a change in the first available replica node (registry node) for a few data items and the registry information for these data-keys would not be available. In that case, the registry information will be rebuilt on the new registry node by sending successive *advertisement messages* to this node. This may lead to small and time limited inconsistencies in the system. Therefore, LibRe sacrifices Consistency in favor of Availability: cf. algorithm 2. If a registry node that has been unavailable joins back the cluster, the stale registry information has to be flushed during a handshaking phase. Besides, periodic local garbage collection is needed to keep the registry information clean between replica nodes.

2.6 LibRe Cost

The tradeoff provided by the LibRe protocol comes at the expense of some additionnal cost on message transfers and memory consumption.

Extra Message Transfers: In LibRe, a lookup in the registry is required during a read operation on contacting the *availability manager* of the registry node to read from a right replica node. However, this operation represent constant cost, as the number of messages sent for achieving the consistent read does not depend on the number of replicas involved, as mentioned in section 2.3. Besides, the latency spent during this lookup can be gained back via managing the cache memory efficiently. During write operations, notifying the *advertisement manager* about an update is asynchronous and does not affect the write latency. Although these messages are an additional effort when compared to the default eventual consistency option, it is better than the strong consistency options that communicate to a majority or all replica nodes during reads and/or writes.

Registry in-memory data structure: LibRe manages the registry information in-memory. This information is distributed among all the nodes in the cluster and is maintained only for the data items whose recent update is not effected on all replica nodes. Moreover, eventual consistency guarantees of the targeted system and the periodic local garbage collection of the LibRe protocol helps to reduce the amount of information to be kept in-memory.

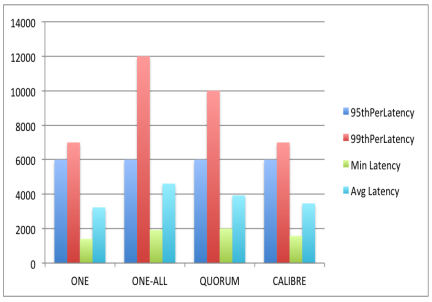
3 CaLibRe: Cassandra with LibRe

Cassandra [14] is one of the most popular open-source NoSql systems that satisfies the system model specified in section 2.1. Hence, we decided to implement the LibRe protocol inside the Cassandra workflow and evaluate its performance against Cassandra's native consistency options: ONE, QUORUM and ALL. Although Cassandra is a column family store, we used it as a Key-Value store during test setup: refer section 3.1. LibRe protocol was implemented inside Cassandra release version 2.0.0. In the native workflow, while querying a data, the endpoints (replica nodes) addresses are retrieved locally via matching the token number of the data item over the nodes token numbers. The IP-address of the first alive endpoint (without sorting the endpoints by proximity), will be chosen as the registry node for the replica sets it is responsible. A separate thread pool for the LibRe messaging service is designed for handling the LibRe messages effectively. On system initialisation, all LibRe registries are empty until a write request is executed, which will trigger the protocol and start filling up the registries. CaLibRe can be configured to work either by passing a list of data items in a configuration file, or by specifying directly the name of the table(s) to monitor. Currently, the *version-id* used in CaLibRe is the hash of the modifying value. However, it will be replaced by a timestamp in the future.

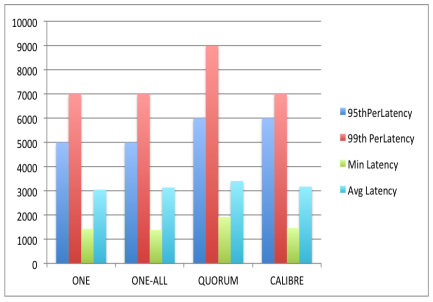
3.1 CaLibRe performance evaluation using YCSB

Test Setup The experiment was conducted on a cluster of 19 Cassandra and CaLibRe instances that includes 4 medium, 4 small and 11 micro instances of Amazon EC2[3] cloud service and 1 large instance for the YCSB test suite [6]. All instances were running Ubuntu Server 14.04 LTS - 64 bit. The workload pattern used for the test suite was the "Update-Heavy" workload (*workload-a*), with a record count of 100000, operation count of 100000, thread count of 10 and the Replication-Factor as 3. YCSB by default stores 10 columns per RowKey. We used RowKey as the data key, for which, an entry will be managed in the LibRe Registry. Using RowKey as the data key could leads to a situation like, if one or few columns of a RowKey is updated on a replica node r_n, the registry would assume r_n contains the recent version for all the columns of the RowKey. In order to avoid this situation, we update all the 10 columns during each update. The test case evaluate the performance and consistency of the 19 Cassandra instances with different consistency options (ONE, QUORUM and ALL) against 19 CaLibRe instances with a consistency option ONE. Performance is evaluated by measuring read and write latencies and consistency is evaluated for each level by counting the number of stale reads. In order to simulate a significative number of stale reads, a partial update propagation mechanism was injected into the Cassandra and CaLibRe cluster to account for the system performance under this scenario [13]. Hence, during update operations, instead of propagating the update to all 3 replica, the update will be propagated to only 2 of the replicas.

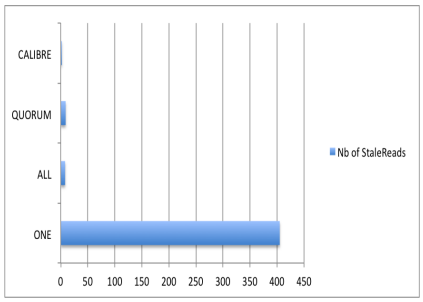
[3] `http://aws.amazon.com`



(a) Read Latency



(b) Write Latency



(c) Stale Reads

Fig. 3: CaLibRe Performance Evaluation

Test Evaluation Figures 3a, 3b and 3c respectively show the evaluation of Read Latency, Write Latency and the number of Stale Reads of Cassandra with different consistency options against CaLibRe: Cassandra with LibRe protocol. In 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 indicated 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 our implementation of the LibRe protocol developed inside Cassandra. Due to the injection of the partial update propagation, ROWA (Read One, Write All) principle could not be tested, as writes would always fail.

The read latency graph in figure 3a, shows that the 95th Percentile Latency of CALIBRE is similar to 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 are slightly higher when compared to Cassandra with consistency level ONE but better than the consistency op-

tions QUORUM and ONE-ALL. This is due to the fact that LibRe protocol imposes an additionnal call to the registry for all requests.

The write latency graph in figure 3b shows that the 95th percentile write latency of CALIBRE is the same as the 95th percentile latency of QUORUM, and that CALIBRE is faster in other metrics: 99th Percentile, Minimum and Average latencies of QUORUM. However, while comparing to the entities ONE and ONE-ALL, some of the write latency metrics of CALIBRE are slightly higher (but are not significant). This is due to the fact that both in ONE and ONE-ALL, writes need only one acknowledgement from a replica node. In CALIBRE also writes need only one acknowledgement but there is an extra messaging service in the background.

Graph 3c shows the number of stale reads for each level of consistency. Cassandra with consistency level ONE shows the highest number of stale reads. There were a few stale reads in the other consistency options, but these numbers are negligible when compared to the total number of requests. From these results, it is possible to conclude that CaLibRe offers a level of consistency similar to one provided by the QUORUM and ONE-ALL levels with better latency.

4 Related Works

Quorum systems are well studied in the literature. There are multiple works aiming at improving the performance and reliability of quorum systems [16,15,2]. However, in all these works, a sufficient amount of nodes has to be contacted in order to satisfy the intersection property. Apart from the works on quorum systems, there are also a few works in the literature whose approaches are similar to some of the approaches followed in the LibRe protocol. One of the most famous work that has similar approach of the LibRe protocol is the 'NameNode' of the Hadoop Distributed File System (HDFS) [20].

The HDFS NameNode manages metadata of files in the file system and helps to locate needed data in the cluster. But, on the contrary to the LibRe registry, which maintains metadata about small data items, the HDFS NameNode manages metadata of large file blocks. In addition to this, the NameNode is a centralized registry that stores information about the whole cluster, which can make the whole system unavailable in case of failure of the NameNode. In our approach, the LibRe registry only stores the location of partially propagated writes, and in case of failure of a registry node, availability of the system is not affected.

BigTable [4], which is a data store designed by Google, uses a two level lookup before contacting the actual data node for accomplishing reads and writes. In BigTable, the UserTable that needs to be contacted for accomplishing reads and writes is found by looking at a ROOT tablet followed by a METADATA tablet. This enables to have a scalable and fast lookup. The earlier version of HBase [8], which is an open source implementation of BigTable, used a similar two-level lookup for finding data in the system. In the later version, the two-level lookup is reduced to a single lookup in the METADATA table. However, BigTable and

HBase ensure strong consistency, so there is no context of stale replicas in these data stores. In LibRe, we use a single lookup to identify a fresh replica node only for reading some data items that are configured to use LibRe protocol.

In [21], Tlili et al. designed a reconciliation protocol for collaborative text editing over a peer to peer network using a Distributed Hash Table (DHT). According to the protocol, for each document, a master peer is assigned via the lookup service of the DHT. The master peer holds the last modification timestamp of the documents in order to identify missing updates of a replica peer in order to avoid update conflicts. This master-peer assignment is similar to the *Registry Node* assignment in the LibRe protocol. However, in LibRe, the registry node holds the version-id of the recent update and the replica nodes holding it in order to avoid reading from a stale replica.

The Global sequence protocol designed by Burckhardt et al. in [3] uses two states for an update: known sequence and a pending updates sequence. When an update is issued by a client, the update is kept in the pending updates list, and broadcasts its origin and a sequence number to all the replicas. Once an echo is received confirming all the needed copies received the update, the particular update from the pending updates list is removed. Similarly, in LibRe, when an update is applied on a replica copy, the version-id of the update along with the replica id is kept in a *Registry*. Once a confirmation is received from all the replica nodes, the entry for the corresponding data item will be removed from the registry. However, GSP focusses on ordering the write operations, whereas LibRe focusses on reading the value of the recent write.

The PNUTS Database [5] from Yahoo uses a per-record mastership over per-table or per-tablet mastership and forwards all updates of the record to this master in order to provide timeline consistency during read operations. In contrast, LibRe allows any replica to process an update and chooses a registry node per data item in order to identify the most recent version of the data item.

5 Conclusion and Future Works

The work described in this document aims at enhancing the tradeoffs between Consistency, Latency and Availability of an eventually consistent Key-Value store. Our protocol: LibRe prevents the system from forwarding read requests to the replica nodes that contain stale replica for the needed data item. In order to identify replica nodes that contain stale replicas, LibRe uses a monotonically increasing version-id for each data item. The initial implementation of LibRe protocol was developed inside Cassandra NoSql data store. This so-called 'CaLibRe' implementation offers LibRe protocol as an additional consistency option for Cassandra storage system.

The performance of the CaLibRe implementation was benchmarked against the native consistency options of Cassandra using YCSB on a 19 nodes CaLibRe and Cassandra cluster. The performance results prove that CaLibRe provides lower request latency compared to the strong consistency levels offered by Cassandra, combined to a similar number of stale reads. Hence we can safely

conclude that using the LibRe protocol gives a new tradeoff between consistency, latency and availability. However, the performance results were not tested under nodes joining or leaving the clusters. During such events, LibRe protocol would experience temporary inconsistency, which has to be studied in the future works.

Additional works are required to optimize the performance of the CaLibRe implementation. Another perspective to this work is to study the influence of the nature of the version-id (timestamp, version vector, vector clocks, ...). Also, evaluating the LibRe performance under a real world use case is considered.

References

1. Abadi, D.J.: Consistency tradeoffs in modern distributed database system design: Cap is only part of the story. Computer 45(2), 37–42 (2012)
2. Agrawal, D., El Abbadi, A.: The generalized tree quorum protocol: An efficient approach for managing replicated data. ACM Trans. Database Syst. 17(4), 689–717, http://doi.acm.org/10.1145/146931.146935
3. Burckhardt, S., Leijen, D., Protzenko, J., Fähndrich, M.: Global sequence protocol: A robust abstraction for replicated shared state. Tech. rep., Microsoft Research (2015), http://research.microsoft.com/apps/pubs/default.aspx?id=240462
4. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: A distributed storage system for structured data. ACM Transactions on Computer Systems (TOCS) 26(2), 4 (2008)
5. Cooper, B.F., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H.A., Puz, N., Weaver, D., Yerneni, R.: Pnuts: Yahoo!'s hosted data serving platform. Proc. VLDB Endow. 1(2), 1277–1288 (Aug 2008), http://dx.doi.org/10.14778/1454159.1454167
6. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with ycsb. In: Proceedings of the 1st ACM symposium on Cloud computing. pp. 143–154. SoCC '10, ACM, New York, NY, USA (2010), http://doi.acm.org/10.1145/1807128.1807152
7. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: amazon's highly available key-value store. SIGOPS Oper. Syst. Rev. 41(6), 205–220 (Oct 2007), http://doi.acm.org/10.1145/1323293.1294281
8. George, L.: HBase: The Definitive Guide. O'Reilly Media, 1 edn. (2011)
9. Gilbert, S., Lynch, N.: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. SIGACT News 33(2), 51–59 (Jun 2002), http://doi.acm.org/10.1145/564585.564601
10. Hunt, P., Konar, M., Junqueira, F.P., Reed, B.: Zookeeper: Wait-free coordination for internet-scale systems. In: Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference. pp. 11–11. USENIXATC'10, USENIX Association, Berkeley, CA, USA (2010), http://dl.acm.org/citation.cfm?id=1855840.1855851
11. Klophaus, R.: Riak core: building distributed applications without shared state. In: ACM SIGPLAN Commercial Users of Functional Programming. pp. 14:1–14:1. CUFP '10, ACM, New York, NY, USA (2010), http://doi.acm.org/10.1145/1900160.1900176

12. Kumar, S.P., Chiky, R., Lefebvre, S., Soudan, E.G.: Libre: A consistency protocol for modern storage systems. In: Proceedings of the 6th ACM India Computing Convention. pp. 8:1–8:9. Compute '13, ACM, New York, NY, USA (2013), http://doi.acm.org/10.1145/2522548.2522605
13. Kumar, S., Lefebvre, S., Chiky, R., Soudan, E.: Evaluating consistency on the fly using ycsb. In: IWCIM, 2014. pp. 1–6 (Nov 2014)
14. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. SIGOPS Oper. Syst. Rev. 44(2), 35–40 (Apr 2010), http://doi.acm.org/10.1145/1773912.1773922
15. Malkhi, D., Reiter, M.: Byzantine quorum systems. pp. 569–578. STOC '97, ACM (1997), http://doi.acm.org/10.1145/258533.258650
16. Malkhi, D., Reiter, M., Wright, R.: Probabilistic quorum systems. pp. 267–273. PODC '97, ACM (1997), http://doi.acm.org/10.1145/259380.259458
17. Marc Shapiro, Nuno Preguica, C.B., Zawirski, M.: A comprehensive study of convergent and commutative replicated data types. RR-7506, INRIA (2011)
18. Naor, M., Wool, A.: The load, capacity, and availability of quorum systems. SIAM J. Comput. 27(2), 423–447 (Apr 1998), http://dx.doi.org/10.1137/S0097539795281232
19. Saito, Y., Shapiro, M.: Optimistic replication. ACM Comput. Surv. 37(1), 42–81 (Mar 2005), http://doi.acm.org/10.1145/1057977.1057980
20. Shvachko, K., Kuang, H., Radia, S., Chansler, R.: The hadoop distributed file system. In: Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on. pp. 1–10 (May 2010)
21. Tlili, M., Akbarinia, R., Pacitti, E., Valduriez, P.: Scalable p2p reconciliation infrastructure for collaborative text editing. In: Advances in Databases Knowledge and Data Applications (DBKDA), 2010 Second International Conference on. pp. 155–164 (April 2010)
22. Voldemort, P.: Physical architecture options. http://www.project-voldemort.com/voldemort/design.html (April 2015)
23. Vukolic, M.: Remarks: The origin of quorum systems. Bulletin of the EATCS 102, 109–110 (2010), http://dblp.uni-trier.de/db/journals/eatcs/eatcs102.html
24. Zhang, H., Wen, Y., Xie, H., Yu, N.: Distributed Hash Table - Theory, Platforms and Applications. Springer Briefs in Computer Science, Springer (2013), http://dx.doi.org/10.1007/978-1-4614-9008-1