Add-Win Conflict-free Replicated Priority Queue

Yuqi Zhang, Yu Huang, Hengfeng Wei, Jian Lu
State Key Laboratory for Novel Software Technology
Nanjing University, Nanjing 210023, China
Email: cs.yqzhang@gmail.com, {yuhuang, hfwei, lj}@nju.edu.cn

1. Introduction

Internet-scale distributed systems often replicate data within and across data centers to provide low latency and high availability despite node and network failures. Replicas are required to accept updates without coordination with each other, and the updates are then propagated asynchronously. Moreover, to provide certain guarantee to developers of upper-layer applications, *Strong Eventual Convergence* (SEC) is widely accepted, which ensures that when any two replicas have received the same set of updates, they reach the same state [3]. This brings the issue of conflict resolution among concurrent updates, which is often challenging and error-prone. The Conflict-free Replicated Data Type (CRDT) framework provides a principled approach to address this challenge [2].

In this work we focus on the design of one specific CRDT: Conflict-free Replicated Priority Queue (CRPQ). CRPQs focus on storing order information in the distributed system, such as ranking of the score of players in a game, or the rating of movies/products in websites like IMDB/Amazon. There is an existing CRPQ design that choose Remove-Win strategy for conflict resolutions [4]. Here we present the design of Add-Win CRPQ. Our design is based on the algorithm of Add-Win Set (also known as OR-Set) [2].

2. Preliminaries

2.1. System Model

We consider a distributed system consisting of processes that can only fail by crash. Processes are interconnected by an asynchronous network. Messages may be delayed, dropped, but cannot be forged. The network ensures that eventually all messages are delivered successfully.

The CRPQ is designed to be replicated at a system of processes. Any replica can be modified without coordinating with any other replicas, and the updates are then propagated asynchronously. The design of CRPQ guarantees SEC, which is achieved by making each pair of possible concurrent operations commute [1].

The design of a CRPQ mainly focuses on the design of update operations. Following the CRDT framework, each update operation consists of two parts. In the **prepare** part, the immediate local processing on the replica, where the update operation is triggered, is specified. In the **effect** part, it is specified how the remote replica handles the update asynchronously propagated to it. Essentially, conflict resolution is conducted in this part to ensure that all replicas eventually converge to the same state when they receive the same set of update operations.

2.2. CRPQ Operations

The CRPQ is a container of elements of the form $e = (id, priority)$. The id of an element is unique, which can be used to lookup the element in the CRPQ. Priorities of elements are totally ordered, which decides the order of dequeuing of elements.

The process can modify (the replica) the CRPQ by the following update operations:

- $add(e, x)$: enqueue an element e with initial priority x.
- $rmv(e)$: remove the element e.
- $inc(e, i)$: increase the priority of element e by i.

Additionally, we assume that the CRPQ supports the query operations below to better illustrate our CRPQ design:

- $empty()$: returns $true$ if the CRPQ is empty.
- $lookup(e)$: returns $true$ if the element e is in the CRPQ.
- $get\_pri(e)$: returns the priority value of element e.
- $read\_max()$: return the id and the $priority$ value of the element with the highest priority.

1. The CRDT framework includes the operation-based and the state-based approaches. In this work, we adopt the operation-based approach.

3. Add-Win CRPQ

3.1. Design Rationale

There are two critical issues when designing the commutative semantics. The existence of an element. The priority of an element. The existence can be handled by Conflict-free Replicated Set, here we choose Add-Win Set. The essential challenge is how to design the semantics concerning the priority.

The priority value of an element e is influenced by all the three types of operations: $add(e, x)$, $inc(e, i)$ and $rmv(e)$. *add* and *rmv* operations have the semantics of priority value assignment (the *rmv* operation is regarded as writing a special *null* value to the priority of the element), and *inc* operations increment existing priority values. The conflict between two operations o_1 and o_2 can be divided to three types:

- AA-conflict [2]: Both o_1 and o_2 are *add* or *rmv* operations.
- II-conflict [3]: Both o_1 and o_2 are *inc* operations.
- AI-conflict: One of o_1 and o_2 is an *inc* operation. The other one is an *add* or *rmv* operation.

We need to explore the design of Add-Win Set to resolve all these types of conflicts simultaneously. An Add-Win set stores the history of *add*. When a *rmv* conflicts with *add*, the *rmv* is restricted to the *add* it observes. We inherit this rationale. We record the detailed history of *inc* (in addition to the detailed history of *add*). When conflicts occur (WW, II, WI), updates of the priority is restricted to the observed history.

We will discuss the basic rationale of the Add-Win strategy centering on the *existence* and the *priority* of an element.

3.1.1. Add-Win set. Conflict resolution concerning the existence of an element is inspired by the conflict-free replicated set named Add-Win Set (also known as Observed-Remove Set, or OR-Set).

Each *add* operation is attached a unique tag, without exposing the unique tags in the interface. The effect of *rmv* operations are restricted to *add* operations it observes.

When $add(e)$ is concurrent with $rmv(e)$, the *add* takes precedence, as the unique tag generated by *add* cannot be observed by *rmv*. The *add* has the precedence and the *rmv* has no effect. It is not difficult to verify that two adds and two rmvs commute.

To enable local processing of *add* and *rmv* operations, two grow-only sets A and R are used to store added and removed elements respectively. Local updates are propagated asynchronously to remote replicas. These two sets are grow-only, local processing is enabled. The existence of an element is judged by the difference of A and R.

2. 'A' stands for assignment.
3. 'I' stands for increase.

3.1.2. From set to priority queue. The rationale of Add-Win set is inherited by the Add-Win CRPQ. The challenge is how to handle the priority values of elements, enabling local processing and eventual consistency at the same time.

To handle the conflicts in priority updates, we need to explore how the priority values are obtained. The key observation is that, the priority value is first obtained by some *add* operation, using value assignment semantics. The values varies by *inc* operations, using increase/decrease semantics.

Two different source of priority values inspire us to differentiate the processing of priority values. The priority value is divided into two parts and stored separately:

- *innate* values: The innate value is initiated by *add* operations. Thus, the add-win set mechanism is utilized to handle the conflicts in innate value updates.
- *acquired* values: the increment part that brought *inc* operations. Since the *inc* operations naturally commute, such conflict resolution is trivial.

The *rmv* operation has value assignment semantics. Conflicts concerning *rmv* operations are resolved principally following the design of the Add-Win Set.

3.1.3. Key structures used. Generally an Add-Win CRPQ is a set S, in which each element is $e = (id, A, R)$, where the *id* is the unique tag identifying the element.

A is the add set. The element in A is $a = (t, x, inc, count)$, where t is the timestamp that records the $add(e)$ generates it, x is the innate value brought by the *add*, $inc = \sum_{o \in I} o.value$, suppose I is the set of $inc(e, i)$ that have observed t, and $count = \sum_{o \in I} |o.value|$.

R is the remove set, in which the elements are t, the timestamps of $a \in A$ that have been observed by some $rmv(e)$.

TABLE 1.

id	the unique tag which identifies the element
A	the add set $a = (t, x, inc, count)$
t	the timestamp
x	the innate value
inc	values
$count$	the times of updates
R	set of t

3.2. Handling the Existence of Elements

3.2.1. Inserting an element. Now we look into the design details of the Add-Win CRPQ. Like Add-Win Set, the Add-Win CRPQ consists with two sets: an add set A and a remove set R. The A stores insertion information and priority value information, and the R stores tombstone information. Now we look into the details of these information.

The insertion information is similar to the records in add sets of Add-Win Sets. In an Add-Win CRPQ, each time an *add* is executed, a timestamp t is generated by function $now()$ and is attached to the operation.

Since the precondition of *add* requires that the element does not exist, we do not need to detect concurrency, for two coexisting insertion tags are concurrent. Therefore we just need the timestamp to be unique and totally ordered. The design of such a time service is orthogonal to our CRPQ design. For example, you may use Lamport's clock [1], or simply the process id, together with an integer that increases every time a new timestamp is generated.

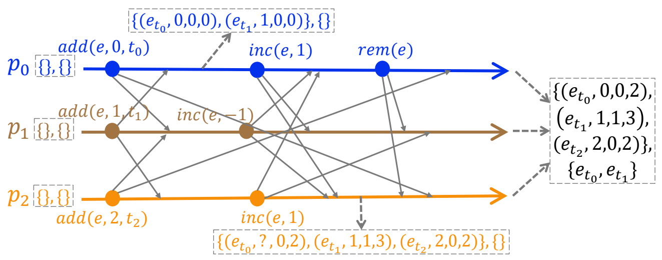
Here we use the timestamps as unique tags to identify each *add*. The *inc* and *rmv* operations will observe the timestamps. This is different from add-win set which can use any kind of unique tags. The difference is caused by the fact that we need to handle priority values, not simply the existence. We need the timestamp to determine the innate value of the element when concurrent insertion occurs.

3.2.2. Removing an element. The Add-Win CRPQ holds the record of every *add*, using timestamps as unique tags to make them distinct. *rmv* operations observe the current insertion record first and remove the insertion records it has observed.

We implement the *rmv* operation by tombstones. When an insertion record is removed, it is only a logical removal. This means that rather than removing the element from the CRPQ, its tombstone is added into the CRPQ to show that this record is logically no longer in the CRPQ.

Garbage collection of meta data used is an optimization. We omit the detailed discussions here.

We show the example in Fig. 1. The three processes p_0, p_1 and p_2 concurrently add element e, which are differentiated by timestamps t_0, t_1 and t_2. When process p_0 initiated $rmv(e)$, $add(e, 2, t_2)$ had not arrived yet. And the $rmv(e)$ only removes the insertion records it locally observed, which are e_{t_0} and e_{t_1}. Finally the record e_{t_2} remains active for all the three processes, and e exists. They are consistent at the aspect of the existence of element e.



$\{(e_{t_0},0,0,0),(e_{t_1},1,0,0)\},\{\}$
$add(e,0,t_0)$
$inc(e,1)$
$rem(e)$
p_0 {},{}
$add(e,1,t_1)$
$inc(e,-1)$
p_1 {},{}
p_2 {},{}
$add(e,2,t_2)$
$inc(e,1)$
$\{(e_{t_0},?,0,2),(e_{t_1},1,1,3),(e_{t_2},2,0,2)\},\{\}$
$\{(e_{t_0},0,0,2),(e_{t_1},1,1,3),(e_{t_2},2,0,2)\},\{e_{t_0},e_{t_1}\}$

Figure 1. An example for Add-Win CRPQ

The design is presented in Algorithm 1.

Note that auxiliary functions are required to implement the core *add*, *inc* and *rmv* operations.

Notice that $(e, t, ...)$ in A set means the insertion records of e, which are added by *add* operations, and (e, t) in R set means the tombstone of e, which are added by *rmv* operations. Element e exists if there is an insertion record that is not tagged as tombstone, which means $\exists t : (e, t, ...) \in A \wedge (e, t) \notin R$.

3.3. Handling the Priority of Elements

3.3.1. Updating the priority value. We have discussed before that the priority value of an element consists of two parts: innate value and acquired value. We now show how they are stored in our Add-Win CRPQ and how they are influenced by CRPQ operations.

The innate values are stored in the insertion records in A set, because they are also generated by *add* operations. The innate value of an element e is decided from all its active insertion records (records that do not have tombstones). Now that the insertion records are attached with total ordered timestamps, it is easy to come up with the solution that we can adopt the Last-Write-Win principle and choose the innate value with the biggest timestamp as the innate value of element e.

3.3.2. Interpreting the priority value. As for the acquired values, they are generated by *inc* operations. We here make *inc* operations take effect base on the observation of local insertion records, which is the same as *rmv* operations. Therefore the acquired values are also stored in the insertion records.

The updates of priority value guarantees commutativity. An important subsequent issue is how to interpret the priority values, stored in different elements in A and R.

Again we need to choose one as the true acquired value of the element. There might be many ways and reasons to solve the problem. Here we choose the perspective of preserving the user's intention. When a user calls an *inc* operation, we think that he intends to add an increment to the value, and his intention is embodied in the absolute value of the increment, in other words, the change value. We want to preserve users' intention as much as we can, so we now choose the acquired value who has accumulated the most change value. We call this Most-Change-Win.

If you choose the same strategy for choosing the innate value and the acquired value, you can store only one value as the final priority value of the element in one insertion record.

The interpretation scheme is orthogonal. Our design above is exemplar. Different schemes can be integrated.

We use the example in figure 1 to show how to obtain a consistent view from insertion records of the same element. Look at the position we point out at p_2. There are three active insertion records for element e. For the innate value of e we choose the record with the biggest timestamp, which is $(e_{t_2}, 2, 0, 2)$, suppose $t_0 < t_1 < t_2$. So the innate value of e is 2. For the acquired value we

Algorithm 1: Add-Win CRPQ

```
1  payload $A$: set of $(e, t, x, inc, count)$ tuples, $R$:
     set of $(e, t)$ tuples
2  initial $A = \emptyset, R = \emptyset$
3  query $empty()$: boolean
4      return
         $\forall t : (e, t, x, inc, count) \in A \rightarrow (e, t) \in R$
5  query $lookup(e)$: boolean
6      return
         $\exists t : (e, t, x, inc, count) \in A \wedge (e, t) \notin R$
7  query $priority(e)$: integer
8      pre $lookup(e)$
9      let $x : (e, t, x, inc, count) \in A \wedge (e, t) \notin$
         $R \wedge \forall t' : ((e, t', x', inc', count') \in$
         $A \wedge (e, t') \notin R) \rightarrow t' \leq t$
10     let $inc : (e, t, x, inc, count) \in A \wedge (e, t) \notin$
         $R \wedge \forall count' : ((e, t', x', inc', count') \in$
         $A \wedge (e, t') \notin R) \rightarrow count' \leq count$
11     return $x + inc$
12 query $read\_max()$: id, integer
13     pre $\neg empty()$
14     let $e : lookup(e) \wedge \forall o : lookup(o) \rightarrow$
         $priority(o) \leq priority(e)$
15     return $e, priority(e)$
16 update $add(e, x)$
17     prepare $(e, x)$
18         pre $\neg lookup(e)$
19         let $t = now()$
20     effect $(e, t, x)$
21         if $\nexists (e, t, null, inc, count) \in A$ then
             $A := A \cup \{(e, t, x, 0, 0)\}$
22         else
23             let
                 $inc, count : (e, t, null, inc, count) \in A$
24             $A := A \setminus \{(e, t, null, inc, count)\} \cup$
                 $\{(e, t, x, inc, count)\}$
25         end
26 update $inc(e, i)$        ▷ $i \in \mathbb{R}$, $i < 0$ means decrease
27     prepare $(e, i)$
28         pre $lookup(e)$
29         let $O = \{t | (e, t, x, inc, count) \in$
             $A \wedge (e, t) \notin R\}$
30     effect $(e, i, O)$
31         foreach $t \in O$ do
32             if $\nexists (e, t, x, inc, count) \in A$ then
                 $A := A \cup \{(e, t, null, 0, 0)\}$
33             let
                 $x, inc, count : (e, t, x, inc, count) \in A$
34             $A := A \setminus \{(e, t, x, inc, count)\} \cup$
                 $\{(e, t, x, inc + i, count + |i|)\}$
35         end
36 update $rmv(e)$
37     prepare $(e)$
38         pre $lookup(e)$
39         let $O = \{t | (e, t, x, inc, count) \in$
             $A \wedge (e, t) \notin R\}$
40     effect $(e, O)$
41         $R := R \cup (\{e\} \times O)$
```

choose the record that has accumulated the most change value, which is $(e_{t_1}, 1, 1, 3)$, whose accumulated change value is 3. So the acquired value of e is 1. Therefore the priority value of element e is $2 + 1 = 3$.

3.4. Proof of Commutativity Between Operations

We conclude the design by a thorough examination of the commutativity between an operation pair in all cases.

In conclusion, the Add-Win CRPQ stores all the *add* operations who are attached with unique and total ordered timestamps. An element exists in the CRPQ means that it has living insertion records, which means they do not have tombstones. From these living insertion records we choose the innate value by Last-Write-win principle and the acquired value by Most-Change-win principle, and add them up as the value of the element.

The detailed design of the Add-Win CRPQ is shown in 1. We now prove the commutativity of update operations as follows.

- $add(e, x_1, t_1)$ and $add(e, x_2, t_2)$. Both of them will be recorded in set A, no matter who executes first.
- $inc(e, i_1, O_1)$ and $inc(e, i_2, O_2)$. In set A, as for the intersection part of O_1 and O_2, there is the commutativity of addition.
- $rmv(e, O_1)$ and $rmv(e, O_2)$. No matter who executes first, both of them will be recorded in set R as tombstones.
- $add(e, x, t)$ and $inc(e, i, O)$. Consider $t \in O$. The operation that execute first will add the insertion record into set A. The *add* operation will write the innate value, and the *inc* will take effect in the acquired value.
- $add(e, x, t)$ and $rmv(e, O)$. The former takes effect in set A, and the latter set R.
- $inc(e, i, O_1)$ and $rmv(e, O_2)$. The former takes effect in set A, and the latter set R.

References

[1] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[2] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Research Report RR-7506, Inria – Centre Paris-Rocquencourt ; INRIA, Jan. 2011.

[3] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS'11, pages 386–400, Berlin, Heidelberg, 2011. Springer-Verlag.

[4] Y. Zhang, Y. Huang, H. Wei, and J. Lu. Remove-win: a design framework for conflict-free replicated data collections. 2019.