Eventual Consistency with Conflict-free Replicated Data Types (CRDTs)

Dui Lin dui@g.ucla.edu
Sheng Zhang shengzhang@g.ucla.edu
Yu Cai caiyu0227@g.ucla.edu

Abstract: With the growth of cloud applications, the data needs to be distributed in multiple machines to fight against possible node failure. Classical distributed systems do not allow concurrent updates on same object to occur by imposing strong synchronisation operations, thus ensuring a consistent state. However, this reduces the scalability and usability of many systems by limiting the concurrency. In this paper, we introduce the eventual consistency realized by Conflict-free Replicated Data Types (CRDTs), which are mergeable object types allowing concurrent updates on an object while eventually converging to a consistent state without losing any update. And we further implement state-based grow-only counter (G-Counter), positive-negative counter (PN-Counter), grow-only set (G-set) in Java. These data types can be integrated into lots of current distributed system and provide eventual consistency.

Keywords: Eventual Consistency, Replicated Shared Objects, Distributed Systems, Concurrency Data Types

I. Introduction

In distributed system, data replication is an important way to ensure consistency. There are basically two reasons for replicating data. First, data are replicated to increase the reliability of a distributed system [1]. If a file system has been replicated, it could continue working after one replica crashes by simply switching worker node to another, or when a write operation fails, we can safeguard ourselves against a single, failing write operation by considering the value that is returned by at least majority of copies as being the correct one. Second, replicating data can improve system performance [2], which is important when a distributed system needs to scale in terms of size or geographical area it covers. For example, when an increasing number of processes needs to access data that are managed by a single server, in that case, performance can be improved by replicating the server and subsequently dividing the workload among the processes accessing the data.

However, having multiple copies of data may lead to consistency problems. Therefore, we introduce consistency model that is essentially a contract between processes and the data store. Consistency models can be categorized into two types, strong and weak consistency models:

- Strong consistency models (capable of maintaining a single copy)
- Weak consistency models (In this paper, this refers to eventual consistency)

Strong consistency models can guarantee that the apparent order and visibility of updates is equivalent to a non-replicated system, while weak consistency model, on the other hand, do not make such guarantees. There are many cases in which strong consistency guarantees need to be met, such as in the case of electronic stock exchange and bank transfer, even ensuring these properties may lead to less availability and system performance. However, some cases like worldwide naming system (DNS) can be viewed as large scale distributed system and

replicated databases that tolerate a relatively high degree of inconsistency. They have in common that if no updates take place for a long time, all replicas will gradually become consistent, that is, have exactly the same data stored. Therefore, behaving like a single system by default is not that desirable. What we want is a system where we can write code that does not use expensive coordination, and yet returns a "usable" value. Instead of having a single truth, we will allow different replicas to diverge from each other - both to keep things efficient but also to tolerate partitions - and then try to find a way to deal with the divergence in some manner. This form of consistency is called eventual consistency, Figure 1.1 shows how eventual consistency works [3].

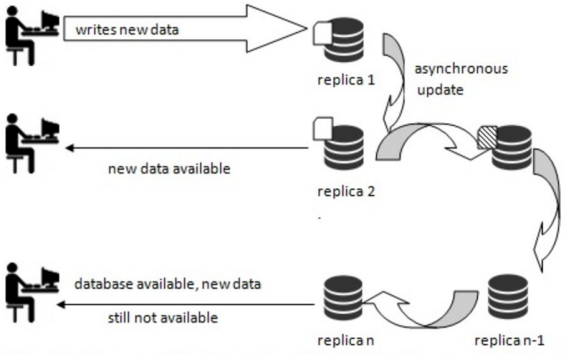

Figure 1.1. Eventual consistency in work flow

Within the set of systems providing eventual consistency, there are two types of system designs [4]:

- Eventual consistency with probabilistic guarantees. This type of system can detect conflicting writes at some later point, but does not guarantee that the results are equivalent to some correct sequential execution. In other words, conflicting updates will sometimes result in overwriting a newer value with an older one and some anomalies can be expected to occur during normal operation (or during partitions). In recent years, the most influential system design offering single-copy consistency is Amazon's Dynamo, which is also an example of the system that offers eventual consistency with probabilistic guarantees.
- Eventual consistency with strong guarantees. This type of system guarantees that the results converge to a common value equivalent to some correct sequential execution. In other words, such systems do not produce any anomalous results; without any coordination you can build replicas of the same service, and those replicas can communicate in any pattern and receive the updates in any order, and they will eventually agree on the end result as long as they all see the same information.

Here, we are using Conflict-free Replicated Data Types (CRDTs) to realize eventual consistency. A common property of all CRDTs is that they will converge eventually, so if two replicas have seen the same set of updates they are guaranteed to be in the same state. This guarantee is also preserved when replicas are not synchronized for a longer period of time and therefore, CRDT is a good approach to address eventual consistency.

This paper is organized into three parts. The first part contains this introduction and chapter 2, which review and discuss the general background of our work: eventual consistency in distributed system and CRDTs. The second part focuses on the implementation of CRDTs, especially about the Counters and Sets, the programming language we choose is Java, which is popular in industry and can be easily integrated into most distributed systems. The third part concludes the paper and outlooks the future work.

II. Conflict-free Replicated Data Types (CRDTs)

The most basic problem now is what data types can achieve eventual consistency. So far, CRDT is the answer to this problem in theoretical cycle. But actually, in reality, CRDTs also have been used in online chat systems, online gambling and other distributed system applications [5].

In distributed system, there are two approaches to CRDTs, both of which can achieve strong eventual consistency and high availability: operation-based CRDTs and state-based CRDTs. In both situations, the goal is to avoid coordination through safe operations.

A. Operation-based CRDT: Commutative Replicated Data Type

In operation-based CRDTs, also known as commutative replicated data types, updates contain the operations that modify the replicas and must be executed in all replicas. A trivial example is that the leader node broadcast an increment operation among replicas. When replicas receive the updates, they apply them locally. Therefore, instead of the state changes, the state changing events are stored. These event are persistent in the log so that the replicas can be brought up to the latest state by replaying the event log. As a result, the problem like unbound growth of history can be mitigated since the history is kept in a event log, rather than the CRDTs.

The conditions of the convergence for CmRDTs are [6]:

- Reliable broadcast channel which guarantees that operations are delivered only once.
- Commutative operations that can yield predictable result without dependency.
- Operation-based objects can be more expressive and simpler since the state is actually offloaded to the channel. However, CmRDTs are demanding of the channels.

B. State-based CRDT: Convergent Replicated Data Type (CvRDT)

In state-based CRDTs, also referred to the CvRDTs, updates contain the full state of the objects. In contrast to CmRDTs, CvRDTs send updates containing their full local states to other replicas. When a replica receives updates, it merges the updates to its local state. Since every instance keeps full history of changes of the data structure, it is self-contained and fault tolerant. The merge function needs to be commutative, associative and most importantly, idempotent. This nature ensures the convergent nature of CvRDTs, which means that the state updates need not to be applied in any ordering requirements, as long as all the updates eventually reach all replicas.

The conditions for the design of CvRDTS are [6]:

- Updates to the CvRDTs are monotonic, which means that new values are always greater than before or always less than before.
- Conflicting updates yield new values which are siblings to each other (equally greater than the original value).
- A resolution must always exist that allows siblings to merge into a new value.

The properties of CvRDTs make them relatively straightforward to implement in a reliable manner and they do not have too much requirements on the underlying infrastructure. However, the size of an object state is usually larger than the size of an operation. Therefore, transmitting the full state of an object is costlier.

C. Relation Between CmRDTs and CvRDTs

State-based approach (CvRDT) is simpler to design and implement since the full state determine all the necessary information. Also, they only depend on unreliable propagation channel, so that clients can go offline, doing updates and then merge its state. Once shared state is known, old state can be collected. However, garbage collection in a consistent way across all nodes becomes a problem in CvRDTs [7].

Operation-based approach (CmRDT) is more complex since the implementation should consider the history. Also it is more demanding

of the propagation media to have reliable broadcast. But conversely, it is easier to create dynamic clusters where nodes can leave and rejoin at any time [7]. In CvRDTs, a full replication need of the database need to be performed while in CmRDTs, you just redo all the update in the event history from the last snapshot.

III. Implementation

Basically, there are two ways to implement CRDTs: Convergent Replicated Data Types and Commutative Replicated Data Types. For the Convergent-Only Replicated Data Type, one thing we should mention is the Register. There are only two operations Set and Get that we can perform on it. For Register, there are two ways to implement it. One is Last-Write-Win (LWW) [8]. For LWW, we keep the value and a timestamp of when the value was set on a given replica. The timestamp allows the value to be merged on any given replica by comparing the local replica timestamp with that of the one to be merged. The local value can be updated if the timestamp of the local value is older than the sent value that is to be merged. Another is Multi-Value. We may keep multiple copies of the same conceptual value, but maintain information that indicates when the value came to be set. It is possible to hand a client multiple values of the same thing and allow the client to apply conflict resolution to determine which of the multiple values it shall choose as current value. The advantage of Multi-Value is, it provides the most versatile, but also the most complex approach.

Here are some Commutative Replicated Data Types. The first one is Counters. There are two types of counters, one is G-Counter. It is a grow-only counter. And the other is Set. For Counter, one that only apply addition. The implementation of a G-Counter basically implements a Version Vector or Vector Clock[5]. As shown in the Figure 3.1, to explain how G-Counter works, we consider its operations on two replicas.

- R1 increments the G-Counter A, yielding a value of 1
- R1 increments the G-Counter A again, yielding a value of 2
- R2 increments the G-Counter A for the first time, yielding a value of 1

Now, the increment operations are sent from R1 to R2, and from R2 to R1:

- R2 receives an increment notification from R1 and increments G-Counter A from 1 to 2
- R2 receives an increment notification from R1 and increments G-Counter A from 2 to 3
- R1 receives an increment notification from R2 and increments G-Counter A from 2 to 3
- Both R1 and R2 are now up to date with an A value of 3

Another is PN-Counter. It is a positive-negative counter, one that understands how to add and subtract. To support this there are actually two G-Counters that are used to maintain a Positive G-Counter and a Negative G-Counter.

Now consider an example of how the PN-Counter works between two replicas.

- R1 increments PN-Counter B, yielding a PN value of {1,0}
- R1 increments PN-Counter B, yielding a PN value of {2,0}
- R2 increments PN-Counter B, yielding a PN value of {1,0}
- R2 decrements PN-Counter B, yielding a PN value of {1,1}

Now, the increment and decrement operations

are sent from R1 to R2 and from R2 to R1.

- R2 receives an increment notification from R1 and increments its P counter from 1 to 2, yielding {2,1}
- R2 receives an increment notification from R1 and increments its P counter from 2 to 3, yielding {3,1}
- R1 receives an increment notification from R2 and increments its Pcounter from 2 to 3, yielding {3,0}
- R1 receives a decrement notification from R2 and increments its N counter from 0 to 1, yielding {3,1}
- Both R1 and R2 are now up to date with a value of B {3,1}. The actual value of the PN-Counter B is yielded by subtracting the N value from the P value. So 3-1 is 2, which is the actual value of PN-Counter B.

For G-set (grow-only set), is implemented very much like a G-Counter. Again, we will examine the use of a G-Set on two replicas:

- R1 adds the value 1 to G-Set C, yielding a set of {1}
- R1 adds the value 2 to G-Set C, yielding a set of {1,2}
- R2 adds the value 3to G-Set C, yielding a set of {3}

Now the add operations are sent from R1 to R2, and from R2 to R1:

- R2 receives an [add 1] notification from R1 and R2 adds the value 1 to G-Set C, yielding {1,3}
- R2 receives an [add 2] notification from R1 and R2 adds the value 2 to G-Set C, yielding {1,2,3}
- R1 receives an [add 3] notification from R2 and R1 adds the value 3 to G-Set C, yielding {1,2,3}
- Both R1 and R2 are now up to date with a set of C as {1,2,3}

Besides what we mentioned above, there also exists a more complex data type, Directed Graph CRDT [5, 9]. Important algorithms and applications like shortest-path or web page-rank work on graphs, which indicates its importance. But due to the complication, we decide not to implement it in our final work.

Finally, we use Java to implement several important data types mentioned before: G-Counter, PN-Counter and G-Set. For the G-Counter, we use Map and HashMap data structure to implement it. Including the increment (increment a given key) method, get (get the counter value) method, merge (merge another counter into this one) method. Likewise, for the PN-Counter, we have added the decrement method. And some slightly changes made to get method and merge method.

IV. Conclusion

In this paper, we present the current study of the CRDTs. We introduced both the definition and the implementation of CRDTs, including operation-based and state-based approach. In the third part of our paper, we briefly introduce the basic knowledge of our implementation. Some example like sets and counters are specifically mentioned in our paper. Besides, the implementation part is shown in our code write in Java. We have not test the outcome of the implementation (e.g. time complexity and space complexity) due to the current situation we were in (limited time and hardware unavailable).

Future work in CRDTs may focused on the implementation of Directed Graph CRDTs. We know that the directed graphs are really important in future network analysis and communication. Being able to implement the

Directed Graph CRDT may significantly impact the future large-scale asynchronous systems.

Appendix

1. Code: https://github.com/aduispace/lovecrdt

References

[1] Bailis, P., & Ghodsi, A. (2013). Eventual consistency today: Limitations, extensions, and beyond. *Communications of the ACM*, *56*(5), 55-63.

[2] Tanenbaum, A. S., & Van Steen, M. (2007). *Distributed systems: principles and paradigms*. Prentice-Hall.

[3] Vogels, W. (2009). Eventually consistent. *Communications of the ACM*, *52*(1), 40-44.

[4] Lakshman, A., & Malik, P. (2010). Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, *44*(2), 35-40.

[5] Shapiro, Marc; Preguiça, Nuno; Baquero, Carlos; Zawirski, Marek (13 January 2011). "A Comprehensive Study of Convergent and Commutative Replicated Data Types". *RR-7506*. HAL - Inria.

[6] Shapiro, Marc; Preguiça, Nuno (2007). "Designing a Commutative Replicated Data Type". *Computing Research Repository (CoRR)*. abs/0710.1784.

[7] Letia, Mihai; Preguiça, Nuno; Shapiro, Marc (2009). "CRDTs: Consistency without Concurrency Control". *Computing Research Repository (CoRR)*. abs/0907.0929.

[8] Tauro, C. J., Ganesan, N., Easo, A. A., & Mathew, S. (2013, August). Convergent replicated data structures that tolerate eventual consistency in NoSQL databases. In Advances in Computing and Communications (ICACC), 2013 Third International Conference on (pp. 70-75). IEEE.

[9] Shapiro, M., Preguiça, N., Baquero, C., & Zawirski, M. (2011, October). Conflict-free replicated data types. In Symposium on Self-Stabilizing Systems (pp. 386-400). Springer, Berlin, Heidelberg.