Remove-Win: a Design Framework for Conflict-free Replicated Data Collections

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

ABSTRACT

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. 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.

This work focuses on a special type of CRDT, namely the Conflict-free Replicated Data Collection (CRDC), e.g. list and queue. The CRDC can have complex and compound data items, which are organized in structures of rich semantics. Complex CRDCs can greatly ease the development of upper-layer applications, but also makes the conflict resolution notoriously difficult. This explains why existing CRDC designs are tricky, and hard to be generalized to other data types. A design framework is in great need to guide the systematic design of new CRDCs.

To address the challenges above, we propose the Remove-Win Design Framework. The remove-win strategy for conflict resolution is simple but powerful. The remove operation just wipes out the data item, no matter how complex the value is. The user of the CRDC only needs to specify conflict resolution for non-remove operations. This resolution is destructed to three basic cases and are left as open terms in the CRDC design skeleton. Stubs containing user-specified conflict resolution logics are plugged into the skeleton to obtain concrete CRDC designs. We demonstrate the effectiveness of our design framework via a case study of designing a conflict-free replicated priority queue. Performance measurements also show the efficiency of the design derived from our design framework.

PVLDB Reference Format:
Yuqi Zhang, Yu Huang, Hengfeng Wei, Jian Lu. Remove-Win: a Design Framework for Conflict-free Replicated Data Collections. *PVLDB*, 12(xxx): xxxx-yyyy, 2019.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx

Proceedings of the VLDB Endowment, Vol. 12, No. xxx
ISSN 2150-8097.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx

1. INTRODUCTION

Internet-scale distributed systems often replicate application state and logic within and across data centers, to reduce user-perceived latency and improve application throughput, while tolerating partial failures without compromising overall service availability [17, 20]. In such distributed systems, user-perceived latency and overall service availability are widely regarded as the most critical factors for a large class of applications. For instance, the experiments from Google demonstrate that increasing web search latency 100 to 400ms reduces the daily number of searches per user by 0.2% to 0.6% [10]. Thus, many Internet-scale distributed systems are designed for low latency and high availability in the first place.

To provide low latency and high availability, the update requests must be handled immediately, without communicating with remote replicas. Updates to the replica can only be asynchronously transmitted to remote replicas, and rolling-back updates to handle conflicts is not acceptable. According to the CAP theorem, the low latency and high availability can only be achieved at the cost of accepting weak consistency [9]. 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 [19]. Eventually consistent replicated data types are widely used in scenarios where responsiveness is critical, e.g. in collaborative editing [22] or distributed caching [11]. The design of replicated data types satisfying SEC brings the challenge of conflict resolution for concurrent updates on different replicas of logically the same data. The conflict resolution is especially hard and error-prone when the replicated data type is complex and has rich semantics. The Conflict-free Replicated Data Type (CRDT) framework provides a principled approach to address this challenge [18, 17].

In this work, we mainly focus on a special type of CRDT, namely the Conflict-free Replicated Data Collections (CRDCs). The CRDC is a collection of data items, which have user-specified values and are organized in certain structure. Different types of data collections, e.g. sets, lists, queues and graphs, are widely used in nearly all applications and greatly ease the development of upper-layer applications. However, complex data collection types also make the conflict resolution for concurrent updates notoriously difficult and error-prone. This explains why existing CRDC designs are tricky, and are hard to be generalized to the design of other CRDCs. A design framework is in great need to guide the systematic design of new CRDCs, and the design of CRDCs needs to shift from a craft to an engineering discipline.

To address the challenges above, we propose the Remove-Win Design Framework. The conflict resolution of concurrent updates is decomposed into two issues: handling the existence of the element and handling the value of the element, and are addressed by the Remove-Win Set (RWSet) and the Remove-Win Skeleton (RWSkeleton) respectively:

- Concerning the existence of elements, the remove-win strategy for conflict resolution is simple but powerful. The remove operation just wipes out the data item, no matter how complex the value is. The conflict resolution involving remove operations is implemented as the RWSet.

- Concerning the value of elements, the RWSet is then augmented into the RWSkeleton, where the user can add/remove elements and initialize/update their values. The CRDC user only needs to specify conflict resolution for non-remove operations, i.e. initializing a value and updating the value. This resolution is destructed to three basic cases and is implemented as three open terms in the RWSkeleton. Stubs implementing user-specified logic for conflict resolution among non-remove operations can be plugged into the skeleton to obtain a full CRDC design.

We demonstrate the effectiveness of our Remove-Win Design Framework via a case study of designing a conflict-free replicated priority queue. Performance measurements show the efficiency of the replicated priority queue design derived from our design framework.

The rest of this work is organized as follows. Section 2 overviews the Remove-Win Design Framework. Section 3 and 4 present design of the RWSet and the RWSkeleton respectively. Section 5 presents the design of a CRPQ based on the design framework, and Section 6 presents the performance measurements. Section 7 reviews the existing work. Finally, Section 8 concludes this work and discusses the future work.

2. REMOVE-WIN DESIGN FRAMEWORK

We first overview the basics of CRDC. Then we present the design framework.

2.1 Conflict-free Replicated Data Collections

A CRDC is an abstract data collection type, whose design follows the CRDT framework [18, 17]. Examples of CRDCs include sets, lists and queues. One CRDC design has its payload, which stores information used to implement the CRDC. A CRDC has well-defined interfaces, and the APIs can be divided into two types: *update* and *query*. The process can modify the state of the replica by update operations, while it can also obtain the state of the replica by query operations, without any side effect.

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

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

The design of a CRDC 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. See Algorithm 1 for an example of the algorithms designed following the CRDT framework.

2.2 Remove-Win Conflict Resolution

The conflict resolution for concurrent updates on a CRDC needs to consider both the existence of elements in the data collection and the value of elements. These two issues are entangled with each other and complicates the resolution of conflicts. However, the remove-win strategy we employ is simple but powerful in that it can decouple the existence and the value of elements and simplify the conflict resolution. Essentials of the remove-win design framework are shown in Fig. 1.

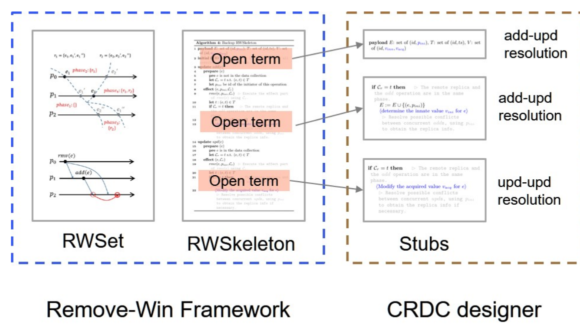

Figure 1: Remove-Win Design Framework.

As for the existence of one element, the remove-win strategy is simple. The remove operation wipes out the effect of all operations which are before or concurrent with it, no matter how the data item was initiated and modified. Thus, the execution is segmented into *phases* by remove operations. Non-remove operations create the data item and update its value, which constitute the phase. Remove operations wipe out everything, which ends the current phase and starts the new phase from scratch. The conflict resolution involving remove operation is implemented in the RWSet, as detailed in Section 3.

As for the value of one element, the RWSet is augmented to the RWSkeleton (see Section 4 for the detailed design), which enables the user to set the initial value of the data item and update its value. The conflict resolution concerning the value of elements needs user intervention. It should be conducted within each phase of execution, involving add operations that create a data item with its initial value and value-updating operations that modify value of the data item. The conflict resolution is decomposed to three basic cases, which are left as open terms in the RWSkeleton. Users can specify their logics for all cases of conflict resolution, and implement them as stubs. The stubs are plugged into the skeleton, which yields the full design of a CRDC.

3. RWSET - HANDLING THE EXISTENCE OF ELEMENTS

We first explain the basic rationale of the remove-win strategy and present a straightforward design of RWSet. Then we present optimizations and derive the final design.

3.1 Rationale of the Remove-Win Strategy

Suppose there are n processes/replicas [2] $p_0, p_1, \cdots, p_{n-1}$, each holding one replica of a CRDC. Processes are interconnected by an asynchronous network, and can only fail by crash. Messages may be delayed but cannot be forged. The communication network ensures that eventually all messages are delivered successfully.

3.1.1 Temporal Order among Events and Operations

One update operation o initiated on p_i consists of one local event $o.e_{lcl}$ on p_i, and n remote events, one remote event $o.e_{rmt}$ for each replica (including p_i itself) [3]. We define function $\text{PROC}(o)$, which maps operation o to its initiating replica p_i, and function $\text{TYPE}(o)$, which maps operation o to its type (e.g, *add*, *rmv* or *upd*).

The temporal order among local and remote events are essential to the design of remove-win CRDCs:

Definition 3.1: *order between events.* There are two basic types of order between events:

- *Program order.* Events on the same replica are totally ordered by the program order, denoted by $\xrightarrow{po}$.
- *Local-remote order.* The local event $o.e_{lcl}$ and each remote event $o.e_{rmt}$ belonging to the same operation o has the local-remote order, denoted by $\xrightarrow{lm}$.

The happen-before relation between events, denoted by $\rightarrow$, is defined as the transitive closure of the program order and the local-remote order. □

Given the order between events, we can further define the *visibility relation* between operations:

Definition 3.2: *visibility between operations.* Operation o_1 is visible to o_2, denoted by $o_1 \xrightarrow{vis} o_2$, if:

- o_1 and o_2 are initiated by the same replica and $o_1.e_{lcl} \xrightarrow{po} o_2.e_{lcl}$, or
- o_1 and o_2 are initiated by the different processes, and on the replica which initiates o_2, we have $o_1.e_{rmt} \xrightarrow{po} o_2.e_{lcl}$.

Two update operations o_1 and o_2 are concurrent, denoted by $o_1 \parallel o_2$, if neither $o_1 \xrightarrow{vis} o_2$ nor $o_2 \xrightarrow{vis} o_1$ holds. □

Note that the $\xrightarrow{vis}$ relation is not transitive.

The importance of the $\xrightarrow{vis}$ relation is obvious. The remove-win strategy is interpreted with the $\xrightarrow{vis}$ relation as: non-remove operations which are visible to or are concurrent with a remove operation is wiped out by this remove operation. Given the remove-win strategy, the execution is segmented into phases. Within a phase, non-remove operations initialize a data item and update its value. The remove operation ends the current phase and starts a new phase from scratch. Phase-based resolution is central to the design of RWSet, as detailed below.

[2] We use the terms 'process' and 'replica' interchangeably when no confusion is caused.

[3] For the ease of presentation, the remote event on the initiating process is often omitted.

3.1.2 Segmenting System Execution into Phases

The design below considers one single data item. For each data item in the CRDC, the conflict resolution is conducted independently. Consider concurrent non-remove operations o_1 and o_2. They belong to different phases if there is a remove operation r that "separates" them, as shown in Fig. 2. Here, "separates" means that $\neg(r \xrightarrow{vis} o_1)$ and $r \xrightarrow{vis} o_2$.

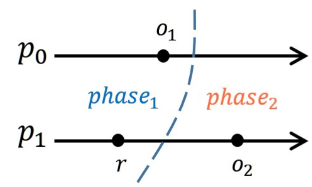

Figure 2: Basic idea of phase.

The remove operation wipes out effects of all operations which are visible to it or are concurrent with it. The current phase ends. The new phase starts when an add operation initiates the data item again, and value-update operations modify the data value. To define the concept of phase, we first define the remove history of an operation/replica:

Definition 3.3: *remove history.* The remove history $\mathcal{H}_r(o)$ of an operation o is the set of all remove operations that are visible to it:

$$\mathcal{H}_r(o) = \{op \mid \text{TYPE}(op) = rmv,\ op \xrightarrow{vis} o\}$$

The remove history of one replica is defined as the union of remove histories of all operations on this replica. Here, we say the operation o is on replica p_i if $o.elcl$ or any of $o.e_{rmt}$ takes place on p_i. □

Note that we define $\mathcal{H}_r(o)$ for both non-remove and remove operations.

With the definition of remove history, we can formally define *phase*:

Definition 3.4: *phase.* Two operations belong to the same phase, if they have the same remove history. Or equivalently, the phases of system execution are the equivalence classes in $O/\approx_{\mathcal{H}_r}$, where O is the set of operations, and $\approx_{\mathcal{H}_r}$ is the equivalence relation defined by $\mathcal{H}_r$: $a \approx_{\mathcal{H}_r} b \triangleq \mathcal{H}_r(a) = \mathcal{H}_r(b)$. We denote the phase that operation a belongs to as $[a]$. □

Since the replica also has its remove history, according to the definition above, we can also say that one replica and one operation are in the same phase when they have the same remove history.

Phases are temporally ordered. We say $[a] \prec [b]$ if $\mathcal{H}_r(a) \subset \mathcal{H}_r(b)$. Fig. 3 give a more complex example of operations belonging to different phases. Assume that remove operation r_1 (r_2) consists of its local event e_1 (e_2) and its remote events e'_1 (e'_2) and e''_1 (e''_2). All non-remove operations (not drawn in the figure) in the left area belong to $phase_1 = \emptyset$, since no remove operations are visible to them. All operations in the right area belong to $phase_4 = \{r_1, r_2\}$. Obviously, $phase_1 \prec phase_4$.

Somewhat counter-intuitively, operations in the middle-upper area and those in the middle-lower area belong to different phases. This is because on p_2, e''_1 is greatly delayed until after e''_2. Operations in the middle-upper area can see r_1 but not r_2, while operations in the middle-lower area can see r_2 but not r_1. Thus, we have $phase_1 \prec phase_2$ and

$phase_1 \prec phase_3$, as well as $phase_2 \prec phase_4$ and $phase_3 \prec phase_4$. However, $phase_2$ and $phase_3$ are not temporally ordered.

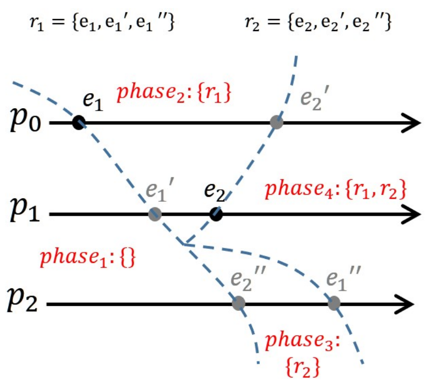

Figure 3: Partial order among phases.

3.2 Basic Design

Concerning the existence of elements, there will only be conflict between one add operation and one remove operation on the same data element. We resolve this conflict based on the phase and remove history of operations. When the value of element is concerned, the conflict resolution is detailed in Section 4.

3.2.1 CRDT Basics

Following the CRDT framework, each RWSet $\mathcal{S}$ is implemented over its payload, two sets E and T. Set E contains the *id* of data elements. Element $id \in E$ basically means that this element is in $\mathcal{S}$. Set T is the set of tuples (e, α), where tag α is the unique tag of one remove operation and T records the tags of all remove operations, i.e. the remove history, on element e.

When an add operation $add(e)$ is initiated on replica p_i, it first conducts the local processing, taking e as the user-specified parameter (the **prepare** part, Line 5 ~7 in Algorithm 1). Replica p_i checks whether e is already in $\mathcal{S}$ (Line 6). If not, the remove history of this add operation is obtained and recorded in $\mathcal{H}_r$ (Line 7).

After the local processing on the initiating replica p_i, p_i broadcasts this $add(e)$ operation and triggers the remote processing on all replicas (the **effect** part, Line 8 ~11 in Algorithm 1). This broadcast has two parameters, the user-specified parameter e and the parameter $\mathcal{H}_r$ prepared in the local processing.

3.2.2 Phase-based Conflict Resolution

The essential issue addressed in the design of the $add(e)$ operation is the conflict resolution in its **effect** part. The key to the conflict resolution is the remove history of *add* operations and remote replicas.

We first need to handle the anomaly caused by the fact that the remove operation can arrive at the remote replica arbitrarily late. This late remove operation can falsely remove data elements. For example in Fig. 4, when p_2 executes the remote event of $add(e)$, it will add element e into E. However, when the $rmv(e)$ is delayed and arrives after $add(e)$ on p_2, it will falsely remove data element e. This is because $rmv(e)$ is visible to $add(e)$ and the effect of these two operations should be "Element e is first removed but laterly added again, and e is now in the set".

We prevent this anomaly by assuming that the underlying communication system guarantees causal message delivery, i.e. the order of message delivery always respects the order of the corresponding message send [21, 8].

Given causal message delivery, the remote event of $add(e)$ on p_2 in Fig. 4 will be delayed until p_2 receives the remote event of $rmv(e)$ first. This is because p_1 has seen $rmv(e)$ before $add(e)$. Thus we know that the broadcast of $rmv(e)$ to all replicas is before that of $add(e)$. Causal message delivery ensure that the delivery on p_2 respects this order.

Note that the assumption of causal message delivery is mainly for the ease of presenting the basic rationale of our RWSet design. We will remove this assumption in the following Section 3.3.

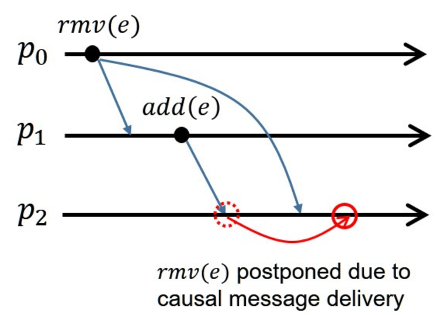

Figure 4: Necessity of causal message delivery.

Given causal message delivery, when a remote replica sees one *add* operation, it is guaranteed to see the *rmv* operations in the remove history of this *add* operation first. With this guarantee, we can now discuss the conflict resolution between concurrent *add* and *rmv* operations. Suppose operation $add(e)$ is initiated at replica p_i. Then the remote event of $add(e)$ arrives at a remote replica p_j.

Note that the remote event from p_i brings with it the remove history $\mathcal{H}_r$ of the $add(e)$ operation (Line 8 in Algorithm 1). The remove history on remote replica p_j is recorded in its local payload T. With the guarantee of causal delivery, we have $\mathcal{H}_r \subseteq T$. This is because operations in $\mathcal{H}_r$ are those remove operations visible to $add(e)$. The delivery of these remove operations on p_j must proceed the delivery of $add(e)$. Given this fact, we have two cases left to handle:

- $\mathcal{H}_r = T$. This means that $add(e)$ and p_j have seen the same set of remove operations. There will be no conflict and we directly add e into payload E on p_j.

- $\mathcal{H}_r \subset T$. This means that $\exists \alpha : (e, \alpha) \in T \wedge (e, \alpha) \notin \mathcal{H}_r$. Denote the remove operation has tag α as $rmv(e)$. We have that r_j has seen a $rmv(e)$ that $add(e)$ did not see. This $rmv(e)$ either is concurrent with $add(e)$ or happens after $add(e)$ (due to causal message delivery). According to the remove-win strategy, the effect of $add(e)$ will be wiped out by $rmv(e)$.

Thus only when we have $\mathcal{H}_r = T$ can we successfully add element e into the payload E. Otherwise, it is to be wiped our by some *rmv* operation and should be safely ignored.

The remove operation is relative simpler to implement. The initiating replica first checks whether this element is actually in $\mathcal{S}$, and then generates the unique tag α for this remove operation. The tag is propagated to all replicas (including p_i itself) and all replicas add α to its remove history. Then e is wiped out from E.

Algorithm 1: RWSet (basic design)

payload E: set of elements, T: set of (e, α) tuples
initial $E = \emptyset, T = \emptyset$
update *add*(e)
  prepare (e)
    pre $e \notin E$
    let $\mathcal{H}_r = \{(e, \alpha) \mid (e, \alpha) \in T\}$
  effect $(e, \mathcal{H}_r)$
    pre $\mathcal{H}_r \subseteq T$ ▷ Casual delivery suffices.
    if $\mathcal{H}_r = T$) **then**
      $E := E \cup \{e\}$
update *rmv*(e)
  prepare (e)
    pre *lookup*(e)
    let α be the unique tag of this operation
  effect (e, α)
    $T := T \cup \{(e, \alpha)\}$
    $E := E \setminus \{e\}$

3.3 Optimizations

The remove-win strategy is centered on the remove history, how it is maintained with the help from causal message delivery and how it is transmitted among replicas. The design above mainly illustrates the basic rationale, and are not necessarily efficient. In this section, we present two optimizations, namely eliminating the use of causal message delivery and encoding of the remove history.

3.3.1 Eliminating causal message delivery

The remove history of an *add* operation only contains all *rmv* operations which are visible to it. The remove history does not contain all *rmv* operations whose have causally affected the *add* operation. Thus, causal message delivery is necessary to prevent the lately arrived but causally affecting *rmv* operations.

Or equivalently, we need causal message delivery to ensure that, when the remove history of an operation/replica contains remove operation r, it also contains r' which is visible to r, contains r'' which is visible to r', and so on.

The requirement of causal message delivery is mainly because the $\xrightarrow{vis}$ relation we define is quite basic. It is the weakest one in a family of possible visibility relations, as discussed in [13]. Further exploring the family of possible visibility relations, we can strengthen the $\xrightarrow{vis}$ relation to the causal visibility relation $\xrightarrow{cvs}$. Then the causal message delivery can be eliminated.

Definition 3.5: *causal visibility.* Operation o_1 is causally visible to o_2, denoted by $o_1 \xrightarrow{cvs} o_2$, iff.

$$o_1 \xrightarrow{vis} o_2,\ or\ \exists o : o_1 \xrightarrow{cvs} o \land o \xrightarrow{cvs} o_2$$

Equivalently, the $\xrightarrow{cvs}$ relation is the transitive closure of the $\xrightarrow{vis}$ relation. □

Given the definition of $\xrightarrow{cvs}$, the definition of remove history is also "upgraded" to causal remove history:

Definition 3.6: *causal remove history.* The causal remove history of an operation o, denoted by $\mathcal{C}_r(o)$ is:

$$\mathcal{C}_r(o) = \{op \mid \text{TYPE}(op) = rmv \land op \xrightarrow{cvs} o\}$$

The causal remove history of one replica is defined as the union of causal remove histories of all operations on this replica. □

Again, we look at the example in Fig. 4. The *add*(e) brings with it its causal remove history. When p_2 sees *add*(e), it first finds that one remove operation (the *rmv*(e) operation) in the causal remove history is missing and executes the missing *rmv* first. Then data element e is added into the set. When *rmv*(e) arrives at p_2 later, the causal remove history of p_2 already has *rmv*(e) in it. So p_2 will (safely) ignore *rmv*(e).

3.3.2 Encoding of causal remove history

Given the definition of causal visibility and causal remove history, we further discuss how each operation/replica correctly and efficiently maintains its causal remove history.

The remove operation has the salient feature that it does not require any parameters (except for e identifying the element of concern), it is idempotent and its effect is always the same (wiping out everything) no matter how status of the data has evolved.

Thus we do not care how many times the remove operations have taken place. If the k^{th} *rmv* operation that is initiated by p_i is causally visible, all remove operation from the 1^{st} to the $(k-1)^{th}$ initiated by p_i are causally visible as well. However, since the remove operation is idempotent, we only need to record the last remove operation initiated on p_i.

This means that, we do not need to transmit the real causal remove history. We only need to transmit certain "encoding" of the causal remove history which records the latest *rmv* operation causally visible on each replica. The encoding/decoding schemes we use is principally the vector clock mechanism [14]. The record of remove operations initiated on an element e can be encoded as a vector $t[1..n]$. All remove operations initiated on replica p_i are totally ordered, and we use the index k to uniquely identify each remove operation. When we have $t[j] = k$ on replica p_i , it means that the latest remove operation that is initiated by replica p_j which has been causally visible to p_i is the k^{th} remove operation p_j initiates.

With the definition of the causal remove history vector (abbreviated as *crh-vector*), we now show how this vector is updated. When replica p_i receives an operation o carrying a vector $t[1..n]$ which encodes $\mathcal{C}_r(o)$, its local vector $t'[1..n]$, which encodes all the remove operations it has causally seen, needs to be updated as $\forall k : t'[k] = max(t'[k], t[k])$.

3.4 Optimized design

The optimized design is principally the same as the basic design. As in the basic design, the optimized design also focuses on one data element e, and different data elements are handled independently. The main difference is that the remove history is upgraded to the causal remove history and only the encoding of the causal remove history, i.e. the crh-vector, is maintained and transmitted. Also, the causal message delivery is no longer necessary.

The set E in the payload is the same with that in the basic design. Element e exists if its identifier is in set E.

As for T, the tag of the remove operation is replaced by the causal remove history vector, i.e. T is the set of tuples (e, t) and t is the crh-vector.

We first discuss the remove operations $rmv(e)$. When replica p_{ini} initiates $rmv(e)$, it first locally increases the crh-vector $t[p_{ini}]$ (Line 15 in Algorithm 2). Then the causal remove history of this operation is prepared in $\mathcal{C}_r$ for the broadcast (Line 13 in Algorithm 2).

The user-specified parameter e and locally prepared parameter $\mathcal{C}_r$ are broadcast to remote replicas on behalf of the operation $rmv(e)$. If in any dimension k, the local crh-vector element $t[k]$ is older than the vector element $\mathcal{C}_r[k]$ from the broadcast, we remove e from E, since there are unseen remove operations due to the message delay (Line 18 in Algorithm 2). Then the local crh-vector $t[1..n]$ is updated to the pairwise maximum of $\mathcal{C}_r$ and t and this update is recorded in the payload T (Line 20˜21 in Algorithm 2).

As for the $add(e)$ operation, it first prepares its crh-vector $\mathcal{C}_r$ (Line 5 in Algorithm 2). Then the element e and the crh-vector $\mathcal{C}_r$ are broadcast to all remote replicas.

When the remote replica checks the crh-vector $\mathcal{C}_r$ and find that there are remove operations it has not seen (but encoded in the crh-vector from the broadcast), it will execute the missing remove operations first. This execution is the same with the **effect** part of the $rmv(e)$ operation (Line 7 in Algorithm 2). After this supplementing remove operation, we have that $\mathcal{C}_r \subseteq t$.

Then if $\mathcal{C}_r = t$, it means that local replica is currently in the same phase with this $add(e)$. Thus data element e is put into E. Otherwise ($\mathcal{C}_r \subset t$), the local replica has causally seen a $rmv(e)$ that this $add(e)$ did not. The $add(e)$ is discarded.

Algorithm 2: RWSet (optimized design)

1 **payload** E: set of elements, T: set of (e, t) tuples
2 **initial** $E = \emptyset, T = \emptyset$
3 **update** $add(e)$
4   **prepare** (e)
5   **pre** $e \notin E$
6   **let** $\mathcal{C}_r = t$ s.t. $(e, t) \in T$
7   **effect** $(e, \mathcal{C}_r)$
8         $rmv(e, \mathcal{C}_r)$ ▷ Execute the effect part of $rmv(e)$ with parameters $(e, \mathcal{C}_r)$.
9   **let** $t : (e, t) \in T$
10   **if** $\mathcal{C}_r = t$ **then** $E := E \cup \{e\}$
11 **update** $rmv(e)$
12   **prepare** (e)
13   **pre** $e \in E$
14   **let** $\mathcal{C}_r = t$ s.t. $(e, t) \in T$
15   **let** p_{ini} be id of the initiator of this operation
16         $\mathcal{C}_r[p_{ini}] := \mathcal{C}_r[p_{ini}] + 1$
17   **effect** $(e, \mathcal{C}_r)$
18   **let** $t : (e, t) \in T$
19   **if** $\exists k : t[k] < \mathcal{C}_r[k]$ **then**
20             $E := E \setminus \{e\}$
21   **let** $t' : \forall k : t'[k] := max(\mathcal{C}_r[k], t[k])$
22             $T := T \setminus \{(e, t)\} \cup \{(e, t')\}$

4. RWSKELETON - HANDLING THE VALUE OF ELEMENTS

The RWSet can be augmented to store user-specified application-specific values. Since the conflict concerning the existence of elements is handled by the RWSet, the user can focus on the conflicts concerning the value of elements. The conflict resolution concerning values can be destructed into three basic cases. Thus the RWSkeleton is proposed, where three open terms are left for the user to develop stubs containing their own conflict resolution logics. With the RWSkeleton, the concrete design of a CRDC can be obtained by specifying how the values are initialized and updated via the CRDC APIs and plugging the conflict-resolving stubs.

In this section, we first briefly overview conflict resolution involving remove operations. Then we focus on the three basic cases of conflict resolution among non-remove operations. An exemplar design of a conflict-free replicated priority queue will be presented in the following Section 5

4.1 Remove-win Resolution

The RWSkeleton has the new value-updating operation *upd*, which enables the user to modify the values of existing data elements. Comparing with the RWSet, the *add* operation in the RWSkeleton not only creates a data element in the CRDC, but also sets its initial value.

However, owing to the remove-win strategy, the conflict resolution between remove and non-remove operations (*add* and *upd*) are principally the same. The rmv operations win, and the effect of (concurrent or causally visible) non-remove operations is wiped out.

The execution is still segmented into phases by the rmv operations. When executed on a remote replica, each non-remove operation carries the crh-vector, uses the vector to firstly execute the missing rmv operations at the **effect** part of this operation and then takes effect only if this operation is in the same phase with the replica.

4.2 User-specified Resolution

With the help from the RWSet, the user only needs to care about the conflicts concerning data values among non-remove operations within each phase. Two types of non-remove operations, *add* and *upd*, may modify the value and potentially cause conflicts. Thus, there are three different types of possible conflicts to be considered, as detailed one by one below.

4.2.1 Add-add resolution

When two different *add* operations both add the same element, but setting different initial values, there will be conflict. An open term is left in the skeleton (Line 13 in Algorithm 3) to let the user specify how to handle this conflict. Principally, the user must use certain information of the initiating replicas, in order to differentiate concurrent *add* operations. Thus, the payload E not only contains the element *id*, but also contains p_{ini}, the id of the initiating replica. The p_{ini} can be thought as a handler, with which the *add* operation can access any information of the replica necessary to differentiate concurrent *add* operations. For example, the user may specify "larger replica id wins", assuming that the replica ids are totally ordered. Thus the initial value of element is set to the value from the *add* operation initiated by the replica with larger *id*.

4.2.2 Upd-upd resolution

The value of elements may be modified by application-specific *upd* operations. Conflict between *upd* operations is to be resolved by user-specified resolution logic (Line 35 in Algorithm 3).

For example, for a list, the user may employ an *operational transformation* algorithm to decide the results of all possible conflicting list updates (*insert* and *delete*) [7, 22]. As for a priority queue, the value increase/decrease operations naturally commute. Thus no resolution is needed, as detailed in the following Section 5.

4.2.3 Add-upd resolution

Though the *add* operation and the *upd* operation both can modify the value of data items, they have different types of user intention behind them. Specifically, the *add* operation initializes the value. It has semantics similar to those of value assignments. The *upd* operation modifies value. The semantics is application-specific, and usually are different from those of value assignments. For example, priority values of elements in a priority queue are often modified by increase or decrease of the (numerical) priority values.

According to the two (often) different types of user intention, we divide the value of an element into the *innate value* and the *acquired value* (payload V in Line 1 in Algorithm 3). Accordingly, the *add* operation only modifies the innate value, while the *upd* operation only modifies the acquired value. Thus, the conflict between an *add* and a *upd* operation is resolved by dividing the data value to two parts, one part for each operation.

Note that dividing the element value raises the problem of how to interpret the data value for upper-layer applications. Often, we can let the data value be the sum of innate and acquired values (when they can be added together), but there could be any user-specified interpretation here.

5. BUILDING A CRPQ

We design and implement a Conflict-free Replicated Priority Queue (CRPQ), under the guidance of the Remove-Win Framework. The CRPQ is a container of elements of the form $e = (id, priority)$. Each element is identified by its id, and without loss of generality, we assume that the priority value is an integer. The client can modify (the replica of) the CRPQ by the following update operations:

- $add(e, x)$: enqueue element e with initial priority x.
- $rmv(e)$: remove the element e.
- $inc(e, \delta)$: increase the priority of element e by δ (δ may be negative).

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 e is in the CRPQ.
- $get\_pri(e)$: returns the priority value of e.
- $get\_max()$: returns the *id* and *priority* of the element with the highest priority.

Following the RWFramework, design of the CRPQ is obtained by instantiating the RWSkeleton and develop CRPQ-specific stubs, as detailed below.

Algorithm 3: RWSkeleton

1 **payload** E: set of (e, p_{ini}) tuples, T: set of (e, t) tuples, V: set of (id, v_{inn}, v_{acq}) tuples
2 **initial** $E = \emptyset, T = \emptyset, V = \emptyset$
3 **update** $add(e)$
4   **prepare** (e)
5   **pre** e is not in the data collection
6   **let** $\mathcal{C}_r = t$ s.t. $(e, t) \in T$
7   **let** p_{ini} be id of the initiator of this operation
8   **effect** $(e, p_{ini}, \mathcal{C}_r)$
9         $rmv(e, p_{ini}, \mathcal{C}_r)$ ▷ Execute the effect part of $rmv(e)$ using $\mathcal{C}_r$.
10   **let** $t : (e, t) \in T$
11   **if** $\mathcal{C}_r = t$ **then** ▷ The remote replica and the add operation are in the same phase.
12             $E := E \cup \{(e, p_{ini})\}$
13             $\langle$determine the innate value v_{ini} for $e\rangle$ ▷ Resolve possible conflicts between concurrent adds, using p_{ini} to obtain the replica info.
14 **update** $rmv(e)$
15   **prepare** (e)
16   **pre** e is in the collection
17   **let** $\mathcal{C}_r = t$ s.t. $(e, t) \in T$
18   **let** p_{ini} be id of the initiator of this operation
19         $\mathcal{C}_r[p_{ini}] := \mathcal{C}_r[p_{ini}] + 1$
20   **effect** $(e, p_{ini}, \mathcal{C}_r)$
21   **let** $t : (e, t) \in T$
22   **if** $\exists k : t[k] < \mathcal{C}_r[k]$ **then** ▷ There are unrecorded rmv operations in $\mathcal{C}_r$.
23   Remove (e, p_{ini}) from E
24   Remove (e, v_{inn}, v_{acq}) from V
25   **let** $t' : \forall k : t'[k] := max(\mathcal{C}_r[k], t[k])$
26             $T := T \setminus \{(e, t)\} \cup \{(e, t')\}$
27 **update** $upd(e)$
28   **prepare** (e)
29   **pre** e is in the data collection
30   **let** $\mathcal{C}_r = t$ s.t. $(e, t) \in T$
31   **effect** $(e, \mathcal{C}_r)$
32         $rmv(e, p_{ini}, \mathcal{C}_r)$ ▷ Execute the effect part of $rmv(e)$ using $\mathcal{C}_r$.
33   **let** $t : (e, t) \in T$
34   **if** $\mathcal{C}_r = t$ **then** ▷ The remote replica and the add operation are in the same phase.
35             $\langle$Modify the acquired value v_{acq} for $e\rangle$ ▷ Resolve possible conflicts between concurrent upds, using p_{ini} to obtain the replica info if necessary.

5.1 CRPQ Design

Since conflicts concerning element existence is handled by the RWSet, the user only needs to care about element values. The user needs to specify how priority values are initialized and updated by the CRPQ APIs. More importantly, the user needs to develop conflict-resolving stubs and "plug" them into the RWSkeleton.

As for the *add-upd* conflict, the priority value of an element e is divided into two parts: the *innate value* set by its initiating $add(e)$ operation, and the *acquired value* updated by the following $inc(e, i)$ operations. In the CRPQ design, the priority value exposed to the upper-layer application is the sum of innate and acquired values. The *add* and *upd* operations take effects on the innate and acquired values respectively and conflicts are prevented.

As for the *add-add* conflict, the user needs to specify an total order among concurrent *add* operations. This order decides the unique *add* that finally "wins", while other *add*s are overwritten. In our exemplar design, we can simply specify "largest replica *id* wins" (assuming that the *id*s of all replicas are totally ordered).

As for the *upd-upd* conflict, there will be no this type of conflict in the priority queue case. It is because the add/subtraction of priority values (integers) naturally commute.

The detailed CRPQ design is presented in Algorithm 4.

Algorithm 4: Remove-Win CRPQ

payload E: set of (e, p_{ini}) tuples, T: set of (e, t) tuples, V: set of (e, v_{inn}, v_{acq}) tuples
initial $E = \emptyset, T = \emptyset, V = \emptyset$
query *empty*(): boolean
  return $E \neq \emptyset$
query *lookup*(e): boolean
  return $\exists p_{ini} : (e, p_{ini}) \in E$
query *get_pri*(e): integer
  pre $lookup(e)$
  let $x, \delta : (e, x, \delta) \in V$
  return $x + \delta$
query *get_max*(): id, integer
  pre $\neg empty()$
  let $e : lookup(e) \wedge \forall o : lookup(o) \wedge get\_pri(o) \leq get\_pri(e)$
  return $e, get\_pri(e)$
update $add(e, x)$
  prepare (e, x)
    pre $\neg lookup(e)$
    let $\mathcal{C}_r = t$ s.t. $(e, t) \in T$
    let p_{ini} be id of the initiator of this operation
  effect $(e, x, p_{ini}, \mathcal{C}_r)$
    $rmv(e, \mathcal{C}_r)$ ▷ Execute the effect part of $rmv(e)$ using $\mathcal{C}_r$.
    let $pid : (e, pid) \in E$
    let $t : (e, t) \in T$
    if $\mathcal{C}_r = t \wedge p_{ini} > pid$ **then**
      $E := E \setminus \{(e, pid)\} \cup \{(e, p_{ini})\}$
      let $x', \delta : (e, x', \delta) \in V$
      $V := V \setminus \{(e, x', \delta)\} \cup \{(e, x, \delta)\}$
update $inc(e, i)$ ▷ $i \in \mathbb{R}$, $i < 0$ means decrease.
  prepare (e, i)
    pre $lookup(e)$
    let $\mathcal{C}_r = t$ s.t. $(e, t) \in T$
  effect $(e, i, \mathcal{C}_r)$
    $rmv(e, \mathcal{C}_r)$ ▷ The same as the effect part of *add*.
    let $t : (e, t) \in T$
    if $\mathcal{C}_r = t$ **then**
      let $x, \delta : (e, x, \delta) \in V$
      $V := V \setminus \{(e, x, \delta)\} \cup \{(e, x, \delta + i)\}$
update $rmv(e)$
  prepare (e)
    pre $lookup(e)$
    let $\mathcal{C}_r = t$ s.t. $(e, t) \in T$
    let p_{ini} be id of the initiator of this operation
    $\mathcal{C}_r[p_{ini}] := \mathcal{C}_r[p_{ini}] + 1$
  effect $(e, \mathcal{C}_r)$
    let $t : (e, t) \in T$
    if $\exists k : t[k] < \mathcal{C}_r[k]$ **then**
      let $pid : (e, pid) \in E$
      $E := E \setminus \{(e, pid)\} \cup \{(e, -1)\}$
      let $x, \delta : (e, x, \delta) \in V$
      $V := V \setminus \{(e, x, \delta)\} \cup \{(e, 0, 0)\}$
      let $t' : \forall k : t'[k] := max(\mathcal{C}_r[k], t[k])$
      $T := T \setminus \{(e, t)\} \cup \{(e, t')\}$

5.2 Illustrating Examples

We use three examples to better illustrate the design of our CRPQ. This first example mainly shows how the remove-win strategy works. The second example shows how the conflict resolution among non-remove operations within one phase works. The third example mainly discusses the difference between the visibility and causal visibility relations.

In the remove-win example in Fig. 5, the *rmv* operation initiated by p_1 is concurrent with the *add* and *inc* operations initiated by p_0. On p_1, after the *rmv* operation is executed the crh-vector of e in T is set to $v_1 = [0, 1]$, which is larger than the crh-vectors of *add* and *inc* on p_0. So when the remote events of *add* and *inc* arrives at p_1, they will be safely ignored, and the payload on p_1 remains unchanged whether *add* and *inc* arrive or not. When the remote event of *rmv* from p_1 is received by p_0, p_0 will remove the element e from E, since the *rmv* carries the larger crh-vector v_1.

In the example of conflict resolution among non-remove operations in Fig. 6, the payloads of p_0 and p_1 are initially empty. First, we have p_0 and p_1 add the element e concurrently, with the same crh-vector v_0. This indicates that they belong to the same phase and need conflict resolution. Here we adopt the strategy that "larger replica id wins". Thus the *add* of p_1 wins. We find that the tuple in E on p_0 remains (e, p_0) until it finally receives the *add* operation from p_1 and the tuple in E is changed to (e, p_1). Then we have p_0 and p_1 increase e with the crh-vector v_0, and the increased values merged without conflict into the acquired value of e. Finally p_0 and p_1 converge to the same state.

In the example in Fig. 7, we compare the visibility and the causal visibility relations. The *rmv* initiated by p_0 is causally visible to the *inc* initiated by p_2, but the *rmv* is not visible to the *inc*. The crh-vector is initially $v_0 = [0, 0, 0]$. The *rmv* on p_0 updates the crh-vector to $v_1 = [1, 0, 0]$. Then v_1 is transmitted to from p_0 to p_1 and from p_1 to p_2. Thus when the *rmv* operations arrives late at p_2 (bringing with it the crh-vector v_1), it will be safely ignored since p_2 has already obtained the chr-vector v_1 before. If we only use the visibility relation, the rmv from p_0 will arrive at p_2 late and falsely removes element e. Causal message delivery is necessary to ensure that on p_2, rmv is delivered before add.

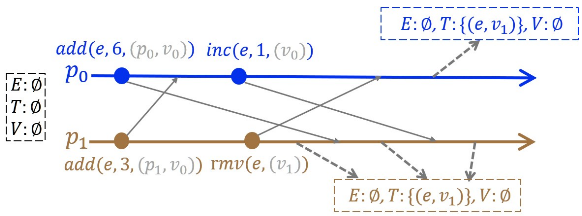

Figure 5: An example showing how a rmv wins, where $v_0 = [0, 0]$, $v_1 = [0, 1]$.

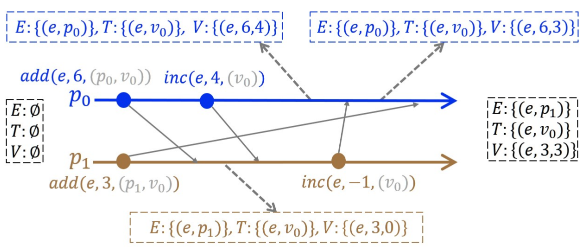

Figure 6: Conflict resolution among non-remove operations, where $v_0 = [0, 0]$.

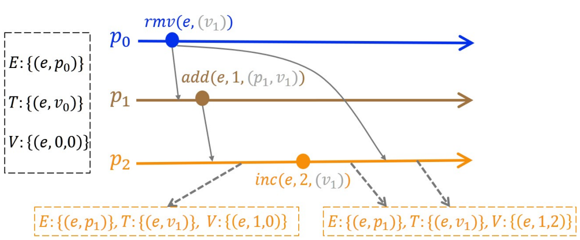

Figure 7: Difference between visibility and causal visibility, where $v_0 = [0, 0, 0]$, $v_1 = [1, 0, 0]$.

6. PERFORMANCE MEASUREMENTS

In this section, we first discuss the implementation of our remove-win CRPQ, and present the design of experiments. Then we discuss the evaluation results.

6.1 Implementation

Our CRPQ is implemented over Redis [1], and we now describe the most important implementation details. Redis is an open-source in-memory key-value store, which supports a variety of data types, e.g, lists, sets and hash maps. However, Redis operates in a master-slave manner and does not provide CRDTs.

To implement and evaluate our CRPQ, we first re-organize the servers/replicas in Redis into the peer-to-peer architecture, where all replicas act as masters and can serve both update and query operations from all the clients. We reuse the basic functionalities of Redis, including the data types and abstractions, the event library and event handling mechanism, as well as the network communication.

Then we implement the CRDT framework where the logic for update operations is explicitly divided to the **prepare** part and the **effect** part. In the **prepare** part, the initiating replica conducts local processing, quickly respond to the client, and prepare information to be broadcast. In the **effect** part, local updates are asynchronously propagated to all the remote replicas. The remote replicas accept the updates, resolve possible conflicts and update the replica state.

Based on the preparations above, we then implemented our CRPQ (Algorithm 4). Our implementation is available at [2].

6.2 Experiment Setup

The experiment is conducted on a PC with an Intel I7-6700 quad core CPU (3.40GHz) and 16GB RAM, running Windows 10 enterprise v1803. We use VMware workstation 14 pro to run 4 virtual machines, as shown in Fig. 8. VM 1˜3 are set to have a dual core CPU and with 3GB RAM, running Ubuntu server 16.04.5 LTS. VM4 is set to have two dual core CPUs and with 4GB RAM, running Ubuntu desktop 16.04.5 LTS.

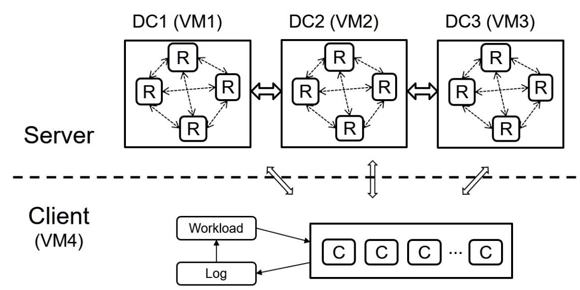

Figure 8: Architecture of the experiment system.

Each of VM 1˜3 simulates a data center. It runs 1˜5 instances of Redis. We use traffic control (TC) [3] provided by the Linux kernel to control the network delay among Redis instances. Another VM4 runs all the clients. The clients obtain when and what operations to issue to the servers from the workload module. The workload module generates workloads of different patterns. The clients log statics about how operations are served by the servers in the log module. When generating the operations, the workload module needs to query the log module from time to time, to obtain current status of the CRPQ. This is because the workload module may need to intentionally generate conflicting update operations. Also, it needs to prevent invalid operations such as removing an element that does not exist in the CRPQ.

The key space for elements in the CRPQ has size 200,00. The workload module randomly chooses elements to be added from all possible ones. The inc and rmv operations are conducted on random elements which are currently in the CRPQ. The initial value of elements are randomly chosen from integers ranging from 0 to 100. The value increased is randomly chosen from -50 to 50.

To evaluate the performance of a CRDC, we need to intentionally create conflicting operations on the same element. When the workload module generates the latest operation o, it will pair o with all operations which are less than μ units of time before o. Here, μ is the average message delay of intra-data center communication. The workload module is concerned of *add-add* and *add-rmv* pairs. All such pairs has probability 15% to execute on the same data element. Note that we do not explicitly control the conflict for *inc-rmv* pairs. It is because there will be fairly high probability of such conflicts. All workloads we consider have 59%~89% operations which are *inc* or *rmv*.

6.3 Experiment Design

Since the CRDT serves operations instantly by design, it statistically has the same performance in terms of query/update delay. However, there is the intrinsic tradeoff between data consistency and response latency. Thus we need to measure the data consistency, in order to show how much data consistency is sacrificed to get the good performance in the response delay. As for a priority queue, we measure the average error (denoted by $\bar{x}$), which is the difference between the return value of *get_max* and the real *max* value. The error is averaged among all *get_max* operations. We also measure the error ratio (denoted by f), which is the probability that a *get_max* operation does not correctly returns the *max* value. The order in which queries/updates are logged on the client side is approximately the order they are served by the servers. We use this total real-time oder to decide the status of the priority queue and calculate the correct *max* values. We also record the meta-data overhead for resolving conflicts by the the CRPQ. The meta-data overhead is averaged among all elements in the priority queue.

We compare our remove-win CRPQ with an add-win CRPQ. The add-win CRPQ is designed by augmenting the Add-Win Set (also known as Observed-Remove Set) [18]. The basic idea of an Add-Win CRPQ is to record all the context of update operations and resolve conflicts basing on the history or context of execution. More details about the design of the Add-Win CRPQ is provided at [2], and the implementation is available at [2].

We design experiments to explore the influences of different key factors on the performance of the CRPQ. The influences of the workload pattern and the concurrency among operations are explored. To control the concurrency among operations, we change the speed at which operations are issued from clients to servers, the network delay and the number of replicas.

In the experiments, all key factors are set to their default values. In each experiment below, we will choose to vary one factor and explore its impact on the performance of the CRPQ. The default workload pattern is the *inc*-dominant pattern defined in Section 6.4. The default operation speed is set to 10,000 ops/s. The default inter-data center communication delay follows $\mathbb{N}(50, 10)$ [4], while the default intra-data center delay follows $\mathbb{N}(10, 2)$. The default setting of replication is 3 Redis instances in each of the three data centers. Statistics reported are the average over 20 runs.

6.4 Impact of Workload Patterns

In this experiment, we compare the performance of two CRPQs under different patterns of workloads. In the *inc* dominant workload, 80% operations are *inc*, 11% operations are *add* and 9% operations are *rmv* [5]. In the *add-rmv* dominant workload, 41% operations are *add*, 39% operations are *rmv* and 20% operations are *inc*. We generate 20,000,000 operations in total for each workload pattern.

[4] $\mathbb{N}(\mu, \sigma)$ stands for the normal distribution, where μ is the mean and σ is the standard deviation.

[5] We make the *add* operations slight more than *rmv* to prevent the CRPQ from being often empty.

Table 1: Statistics of $\bar{x}$ and f.

	inc-dominant		*add/rmv*-dominant	
	$\bar{x}$	f	$\bar{x}$	f
Add-Win	5.19	0.12	4.80	0.27
Rmv-Win	3.80	0.09	4.53	0.25

In the *inc* dominant workload, *get_max* occasionally returns wrong max priority values. How the average error changes over time is shown in Fig. 9. The average error and the error ratio are shown in Table 1. We find that $\bar{x}$ and f are fairly small, for both types of CRPQs, while the remove-win CRPQ is slightly better.

As for the mete-data overhead, the overhead of add-win CRPQ increases linearly, while that of the remove-win CRPQ is stable. This is mainly because the add-win CRPQ needs to record the history of execution, which is linear with the number of operations executed. The remove-win CRPQ only records the latest remove operation in the crh-vector.

In the *add/rmv*-dominant workload, the evaluation results are intrinsically similar, and we focus on the differences here. Both the average error $\bar{x}$ and the error ratio f vibrate more significantly, as shown in Fig. 10. Although the average error values are close, the error probabilities are more than twice of their counterparts in the *inc*-dominant workload, as shown in Table 1. This is mainly because, in the *add/rmv*-dominant workload, data items enter and leave the CRPQ more frequently, while in the *inc*-dominant workload, data elements in the queue are relatively stable, only their priority values change more frequently. Thus in the *add/rmv*-dominant workload, the max priority value in the queue are frequently changed abruptly, due to the add and deletion of data elements. This also explains why the meta-data overhead vibrates more significantly in the *add/rmv*-dominant pattern.

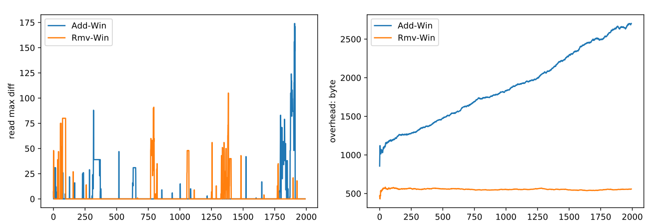

Figure 9: CRPQ performance under the *inc*-dominant workload.

6.5 Impact of Concurrency among Operations

There are three environment factors we can tune to control the impact of concurrency among operations. Thus, we conduct three experiments accordingly, tuning one factor in each experiment. Specifically, to control the concurrency among operations in the time dimension, we tune the speed at which operations are issued from clients to the servers. We increase the operation speed from 500 to 10000 ops/s. To control the concurrency in the space dimension, we change the network delay and the number of replicas. We tune the

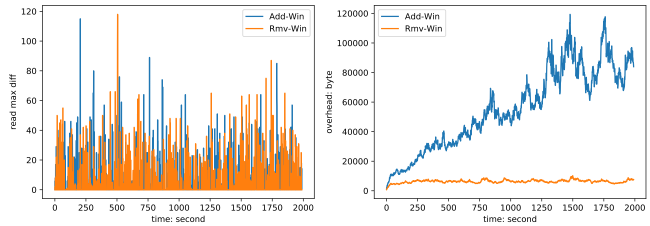

Figure 10: CRPQ performance under the *add*/*rmv*-dominant workload.

inter-data center delay from $\mathbb{N}(20, 4)$ to $\mathbb{N}(380, 76)$, and tune the intra-data center delay from $\mathbb{N}(4, 0.8)$ to $\mathbb{N}(76, 15.2)$. As for the number of replicas, we increase the number of Redis instances from 1 to 5 in every data center.

As for the data consistency, we find that the average error $\bar{x}$ and the error ratio f increase linearly with the concurrency among operations for both types of CRPQs, as shown in Fig. 11, 12 and 13. It is mainly because the CRPQ guarantees strong eventual consistency, and the inconsistency is mainly determined by the number of operations that are yet to be synchronized. As the concurrency among operations increases, the number of operations to be synchronized increases linearly. Thus we have $\bar{x}$ and f increase linearly.

As for the meta-data overhead, at the end of each run of the experiment, we measure the total meta-data overhead and average it over the number of elements in the queue. We find that the operation speed and the network delay have little impact on the meta data overhead, as shown in Fig. 11 and 12. As long as the number of operations conducted on the queue is statistically similar, the meta data overhead is also similar. This is mainly because, the meta-data overhead of our Remove-Win CRPQ is due to the causal remove history vector, while for the Add-Win CRPQ, the meta data overhead is mainly due to the fact that each *rmv* operation needs to record all the *add* operations it has seen. The way meta-data is recorded for both types of CRPQs decides that the operation speed and the network delay have little impact on the final meta-data overhead.

This also explains why the meta-data overhead of our Remove-Win CRPQ increases in Fig. 13. The dimension of the causal remove history vector is equal to the number of replicas in the server side. Thus the meta-data overhead (for recording the vector) increases linearly as the number of replicas increases. As for the Add-Win CRPQ, the meta data is not affected by the increase in the number of replicas for similar reasons discussed above.

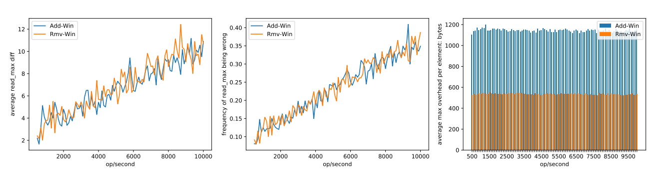

Figure 11: Effects of tuning the operation speed.

6.6 Discussions

The comparison with the Add-Win CRPQ better illustrates the advantages of the remove-win strategy. Owing to

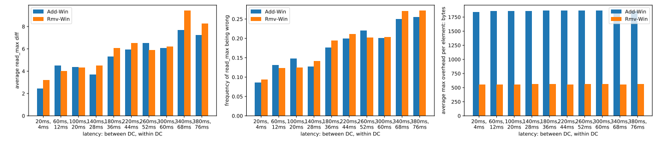

Figure 12: Effects of tuning the network delay.

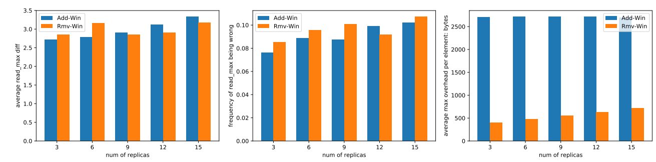

Figure 13: Effects of tuning the number of replicas.

the simple but powerful remove-win strategy and to the efficient encoding of the causal remove history, the meta-data overhead of Remove-Win CRPQ is intrinsically low. The meta-data overhead also remains stable, as more operations are conducted on the queue.

Though the meta-data overhead is low, the data consistency achieved by our Remove-Win CRPQ is statistically similar to (actually slightly better than) that of the Add-Win CRPQ.

For *inc*-dominant workloads, the advantage of the Remove-Win CRPQ is more significant. In the *add*/*rmv*-dominant workload, both types of CRPQs have similar performance while the Remove-Win CRPQ is only slightly better.

7. RELATED WORK

Conflict resolution is the essential issue in the design of CRDTs, and various resolution strategies have been proposed, e.g., add-win, last-write-win and remove-win. The Add-Win Set proposed in [18] lets each *rmv* operation record all *add* operations it has seen. The effect of *rmv* operations is limit to these *add* operations it has seen, which makes the *add* operation wins over the concurrent *rmv*. The Last-Write-Win Set [18] requires a timing service, which labels all events with unique, totally ordered timestamps. With the timing service, the "last" one of conflicting writes is well defined and agreed by all replicas, which resolves the conflict. Although these two strategies are simple and intuitive, it is not clear how to and may be hard to extend them to the design of CRDCs in which data elements have complex values. The design of the Remove-Win Set proposed in [24] is dual to that of the Add-Win Set. Each *add* operation is required to record all the *rmv* operations it has seen. The effect of *add* operations is limited to these *rmv* operations it has seen, which makes the *rmv* operation wins over the concurrent *add*. In this work, we further utilize the potential of the remove-win strategy by defining the visibility relation and then introduce the concept of phase segmented by remove operations. The efficiency of our remove-win design is further improved by the encoding using the crh-vector.

Existing CRDT designs are often obtained via derivations from seminal and widely-used designs, which motivates us to propose our design framework. In the area of collaborative editing, the WOOT model is proposed, which essentially designs a conflict-free replicated list [16]. The basic idea is to record the local order among characters in the string. The local orders from multiple replicas form a partial order, which is linearly extended based on the total order among the replica ids. Multiple improved designs following WOOT were proposed, including WOOTO [23], which used a degree scheme to capture the relative ordering of concurrent object creations and save one round of object sequence search; and WOOTH [6], which used a hash scheme to speed up the search of neighboring objects. In the area of computational CRDTs, a class of CRDTs whose state is the result of a computation over the executed updates, a brief study is presented in [15] and three generic designs are proposed. The non-uniform replication model is further proposed to reduce the cost for unnecessary data replication, which is often seen in computational scenarios [12]. Though existing derivations of CRDT designs are mainly driven by the application scenarios, our Remove-Win Design Framework focus on the data type itself. The design framework is for the widely-used data collection type and can be used in a variety of application scenarios.

CRDTs are also implemented in popular NoSQL databases. Roshi implements a time-series event storage based on the last-write-win set on top of Redis [5] . Riak provides state-based CRDTs called RiakDTs, including Flag, Register, Counter, Set and Map, [4]. The Map in Riak can also be used as a container of complex data values, which motivates the design of our design framework. However, the map in Riak can only contain data elements of the RiakDT. Our framework aims at provide any user-specified data types, and the value-update and conflict resolution logic is left to the user. Our framework only provides the skeleton of the CRDC design. Moreover, as for the conflict resolution strategy, Riak uses add-win, while our framework uses remove-win.

8. CONCLUSION

In this work, we propose the Remove-Win Design Framework to guide the design of CRDCs. The framework has at its core the RWSet, which handles the conflicts concerning the existence of elements. Then the user can augment the RWSet to assign application-specific values to data items in the data collection. Conflict resolution concerning the values can be developed under the guidance of the RWSkeleton. We demonstrate the effectiveness of our approach via a case study of designing a conflict-free replicated priority queue. Performance measurements also show the efficiency of our design.

In the future work, we will design more CRDCs using the Remove-Win Framework. We will also formally specify and verify the designs and implementations of the CRDCs we develop. More comprehensive experimental evaluations under various workloads are also necessary.

9. REFERENCES

[1] http://redis.io/.

[2] https://github.com/elem-azar-unis/CRPQ-Redis.

[3] http://tldp.org/HOWTO/Traffic-Control-HOWTO/index.html.

[4] Riak distributed database. `https://riak.com/`, 2019.

[5] Roshi. `https://github.com/soundcloud/roshi`, 2019.

[6] M. Ahmed-Nacer, C.-L. Ignat, G. Oster, H.-G. Roh, and P. Urso. Evaluating crdts for real-time document editing. In *Proceedings of the 11th ACM Symposium on Document Engineering*, DocEng '11, pages 103–112, New York, NY, USA, 2011. ACM.

[7] H. Attiya, S. Burckhardt, A. Gotsman, A. Morrison, H. Yang, and M. Zawirski. Specification and complexity of collaborative text editing. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, PODC '16, pages 259–268, New York, NY, USA, 2016. ACM.

[8] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Trans. Comput. Syst.*, 9(3):272–314, Aug. 1991.

[9] E. A. Brewer. Towards robust distributed systems (abstract). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC'00, pages 7–, New York, NY, USA, 2000. ACM.

[10] J. Brutlag. Speed matters for google web search. `https://services.google.com/fh/files/blogs/google_delayexp.pdf`. Accessed: 04-13-2019.

[11] S. Bussey. Distributed in-memory caching in elixir. `https://stephenbussey.com/2019/01/29/distributed-in-memory-caching-in-elixir.html`. Accessed: 04-13-2019.

[12] G. M. Cabrita. Non-uniform replication for replicated objects, Mar. 2017.

[13] M. Emmi and C. Enea. Weak-consistency specification via visibility relaxation. *Proc. ACM Program. Lang.*, 3(POPL):60:1–60:28, Jan. 2019.

[14] F. Mattern. Virtual time and global states of distributed systems. In *Proc. International Workshop on Parallel and Distributed Algorithms*, pages 215–226, Holland, 1989.

[15] D. Navalho, S. Duarte, and N. Preguiça. A study of crdts that do computations. In *Proceedings of the First Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC '15, pages 1:1–1:4, New York, NY, USA, 2015. ACM.

[16] G. Oster, P. Urso, P. Molli, and A. Imine. Data consistency for p2p collaborative editing. In *Proceedings of the 2006 20th Anniversary Conference on Computer Supported Cooperative Work*, CSCW '06, pages 259–268, New York, NY, USA, 2006. ACM.

[17] N. Preguiça. Conflict-free replicated data types: An overview. *arXiv preprint arXiv:1806.10254*, 2018.

[18] 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.

[19] 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.

[20] K. Sivaramakrishnan, G. Kaki, and S. Jagannathan. Declarative programming over eventually consistent data stores. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 413–424, New York, NY, USA, 2015. ACM.

[21] P. Veríssimo. Causal delivery protocols in real-time systems: A generic model. *Real-Time Systems*, 10(1):45–73, Jan 1996.

[22] H. Wei, Y. Huang, and J. Lu. Specification and implementation of replicated list: The jupiter protocol revisited. In *22nd International Conference on Principles of Distributed Systems, OPODIS 2018, December 17-19, 2018, Hong Kong, China*, pages 12:1–12:16, 2018.

[23] S. Weiss, P. Urso, and P. Molli. Wooki: A p2p wiki-based collaborative writing tool. In B. Benatallah, F. Casati, D. Georgakopoulos, C. Bartolini, W. Sadiq, and C. Godart, editors, *Web Information Systems Engineering – WISE 2007*, pages 503–512, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[24] M. Zawirski. *Dependable Eventual Consistency with Replicated Data Types*. Theses, Universite Pierre et Marie Curie, Jan. 2015.