Integration Challenges of Pure Operation-based CRDTs in Redis

Georges Younes*
FCT, Universidade Nova de Lisboa
Lisboa, Portugal

Ali Shoker†
HASLab / INESC TEC &
Universidade do Minho
Braga, Portugal

Paulo Sérgio Almeida†
HASLab / INESC TEC &
Universidade do Minho
Braga, Portugal

Carlos Baquero†
HASLab / INESC TEC &
Universidade do Minho
Braga, Portugal

ABSTRACT

Pure operation-based (op-based) Conflict-free Replicated Data Types (CRDTs) are generic and very efficient as they allow for compact solutions in both sent messages and state size. Although the pure op-based model looks promising, it is still not fully understood in terms of practical implementation. In this paper, we explain the challenges faced in implementing pure op-based CRDTs in a real system: the well-known in-memory cache key-value store Redis. Our purpose of choosing Redis is to implement a multi-master replication feature, which the current system lacks. The experience demonstrates that pure op-based CRDTs can be implemented in existing systems with minor changes in the original API.

CCS Concepts

•**Theory of computation** → *Distributed algorithms;*

Keywords

CRDT; Eventual Consistency; Pure operation-based CRDTs

*European Union Seventh Framework Program (FP7/2007-2013) under grant agreement 609551, SyncFree project.

†Project "TEC4Growth - Pervasive Intelligence, Enhancers and Proofs of Concept with Industrial Impact/NORTE-01-0145-FEDER-000020" is financed by the North Portugal Regional Operational Programme (NORTE 2020), under the PORTUGAL 2020 Partnership Agreement, and through the European Regional Development Fund (ERDF).

PMLDC '16, July 17 2016, Rome, Italy

ISBN 978-1-4503-4775-4/16/07…$15.00

DOI: http://dx.doi.org/10.1145/2957319.2957375

1. INTRODUCTION

Eventually consistent replication using Conflict-free Replicated Data Types (CRDTs) has been widely adopted by the industry [6, 2, 3]. CRDTs are a materialization of the Strong Eventual Consistency model, which allows all system replicas to promptly perform read/write operations, leaving the synchronization to a background phase to achieve eventual consistency. More recently, the *pure operation-based CRDT* model [2, 1] has been introduced as a *generic* model to build CRDTs that can leverage the meta-data provided by the middleware and *reduce the overhead* of communication through exchanging more compact messages. However, although pure op-based CRDTs have been theoretically outlined in [2], it is not yet understood how to implement them in concrete systems.

In this paper, we summarize our experience in implementing pure op-based CRDTs to build a multi-master replication feature in Redis [5], the famous in-memory cache system. We choose Redis because: (1) it helps us demonstrate how to integrate pure op-based CRDTS in an existing popular system; (2) we contribute with the community in building a crucial multi-master feature that is currently missing in Redis. Our experience reveals that implementing pure op-based CRDTs for many data types is straightforward, whereas integrating them in an existing system is somehow challenging if the aim is to preserve the legacy API and code-base intact.

2. BACKGROUND

2.1 Redis

Redis [5] is an in-memory cache and key-value store that is widely used by well known companies such as Twitter, GitHub, StackOverflow, Flickr, etc [5]. It significantly reduces the response time of services as it usually stands as a layer between the persistent storage of a service and clients; thus, instead of having to query the database each time data is needed, Redis allows caches the data in memory enabling much faster data retrieval. Redis supports many primitive datatypes like sets, counters, maps, etc. Redis server is implemented in ANSI C and works in most POSIX systems [5]; whereas several Redis clients are available in most of the

programming languages [5].

The popularity of Redis stems from its speed, rich semantics, and stability. It was primarily designed and generally used in a single server deployment model. A single Redis instance is a remote centralized solution, and not a distributed system. However, Redis provides distributed solutions known as Redis Sentinel and Redis Cluster which allow using multiple instances, asynchronous master-slave replication, and automatic fail over process.

Redis Sentinel [5] is a solution to manage Redis instances by monitoring them, notifying other nodes in case of a failover, and starting a failover process. It is mostly used for managing master and slave nodes, notifying a change in the behaviour, handling the failover if a master is down, starting the election of new master, and in reconfiguration. On the other hand, Redis Cluster [5] is a solution for data sharding with failover mechanism and replication as well. It is mostly used when the dataset does not fit on a single machine.

As both replication methods use an asynchronous master-slave replication, which may cause some issues as we described earlier, we found it interesting to have a multi master replication feature in Redis, to improve high availability, through using Pure op-based CRDTs [2] that we summarize in the next section.

2.2 Pure Op-based CRDTs

Conflict-free Replicated Data Types (CRDTs) [6] are used to replicate data across multiple replicas without immediate synchronization between them. They are designed in a formal way to resolve conflicts between replicas using eventual consistency (EC). There are two approaches for CRDTs: Operation-based CRDTs and State-based CRDTs [6]. The former broadcasts the update operation itself while the latter broadcasts the local state to the other replicas. In the "Classical" operation-based approach in [6], the replica sends not only the operation and arguments but also meta-data containing information that will be needed to achieve convergence. This design may result in large state size and leads to confusion with the state-based approach. Pure operation-based CRDTs [2] were introduced as a solution that offers very compact state size and a generic framework, independent of the data type.

In the pure op-based CRDT approach, every operation is applied locally and disseminated to all other replicas via a *Tagged Reliable Causal Broadcast* middleware [2]. The compact size of the message is given by the fact that the meta-data needed to preserve the causal order is provided by the middleware. As a result, the disseminated message size is reduced to containing only the operation name and arguments.

Each pure op-based CRDT has the following specification:

- **prepare**: returns the operation and arguments needed for the dissemination.
- **effect**: removes redundant operations from the POLog and checks for stable operations to move them to the sequential data type.
- **eval**: returns the result of the query from the state by combining the data of the POLog and the sequential data type.

3. ARCHITECTURE

In order to implement the multi-master replication feature using pure op-based CRDTs, we integrated three layers into Redis Server's code. We present below the three layers, shown in Figure 1, with a brief description of each layer.

Request handler.

The handler layer is the intermediate layer between Redis Client API and the TRCB and CRDT layers. Every client request is redirected by the original Redis Client API to this layer. Then, this layer prepares a client object containing the operation and arguments (if any), serializes it and ships it to the TRCB layer for dissemination.

Tagged Reliable Causal Broadcast.

The second layer (TRCB) is used to broadcast client requests (operations and arguments) to all nodes in the cluster. This layer tags the client objects received from the handler with a timestamp needed to guarantee causal delivery at each node. Also, the TRCB is implemented in a way to guarantee exactly-once delivery of each client object to each node in the cluster.

CRDT layer.

The CRDT layer is where we implement the pure op-based CRDTs and their related structures, such as the POLog (Partially-Ordered Log), a map where the key is a timestamp and the value is the operation and arguments, following the designs and specifications in [2]. In addition to that, we perform a two-phase POLog compaction to make CRDTs even more efficient. The first phase removes obsolete and redundant information from the POLog in a way that does not affect the result of the queries, keeping only relevant of operations. The second one is by using *causal stability* information from the middleware to discard the timestamps of causally stable operations and move them from the POLog to the Redis data types.

4. CHALLENGES

In order to implement a multi-master replication feature in Redis using pure op-based CRDTs we had to figure out solutions for the challenges we faced. The challenges are in terms of architecture design, preserving original API with minor changes, reusability of Redis code, configurability of the system, choice of messaging pattern and implementation of causal stability. We address, in each of the subsections below, these challenges as well as our solution for each of them.

4.1 Design

A major challenge was the choice of the architecture design, as shown in section 3. On one hand, implementing pure op-based CRDTs on top of Redis allows us to use Redis primitive data types as sequential data types for our CRDTs, but this would require us to implement a new client API. On the other hand, implementing Redis on top of pure op-based CRDTs preserves the same Redis client API and uses CRDTs to store the data, but we do not benefit from Redis' data types and related APIs. Instead, we integrated, within Redis code, three layers responsible for the replication process, keeping the same client API and benefiting from Redis's data types as sequential data types in our pure op-based model.

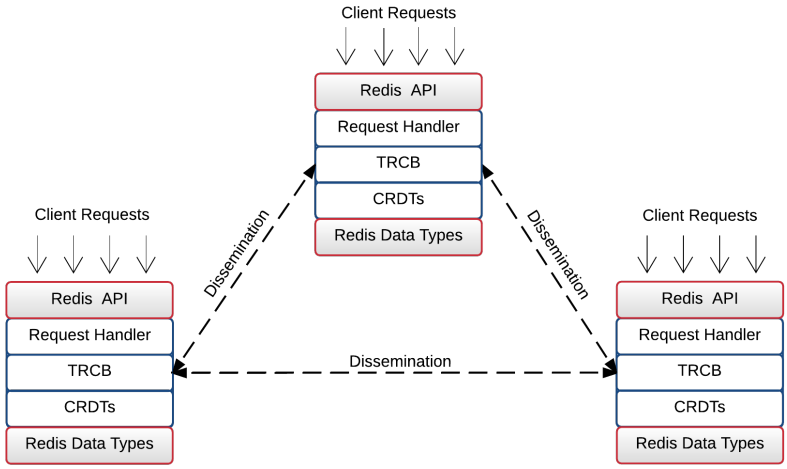

Figure 1: The general architecture of our multi-master proposed solution.

Moreover, from the multithreading perspective, understanding the way these layers work and how they interact with each other is crucial for the pure op-based model to work both correctly and efficiently. The handler layer is always waiting for client requests, independently from the TRCB and CRDT layers. Both the TRCB and CRDT layers work independently and in parallel. The TRCB constantly reads from two buffers (UNIX file descriptors): one containing the client requests and another the peer updates. The TRCB layer has a subthread, using the subscribe pattern, listening to peer updates, which runs continuously and independently from the main TRCB thread which deals with causal reliable dissemination. The CRDT layer reads operations delivered by the TRCB and adds them to the POLog. It has a subthread responsible of the 2-phase POLog compaction process.

4.2 Preserving Redis client API

We implemented most of Redis data types such as keys, strings, sets, hashes, bitmaps and hyperloglogs due to the ability to map these data types to pure op-based CRDTs. In the current version we do not support sorted sets and lists.

The main factor that makes this possible is that the use of CRDTs is only transient and for replication purposes; the storage happens in Redis data types. This preserves the original behaviour of Redis from the application developer's perspective.

We use the Redis Set as an example to illustrate the changes we made to implement the pure op-based ORSet specification.

A Redis Set is an unordered collection of strings. The operation to add an element to a set is *SADD* and is used as *SADD myset element1*, returning 1 if *element1* was added to the set, and 0 if not. The *SREM* command removes an element from the set using *SREM myset element1*, returning 1 if *element1* was removed from the set, and 0 if not. The command *SMEMBERS* allows the client to query the set, returning all the elements in the set.

We use a pure op-based observed-remove set (ORSet) to map the original Redis Set. A pure op-based ORSet has a POLog and the Redis Set itself used as a sequential data type. No changes were made to API: the operations are added to the POLog until they are causally stable and then added to/removed from the Redis Set using *SADD*/*SREM*, returning 1/0 in case of success/failure. For *SMEMBERS*, as in the pure op-based model, elements can exist in both the POLog and in the sequential data type. *SMEMBERS* behaves as the original Redis API, and returns the number of elements by combining those in the POLog with those in the Redis Set.

4.3 Causal Stability Implementation

The pure op-based model uses the notion of causal stability, to discard timestamp information of operations once they become stable and to move them from the POLog to the sequential data type.

A clock t, and corresponding message, is causally stable at node i when all messages subsequently delivered at i will have timestamp $u \geq t$.

In order to detect causal stability we designed a mechanism using two new structures in the TRCB layer, at each node i, in order to implement this notion of causal stability. The first one is an $N \times N$ matrix called Last Timestamp Matrix (LTM), where N is the number of nodes and each row j of the LTM is the version vector of the most recently delivered message from the node j. The second structure is a version vector called *Stable Version Vector*(SVV). At each node i, SVV_i is the pointwise minimum of all version vectors in the LTM. Each operation in the POLog that causally precedes (happend-before) the SVV is considered stable and removed from the POLog, to be added to the sequential data type.

4.4 TRCB communication

The implementation of the pure op-based CRDT model requires a dissemination middleware that guarantees an exactly-once delivery that respects causal order. We used the Publish/Subscribe messaging pattern where each node is subscribed to all the others because it meets best our asyn-

chronous multi-master broadcast requirements. We tried to use the already existing pub/sub communication implementation of Redis Cluster and found it was not feasible for two reasons. The first reason is that Pub/Sub in Redis Cluster works by broadcasting every publish to every other Redis Cluster node through a cluster bus. This limits the pub/sub throughput to the bisection bandwidth of the underlying network infrastructure divided by the number of nodes times message size. Pub/sub thus scales linearly with respect to the cluster size, but in the the negative direction. The second is that the Cluster bus binary protocol is not publicly documented since it is not intended for external software devices to talk with Redis Cluster nodes using this protocol.

Many libraries such as RabbitMQ[1] [4] provide features like reliability, but in order to obtain it can lead to trade-offs in performance. Instead, in order to allow us to have more control over the implementation trade-offs, we used ZeroMQ[2] [7] as it is much lightweight and decided to implement a reliable causal delivery mechanism, enhanced with tagging, over ZeroMQ best-effort Pub/Sub.

4.5 Reusability of Redis code

In the pure op-based CRDT specification in [1], each CRDT uses two main structures: the POLog and the sequential data type. However, the ability to map most of Redis primitive data types to pure op-based CRDTs allowed us to use the Redis data types as sequential data types for the CRDT, with the CRDTs being used only for dissemination and to store data temporarily, before it is stored in Redis when causal stability is achieved.

In some cases we didn't even need to change the command implementation. Considering the previous example in subsection 4.2, the command implementation for both *SADD* and *SREM* were used as is. For *SMEMBERS*, we had to change the implementation as we needed to read elements from both the POLog and the Redis Set and return the combination to the client.

4.6 Configurability

We tried to keep Redis as configurable as possible in the same intuition of Redis Cluster. A Redis server can be used as a normal single instance or as a node in a multi-master cluster depending on the configuration.

5. CONCLUSION

We showed how pure op-based CRDTs can be integrated in Redis with minor changes in the original system's API. Three lessons we learned. First, pure op-based CRDTs are fairly easy to implement, being generic across multiple data types. Second, integrating them in an existing system is challenging if keeping the legacy API intact is an objective. Third, the modular design we used in this implementation makes this model easy to mimic and thus implement multi-master replication in other systems with few changes.

[1]RabbitMQ is an open source message broker software (sometimes called message-oriented middleware) that implements the Advanced Message Queuing Protocol (AMQP).

[2]ZeroMQ (also spelled ∅MQ, 0MQ or ZMQ) is a high-performance asynchronous messaging library, aimed at use in distributed or concurrent applications.

6. REFERENCES

[1] C. Baquero, P. S. Almeida, and A. Shoker. Making operation-based crdts operation-based. In *Proceedings of the First Workshop on Principles and Practice of Eventual Consistency*, page 7. ACM, 2014.

[2] P. S. A. Carlos Baquero and A. Shoker. Making operation-based crdts operation-based. In *Distributed Applications and Interoperable Systems - 14th IFIP WG 6.1 International Conference, DAIS 2014, Held as Part of the 9th International Federated Conference on Distributed Computing Techniques, DisCoTec 2014, Berlin, Germany, June 3-5, 2014, Proceedings*, pages 126–140, 2014.

[3] Paulo Sergio Almeida, Ali Shoker, and Carlos Baquero. Efficient State-based CRDTs by Delta-Mutation. In *Proceedings of the International Conference of Networked sYStems*, NETYS'15. Springer, May 2015.

[4] RabitMQ. Rabbitmq - messaging that just works. https://www.rabbitmq.com.

[5] Redis. Redis Documentation. http://redis.io/documentation.

[6] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Technical report, jan 2011.

[7] ZeroMQ. Distributed Messaging - zeromq. http://zeromq.org.