Implementing Conflict-Free Replicated Data Types

Melina Mast 13-762-588 melina.mast@uzh.ch
Department of Informatics, University of Zurich

Abstract

Vertiefungsarbeit about the implementation of the Observed Remove Set and the usability possibilities for CRDTs.

1 Introduction

The increase in data in the past years led to new requirements for database vendors to make their systems distributed and scalable to a large amount of data. The distributed databases replicate data in the system to improve performance and protect against data loss. But ensuring consistency is a challenge (6).
It needs a lot of communication between the replicas to keep them in the same state and consequently is very expensive. Therefore, eventually consistent databases became more popular. Eventually consistent replicas, on the contrary to strong consistent systems, just eventually reach the same final state. Even if every replica has seen the same updates (6). In some cases eventual consistency is just not enough. It may lead to several problems such as the following:
Imagine two elements A and B are added to the database and element A is added to the first replica R_1 and element B to the second replica R_2. The result is $R_1 = \{A\}$ and $R_2 = \{B\}$. Taking the union of both replicates will lead again to a consistent database $R_1 = \{A, B\}$ and $R_2 = \{A, B\}$. However, if now a new element C is added to R_1 and the element B is removed in R_2 the new state is $R_1 = \{A, B, C\}$ and $R_2 = \{A\}$. At this point neither the union nor the intersection of the two replicas can lead to a consistent state. On the one hand, a union would result in $R_1 = \{A, B, C\}$ and $R_2 = \{A, B, C\}$ and the deleted element B would reappear. On the other hand, an intersection would cause C to disappear again. In both cases, this conflict somehow must be solved *e.g.* with another interaction like asking the user to manually solve the conflict .
Conflict-Free Replicated Data Types (CRDTs) describe a solution to automatically solve the described problem. With CRDTs all correct replicas that have delivered the same updates converge to a common, predictable state (6).

2 The Concept of CRDTs

Conflict-Free Replicated Data Types (CRDTs) are used for storing data in a distributed systems. They are data types that can be replicated across multiple nodes. Operations on only a single replica can be performed without immediately updating the other replicas. Updates can be communicated at a later time. Hence, they supply a middle way between strong consistency and high performance (2)(1).

2.1 Strong Consistency (SC)

In a *Strong Consistent (SC)* all system updates are serialised in a global total order. Therefore, in a SC system users will never see out-of-date values. Which is conflict-free but aggravates performance and scalability (6)(5).

2.2 Eventual Consistency (EC)

In an *Eventually Consistent (EC)* system a read may return an out-of-date value (5).
EC means informally, an update delivered at some replica is eventually delivered to all replicas. Further, all the replicas that have delivered the same updates just eventually reach equivalent state (6).
Practically this means updates are made locally. This improves performance. But EC systems converge not until you merge them somehow. In case of conflicts because of concurrent updates you have *e.g.* to roll back to resolve the conflicts and consequently waste resources (2)(6).

2.3 Strong Eventual Consistency (SEC)

Strong Eventual Consistency provides a stronger condition to avoid wasting resources in conflict situations (6).
An object is *Strong Eventual Consistent (SEC)* if it is on the one hand eventual consistent (EC) and in addition, ensures that replicas that have delivered the same updates have an equivalent state. In SEC systems there are methods to directly solve concurrent updates. Consequently, roll backs are not necessary (6) (1).
CRDTs are guaranteed to be in the same state after all replicas have seen the same set of updates by using methods that directly solve concurrent updates. Hence, CRDTs can provide strong eventual consistency. (2)(1).

2.4 Concurrent Update Operations

A still unaddressed question is what happens if concurrent updates are executed at the same time. For example, if in a replica an *add* operation is executed and in parallel a *remove* operation is executed. It has to be decided which operation wins. There are mainly three possibilities to solve the conflict. One is to throw an error. Another one is to always let the *add* operation win. And the third option is to always let the *remove* win in concurrent updates (1).
In Section 3 a CRDT named Observed Remove Set is presented that has two concurrent updates: *add* and *remove*. The Observed Remove Set always lets *add* win against *remove* (1)(2).

3 The Observed Remove Set

The Observed Remove Set (OR-Set) consists of a set of elements and a set of tombstones. Each of the set may contain items of the same structure. An item *(e,n)* is a tuple that contains the item identifier *e* and a unique token *n*. The set of elements represents the items that have been added to the OR-Set. The set of tombstones represents the removed items. In Figure 1 the construction of an OR-Set is illustrated. In this example the OR-Set contains the elements *e* and *f*, their corresponding tokens and no tombstones. Hence, *e* and *f* has been added to the set while no items have been deleted yet (1).

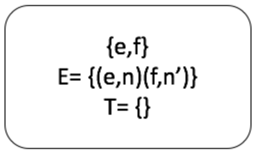

Figure 1: OR-Set elements (1).

3.1 Concept of Add-wins

When a concurrent *add* and *remove* operation over the same element occur, one among several post conditions can be chosen: add-wins, remove-wins or an error mark.
The OR-Set can also be called Add-Wins-Set since it always lets *add* operations win over *remove* operations. Therefore, the OR-Set always leads to a predictable state (1).

3.2 Implementation of the OR-Set

To implement add-wins, the OR-Set distinguishes the different *add* operations on the same element by adding a unique token. This token is stored with the element as a tuple *(e,n)*. The Listing 2 illustrates the add algorithm used in a OR-Set. First a unique token is generated. Second the set of elements is united with the added element (1).

```
Adding Algorithm
Input: element e;
Steps:

 add (element e)
     let n = unique()
     E:= E $\cup \{(e,n)\} \setminus T$
```

Listing 2: Element Addition (1).

An element can be removed by adding the corresponding pair *(e,n)* to the tombstone set illustrated in Listing 3. A remove in the OR-Set can only happen if the corresponding add operation has been executed before. Set *R* is constructed which contains the elements that should be removed from the observed elements in set *E*. In other words, the remove operation affects the already observed *add* operation. Hence, the *add* operation wins if it is executed in parallel to the *remove* operation. Because the addition is not visible to the concurrent *remove* operation. Another addition of the element with the same identifier replica has a different token and is not part of the tombstones.(2) (1).

```
Removing Algorithm
Input: element e;
Steps:
--------------------------------
 remove (element e)
     let R = $\{(e,n) | \exists n : (e,n) \in E\}$
     $E := E \setminus R$
     $T := T \cup R$
```

Listing 3: Element Removing (1).

The OR-Set merges the different replicas by exchanging the payload between them and merging them. The payload is the data that is transmitted. Here it is the set of elements and the set of tombstones. Listing 4 illustrates the merge algorithm if a replica *A* is merged to a replica *B*:
First, the elements are merged by removing the elements equal to the tombstones from replica *B*. Second, the set is united with the elements from *B* that are not included in the tombstones of the set *A*. After that, the tombstones are updated and therefore united with the tombstones of replica *B* (1).

```
Merge Algorithm
Input: OrSet B;
Steps:
--------------------------------
merge(B)
   $E = (E \setminus B.T) \cup (B.E \setminus T)$
   $T = T \cup B.T$
```

Listing 4: Merge (1).

3.3 Space Complexity

The OR-Set presented provides functions that are useful in distributed systems. But the described algorithm is hardly usable in practice, since the payload size of the OR-Set grows with every applied *add* operation (1).
The Optimized Observed Remove Set (OptOR-Set) provides a solution to this problem. The idea is described in Section 3.4 (1).

3.4 The Idea of the Optimized Remove Set (OptOR-Set)

The Optimized Observed Remove Set (OptOR-Set) provides the same functionality as the OR-Set. But the OptOR-Set uses less meta-data to provide this functionality (1).
In the OR-Set the memory requirement grows with the number of operations. However, adding a tuple *(e,n)* always happens-before removing the same tuple. Therefore, the idea of the OptOR-Set is to store happens-before information. Consequently, the OptOR-Set can discard a removed tuple immediately and can reduce the required tombstones (1).

3.5 Business Possibilities

Imagine you own an online shop that is active globally. On the one hand, it should provide high performance for all clients worldwide and on the other hand, a consistent database to provide a good user experience.
A side-effect of the OR-Set is that if a product is concurrently added and removed, the *remove* method concerns only the already observed products. Consequently, the *add* operation wins. This even could be positive for your business if the user then re-thinks his decision to remove the product and buys it instead. (1).
Altogether, the functions provided by the OR-Set can supply a practical solution from a business perspective.

4 Usability of CRDTs in Practice

Due to the enormous growth of large-scale distributed systems and the requirement to ensure availability for all users, no matter which replica they have access to, becomes more important. If availability and low access latency is more important than strong consistency, CRDTs may provide a solution (3) (1).
There are database solutions that already use CRDTs in practice. An example is *riak*. The provider has an eventually consistent NoSQL database system that also uses CRDT based data types (4).
A data type used are sets. The sets are collections of unique binary values that may for example be strings. The sets provide four operations: add an element, remove an element, add multiple elements, or remove multiple elements (4).
The example of *riak* proves that CRDTs can actually be used in practice (4).

5 Conclusion

Conflict Free Replicated Data Types (CRDTs) let a system maintain various replicas and guarantee Strong Eventual Consistency. Availability and access latency can be improved by using CRDTs in a distributed system with multiple data replicas (1).
In this work the OR-Set and his add-wins property has been described and the *add*, *remove* and *merge* operations have been explained. Besides, some possible usability fields have been proposed (1).

References

[1] A. Bieniusa, M. Zawirski, N. M. Preguic a, M. Shapiro, C. Baquero, V. Balegas, and S. Duarte. An optimized conflict-free replicated set. CoRR, abs/1210.3368, 2012.

[2] Peter Zeller. Specification and Verification of Convergent Replicated Data Types. Univerity of Kaiserslautern, 2013.

[3] P. S. Almeida, A. Shoker, C. Baquero. Efficient State-based CRDTs by Decomposition. ACM PaPEC'14, 2014.

[4] `http://docs.basho.com/riak/latest/theory/concepts/crdts/` 01/22/2016

[5] `http://docs.basho.com/riak/latest/theory/concepts/strong-consistency/` 01/22/2016

[6] Marc Shapiro, Nuno Preguica, Carlos Baquero, Marek Zawirski. Conflict-Free Replicated Data Types. France, 2011.