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

Yuqi Zhang, Hengfeng Wei, and Yu Huang

Abstract—Distributed storage systems employ replication to improve performance and reliability. To provide low latency data access, replicas are often required to accept updates without coordination with each other, and the updates are then propagated asynchronously. This brings the critical challenge of conflict resolution among concurrent updates. Conflict-free Replicated Data Type (CRDT) is a principled approach to addressing this challenge. However, existing CRDT 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 CRDTs.
To address this challenge, we propose RWF – the Remove-Win design Framework for CRDTs. RWF leverages the simple but powerful remove-win strategy to resolve conflicting updates, and provides generic design for a variety of data container types. Two exemplar implementations following RWF are given over the Redis data type store, which demonstrate the effectiveness of RWF. Performance measurements of our implementations further show the efficiency of CRDT designs following RWF.

Index Terms—CRDT, remove-win, replicated data store

1 Introduction

Internet-scale distributed systems often replicate application state and logic to reduce user-perceived latency and improve application throughput, while tolerating partial failures [1], [2]. 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. Thus, many Internet-scale distributed systems are designed for low latency and high availability in the first place [3], [4]. To provide low latency and high availability, the update requests must be handled immediately, without communication with remote replicas. Updates can only be asynchronously transmitted to remote replicas, and rolling-back updates to handle conflicts is not acceptable.

According to the CAP theorem, low latency and high availability can only be achieved at the cost of accepting weak consistency [5], [6]. To provide certain guarantees to developers of upper-layer applications, *eventual convergence* is widely accepted, which ensures that when any two replicas have received the same set of updates, they reach the same state [7]. Eventually consistent replicated data types are widely used in scenarios where responsiveness is critical, e.g. in collaborative editing [8], distributed caching [9] or coordination-avoidance in databases [10]. The design of replicated data types guaranteeing eventual convergence brings the challenge of conflict resolution for concurrent updates on different replicas of logically the same data element. The Conflict-free Replicated Data Type (CRDT) framework provides a principled approach to addressing this challenge [1], [2].

- *Yuqi Zhang, Hengfeng Wei, and Yu Huang are with the State Key Laboratory for Novel Software Technology, and Department of Computer Science and Technology, Nanjing University, China, 210023. E-mail: cs.yqzhang@gmail.com, {hfwei, yuhuang}@nju.edu.cn*

(Corresponding author: Hengfeng Wei and Yu Huang.)

The conflict resolution is typically hard and error-prone, especially for data types having complex semantics. This explains why existing CRDT designs are tricky, and why it is hard to generalize design for one type to other similar types [1], [2]. A design framework is in great need to guide the systematic design of new CRDTs, and the design of CRDTs needs to shift from a craft to an engineering discipline. The essential issue of proposing a design framework is to refine the commonalities among different CRDT designs. Thus the developer can focus on designing special features pertinent to each data type and reuse the common design based on the framework. In this way, the design framework can help even not-experienced developers handle complex and error-prone CRDT designs.

Toward this objective, we propose RWF – the Remove-Win design Framework for CRDTs. RWF aims at facilitating the design of replicated data container types. A data container is first a set of unique data elements. Existence of each element is identified by its *key*. Moreover, each data element can have *values*. Complex semantics of the data type and the structure among the data elements are "encoded" in the values of the data elements.

RWF facilitates the design of replicated data container types leveraging the simple but powerful remove-win strategy for conflict resolution. The basic rationale of the remove-win strategy is that when any operation is concurrent with a remove operation, the remove operation wins. This means that the data element involved in the operations will be eliminated from the container. One salient feature of the remove-win strategy is that, it is independent of the semantics of the data type under concern. The remove operation simply eliminate the data element, no matter how complex the semantics of the data type are. Though elimination of one element may affect the overall structure of the data container, the maintenance of the structure of the container is independently handled by each replica and requires no coordination with remote peer replicas. The salient feature

of the remove-win strategy makes it applicable to different data container types and one design framework is proposed to capture the common remove-win resolution for different data types.

Note that the remove-win strategy adopted in RWF is different from the remove-win strategy used in the existing work, e.g. in the Remove-Win Set [11]. When a non-remove operation is concurrent with a remove operation, the remove-win strategy in the existing work makes all replicas put the remove operation behind the non-remove operation. Thus the effect of the remove operation will overwrite that of the preceding operations. In the remove-win strategy used in RWF, the data element is simply eliminated, requiring no further processing. Our strategy is more simple but also more powerful. It can be more easily applied to different data types.

RWF provides a generic algorithm skeleton for conflict-free replicated data container types (denoted as RWF-DTs). User-defined logics are implemented as stubs and inserted into the skeleton to obtain concrete RWF-DT designs. The RWF framework can be implemented over different data type stores. We present an exemplar implementation over the widely used Redis data type store. In the implementation level, RWF provides a template for RWF-DT implementations. Common logics of CRDTs as well as those of RWF-DTs are provided in the template. The user only needs to provide logics pertinent to the specific data type under development.

The usefulness of RWF is illustrated by two exemplar RWF-DT implementations – implementation of a priority queue and that of a list. Performance measurements of our implementations also show the efficiency of CRDT designs following RWF.

The rest of this work is organized as follows. In Section 2, we overview our design framework. In Section 3 and 4, we present the generic design of RWF-DTs and provide an exemplar implementation. Section 5 presents the performance evaluation results. Section 6 discusses the related work. In Section 7, we summarize our work and discuss the future work.

2 RWF Overview

The RWF design framework first decomposes the design of RWF-DTs into two dimensions. It then provides a template for RWF-DT implementations, as detailed below.

2.1 Design of RWF-DTs

The RWF design framework refines the commonalities in CRDT design from two dimensions, as shown in Fig. 1. RWF first extracts the commonalities from different data types. RWF focuses on the data container types. Each element in the container first has its unique existence, which is modified by the add and rmv operations. Each data element can also be associated with values, which is modified by the upd operation[1]. Elements in the container may collectively form complex data structures, such as lists, queues and trees. The data structure info is encoded in the value of each element.

1. Possibly a container type can have multiple upd operations. We mention only one upd operation for the ease of presentation. Also we only consider "pure" operations, i.e. each operation is either a query or an update.

RWF employs the *remove-win* strategy to resolve conflicts between concurrent updates. For conflicting updates involving a rmv and a non-remove operation (i.e., add or upd), the rmv operation just eliminates the existence of the data element, no matter what value the element has. For non-remove operations, RWF requires the user provide conflict resolution logics. The remove-win strategy common to different RWF-DTs is implemented in an *algorithm skeleton*. User-specified conflict resolution logics are implemented as *stubs*, which can be inserted into the skeleton to obtain concrete RWF-DT designs, as detailed in the following Section 3.

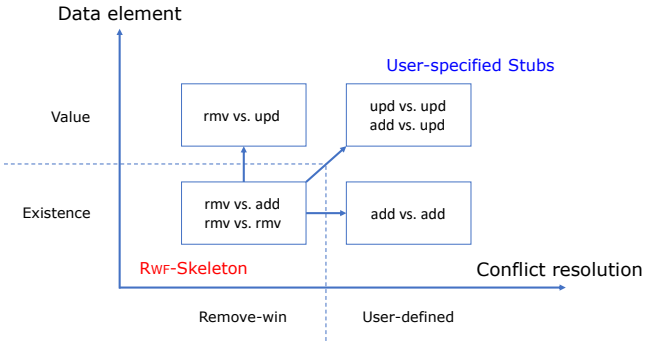

Fig. 1. Two dimensions in RWF-DT design.

2.2 Implementation of RWF-DTs

Based on the commonalities in the design, RWF further provides a template for RWF-DT implementations, as shown in Fig. 2. The template has the "onion" structure and consists of three levels, namely the CRDT level, the RWF level and the user-defined data type level (denoted as the DT level in short).

In the CRDT level, the basic structure of the implementation is decided, following the operation-based CRDT algorithm framework [7]. Common operations required by the CRDT framework are implemented as tool functions/macros and can be reused for different RWF-DTs.

In the RWF level, common metadata pertinent to the predetermined remove-win strategy is defined. Common operations pertinent to the remove-win strategy are also implemented as tool functions.

In both the CRDT level and the RWF level, tool functions contain logics which are generic and independent of the specific type of data element in the data container. The user only needs to pass specific type of the data element to the tool functions in the DT level. Moreover, the user also needs to provide conflict-resolution logics which can only be decided by the users.

3 RWF-DT Design

In this section, we first describe the system model. Then we present design of the RWF-Set, which is the core of RWF-DT design. Finally, an algorithm skeleton is presented.

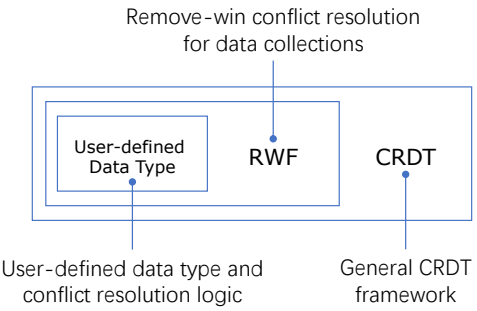

Fig. 2. Three layers in the RWF-DT implementation.

3.1 System Model

We use the typical system model for CRDT [1]. Suppose there are n server processes $p_0, p_1, \cdots, p_{n-1}$, each holding one replica of an RWF-DT. Servers are interconnected by an asynchronous network, and can only fail by crash. Messages may be delayed, reordered 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 $o.e_l$ on p_i, and n remote events, one remote event $o.e_r$ for each replica, including p_i itself[2]. Here, we say the operation o has executed on replica p_i at time t, denoted by $o \in E(p_i^t)$ where p_i^t is the replica state of p_i at time t[3], and $E(p_i^t)$ is the set of executed operations of p_i^t, if $o.e_l$ or any of $o.e_r$ has taken place on p_i. We define function 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 RWF-DTs:

Definition 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 $o.e_l$ and each remote event $o.e_r$ belonging to the same operation o have the local-remote order, denoted by $\xrightarrow{lr}$.

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 2 (visibility). Operation o_1 is visible to o_2, denoted by $o_1 \xrightarrow{vis} o_2$, if $o_1.e_l \rightarrow o_2.e_l$. Operation o is visible to replica state p^t, if $o \in E(p^t) \vee \exists o' : o' \in E(p^t) \wedge o \xrightarrow{vis} o'$. □

Note that the $\xrightarrow{vis}$ relation is transitive. 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.

2. For the ease of presentation, the remote event on the initiating process is omitted.

3. We use p^{cur} to denote the current state of replica p.

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 eliminated by this remove operation.

3.1.2 Segmenting System Execution into Phases

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 wipes off everything and ends the current phase, and then starts a new phase from scratch. Phase-based resolution is central to the design of RWF-DTs, as detailed below.

To define the concept of phase, we first define the remove history of an operation and a replica state:

Definition 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 $\mathcal{H}_r(p^t)$ of one replica state p^t is defined as the union of remove histories of all operations executed on this replica, together with all the remove operations executed on this replica:

$$\mathcal{H}_r(p^t) = \cup_{o \in E(p^t)} \mathcal{H}_r(o) \cup \{o \mid \text{TYPE}(o) = rmv,\ o \in E(p^t)\}$$

□

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

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

Definition 4 (phase). Operations and replica states belong to the same phase, if they have the same remove history. Or equivalently, the phases of the system execution are the equivalence classes in $(O \cup S)/\approx_{\mathcal{H}_r}$, where O is the set of operations, S is the set of replica states, and $\approx_{\mathcal{H}_r}$ is the equivalence relation defined by $\mathcal{H}_r(\cdot)$:

$$a \approx_{\mathcal{H}_r} b \triangleq \mathcal{H}_r(a) = \mathcal{H}_r(b).$$

We denote the phase that operation/replica state a belongs to as $[a]$. □

Phases are temporally ordered. We say $[a] \prec [b]$, if $\mathcal{H}_r(a) \subset \mathcal{H}_r(b)$.

3.2 Design of the RWF-Set

Given the definition of $\xrightarrow{vis}$ and $\mathcal{H}_r(\cdot)$, we can now present the design of an RWF-DT. For the ease of presentation, we first present the core of the design, which is the design of an RWF-Set. Then we augment the design of the RWF-Set into an algorithm skeleton, which greatly simplifies the design of various replicated data container types.

3.2.1 Encoding of Remove History

Since our design is centered around the remove history, we first discuss how to efficiently encode the remove history for each operation. 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 off everything) no matter how the value of the data element has changed. Thus we do not care how many times the remove operations have taken place. If the k^{th} remove operation that is initiated by p_i is visible, all remove operations, from the 1^{st} to the $(k-1)^{th}$, initiated by p_i are visible as well. Since the remove operation is idempotent, we only need to record the last remove operation initiated on p_i.

The encoding/decoding scheme we use is principally the vector clock [12]. The remove operations visible to an operation o or some replica state p^t can be encoded as a vector $v[1..n]$, which we call the remove history vector (abbreviated as rh-vec). 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 $v[j] = k$ on replica p_i, it means that the last remove operation initiated by p_j that is visible to p_i^{cur} is p_j's k^{th} remove operation (remove operations visible to an operation o is defined similarly). When replica p_i receives an operation o carrying a rh-vec $v[1..n]$, p_i's local rh-vec $v_i[1..n]$ needs to be updated as: $\forall j \in [1..n] : v_i[j] = \max(t_i[j], t[j])$.

3.2.2 Payload of an RWF-Set

Following the CRDT framework, each RWF-Set $\mathcal{S}$ is implemented over its payload, two sets E and T. On one replica of $\mathcal{S}$, set E contains the IDs of data elements. Element $e \in E$ basically means that this element is in $\mathcal{S}$. Set T is the set of tuples (e, t), where tag t is the rh-vec encoding the remove history of the current replica state, concerning data element e.

We first discuss how add and rmv operations update the payload. 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 4 – 7 in Algorithm 1[4]). Replica p_i checks whether e is already in $\mathcal{S}$ (Line 5). If not, the remove history of this add operation is obtained as v^{rh} (Line 6). 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 – 13 in Algorithm 1). This broadcast has two parameters, the user-specified parameter e and the parameter v^{rh} prepared in the local processing.

For a remove operation $rmv(e)$, the initiating replica p_{ini} first checks whether this element is actually in $\mathcal{S}$, and then it locally increases the rh-vec $t[p_{ini}]$ to record this remove operation (Line 19 in Algorithm 1). The remove history of this operation is prepared in v^{rh} for the broadcast (Line 17). The user-specified parameter e and locally prepared parameter v^{rh} are broadcast to remote replicas on behalf of the operation $rmv(e)$. If in any dimension k, the local rh-vec element $t[k]$ is older than the vector element $v^{rh}[k]$ from the broadcast, we remove e from E, since there are unseen remove operations (Line 22–24). Then the local rh-vec $t[1..n]$ is updated to the pairwise maximum of v^{rh} and t, and this update is recorded in the payload T (Line 25 – 26).

4. The Algorithm 1 contains the RWF-Set Algorithm, with some detailed extensions like more parameters/steps.

3.2.3 Conflict Resolution for RWF-Set

To resolve the conflict between concurrent operations, we first need to handle the anomaly caused by the fact that the remove operation can arrive at the remote replica arbitrarily late, since we do not require the communication channel provide causal message delivery [13]. This means that when an $add(e)$ operation arrives at p_i, the $rmv(e)$ operations visible to it may have not arrived yet. This means that the phase of p_i^{cur} may precede the phase of $add(e)$. However, since all the $rmv(e)$ do not need additional parameters, and the rh-vec v^{rh} of $add(e)$ encodes all the visible $rmv(e)$, we can do these missing $rmv(e)$ operations first (Line 9 in Algorithm 1), update the remove history of p_i^{cur}, and then do the $add(e)$ operation.

We 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 v^{rh} of the $add(e)$ operation (Line 8 in Algorithm 1). The rh-vec on remote replica p_j is recorded in its local payload T, denoted as t. With the supplement of missing $rmv(e)$ operations, t has been updated by v^{rh}. We now have $v^{rh} \leq t$. Given this fact, we have two cases left to handle:

- $v^{rh} = t$. This means that $add(e)$ and p_j^{cur} have seen the same set of remove operations. There will be no conflict, and we directly add e into payload E on p_j.
- $v^{rh} < t$. This means that $\exists rmv(e) : rmv(e) \xrightarrow{vis} p_j^{cur} \wedge \neg(rmv(e) \xrightarrow{vis} add(e))$. This $rmv(e)$ either is concurrent with $add(e)$ or happens after $add(e)$. According to the remove-win strategy, the effect of $add(e)$ will be wiped off by $rmv(e)$.

Thus only when we have $v^{rh} = t$ can we successfully add element e into the payload E. Otherwise, it is to be wiped off by some rmv operation and can be safely ignored.

3.3 From RWF-Set to RWF-Skeleton

The RWF-Set can be augmented to store application-specific values. Since the conflict concerning the existence of elements is handled by the RWF-Set, the user can focus on the conflicts concerning the value of elements.

The specification of our RWF-Set is $\{e|\exists add(e) : \forall rmv(e) : rmv(e) \xrightarrow{vis} add(e)\}$. This is different from the specification of the existing Remove-Win Set [11], which is $\{e|\exists add(e) \wedge \forall rmv(e) : \exists add(e) : rmv(e) \xrightarrow{vis} add(e)\}$. The existing remove-win strategy actually records all the newest add/remove operations and decide whether the element exist afterwards, which is mostly like the add-win strategy of OR-Set [1] with different concurrent add/remove preference. This kind of strategies that record operations and decide afterwards is not suitable for handling the value of elements, which is needed to further augment the set into container CRDT design framework. Because the validity of value depends on the existence of the element, which can not be decided until all relevant add/remove operations are recorded. This increases the complexity of designing the container type CRDT. In our RWF-Set, system execution is segmented into phases by more powerful remove

operations. This helps designing the RWF for container type CRDTs.

The conflict resolution concerning values can be destructed into three basic cases. Thus the RWF-Skeleton is proposed, where three open terms are left for the user to develop stubs containing their own conflict resolution logics, as shown in Algorithm 1. With the RWF-Skeleton, the concrete design of an RWF-DT can be obtained by specifying how the values are initialized and updated via the RWF-DT APIs and plugging the conflict-resolution stubs.

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 RWF-RPQ design is presented here, and its implementation is presented in Section 4. More exemplar designs are presented in Appendix A-D in [14].

3.3.1 Remove-Win Resolution

The RWF-Skeleton has the new value-updating operation upd, which enables the user to modify the values of existing data elements. Comparing with the RWF-Set, the add operation in the RWF-Skeleton not only creates a data element, but also sets its initial value. 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 effects of (concurrent or causally visible) non-remove operations are wiped off.

The execution is still segmented into phases by rmv operations. When executed on a remote replica, each non-remove operation carries the rh-vec, 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.

3.3.2 User-specified Resolution

With the help from the RWF-Set, 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.

Add-add resolution. When two different add operations both add the same element, but setting different initial values, there will be a conflict. An open term is left in the skeleton (Line 13 in Algorithm 1) 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.

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 1). For example, for a list, the

Algorithm 1: RWF-Skeleton

```
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 = ∅, T = ∅, V = ∅
3  update add(e)
4      prepare (e)
5          pre e is not in the data collection
6          let v^rh = t s.t. (e, t) ∈ T        ▷ v^rh = 0⃗ if there is no (e, t) in T.
7          let p_ini be id of the initiator of this operation
8      effect (e, p_ini, v^rh)
9          rmv(e, v^rh)        ▷ Execute the effect part of rmv(e) using v^rh.
10         let t : (e, t) ∈ T        ▷ t = 0⃗ if there is no (e, t) in T.
11         if v^rh = t then        ▷ The remote replica and the add operation are in the same phase.
12             E := E ∪ {(e, p_ini)}
13             ⟨determine the innate value v_ini for e⟩
                   ▷ Resolve possible conflicts between concurrent adds, using p_ini to obtain the replica information.
14 update rmv(e)
15     prepare (e)
16         pre e is in the data collection
17         let v^rh = t s.t. (e, t) ∈ T        ▷ v^rh = 0⃗ if there is no (e, t) in T.
18         let p_ini be id of the initiator of this operation
19         v^rh[p_ini] := v^rh[p_ini] + 1
20     effect (e, v^rh)
21         let t : (e, t) ∈ T ▷ t = 0⃗ if there is no (e, t) in T.
22         if ∃k : t[k] < v^rh[k] then        ▷ There are unrecorded rmv operations in v^rh.
23             Remove (e, p_ini) from E if any
24             Remove (e, v_inn, v_acq) from V if any
25             let t' : ∀k : t'[k] := max(v^rh[k], t[k])
26             T := T \ {(e, t)} ∪ {(e, t')}
27 update upd(e)
28     prepare (e)
29         pre e is in the data collection
30         let v^rh = t s.t. (e, t) ∈ T        ▷ v^rh = 0⃗ if there is no (e, t) in T.
31     effect (e, v^rh)
32         rmv(e, v^rh)        ▷ Execute the effect part of rmv(e) using v^rh.
33         let t : (e, t) ∈ T        ▷ t = 0⃗ if there is no (e, t) in T.
34         if v^rh = t then        ▷ The remote replica and the add operation are in the same phase.
35             ⟨Modify the acquired value v_acq for e⟩
                   ▷ Resolve possible conflicts between concurrent upds, using p_ini to obtain the replica information if necessary.
```

user may employ an *operational transformation* algorithm to decide the results of all possible conflicting list updates ($insert$ and $delete$) [8], [15]. As for a priority queue, the value increase/decrease operations naturally commute. Thus no resolution is needed, as detailed in Appendix A of [14].

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 intentions, we divide the value of an element into the *innate value* and the *acquired value* (payload $V = (id, v_{inn}, v_{acq})$ in Line 1 in Algorithm 1). Accordingly, the innate value stores the initial value of the element brought by add operations, whose conflict have been resolved. And the acquired value stores the relative change of the actual value of the element from the innate value brought by upd operations. The result of *upd-upd resolution* related to the value of the element is stored here.

Thus, the conflict between an add and an upd operation is resolved by dividing the data value into two parts, one part for each operation. And the actual value of the element is the summary of the innate value (initial value set when added) and the acquired value (relative change that summarizes all upd operations). Such division of value is rather conceptual here, and requires further implementation by the CRDT designer.

4 RWF-DT Implementation

In this section, we explain how to use the RWF design framework in practice, with an exemplar priority queue implementation over Redis. More details of another list implementation can be found in Appendix C of [14]. Redis is a widely-used in-memory data type store. It adopts the master-slave architecture[5]. We modify Redis to work in the multi-master mode, and CRDTs are used for conflict resolution. Note that the adoption of RWF is orthogonal to that of the underlying data store, and RWF can be applied to other data type stores like Riak [16]. All the implementation can be found at the GitHub repository [17].

The implementation of an RWF-DT has the "onion" structure, and proceeds through three levels – the CRDT level, the RWF level and the DT level, as shown in Fig. 2. In the outermost level, the data type is first a CRDT. The basic template for local processing and asynchronous propagation of data updates is specified. In the middle level, the data type uses RWF for conflict resolution. Common metadata and conflict resolution logics following the RWF-Skeleton are specified. In the innermost level, definition of the specific data type and user-specified logics for conflict resolution are provided. In this section, we introduce these three levels one by one.

5. The enterprise version of Redis supports the multi-master architecture, and uses CRDT to handle conflicts. However, this version is not open source.

4.1 CRDT Level Implementation

In the outermost level, we implement the CRDT framework as a code template over Redis, as shown in Fig. 3. Operations which are common to different CRDT designs are abstracted as four macros and 2 types of tool functions, as detailed below.

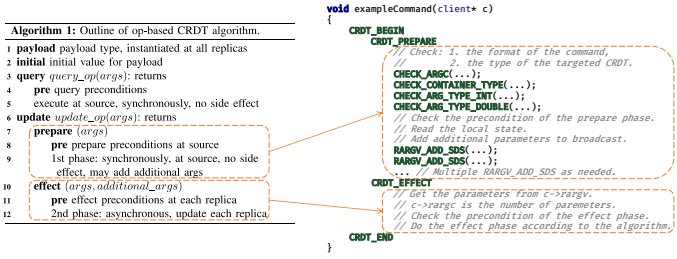

Fig. 3. Implement one CRDT operation with framework.

4.1.1 CRDT_BEGIN

The `CRDT_BEGIN` macro checks if the data store (Redis instance) works in the multi-master replication mode. If not, it is invalid to use CRDTs.

4.1.2 CRDT_PREPARE

When receiving a request, the server first needs to check its type, i.e., a *client request*, or a *server request*. In case of a client request, the server proceeds to the prepare part processing. For a server request, the server directly jumps to the effect part. In the local processing (in the prepare part) of a client request, two types of operations are common to different CRDTs.

First, the server needs to check whether the client is using the correct API the server provides. In case the API is correct, the server further checks whether the client is providing correct parameters for the API invocation. Note that the number of parameters and the type of each parameter can only be decided in the DT level. Now in the CRDT level, we provide tool functions, which encapsulate the logic for checking the number of parameters, while the actual number of parameters to be checked will be passed in as parameters later in the DT level. We also provide tool functions for parameter type checking, for widely used types such as `INT` and `DOUBLE`. The user just chooses the correct tool function and passes the correct parameter in the DT level. In case the parameter type checking function is not provided, e.g. checking functions for user defined types, the user needs to implement the checking functions themselves, following the existing tool functions.

Second, the local processing needs to prepare multiple parameters to be broadcast for the remote processing. A dynamic array is used to contain any number of parameters, and in our Redis implementation, each parameter is in the Simple Dynamic String (SDS) format defined by Redis. For any type of parameters to be broadcasted, the user only needs to provide the serialization and de-serialization functions to and from the SDS format.

4.1.3 CRDT_EFFECT

In the effect part, the server first acknowledges its reception of the server request. The concrete logics for the processing, mainly the conflict resolution logics, are filled in later in the RWF level and the DT level.

4.1.4 CRDT_END

At the end of a CRDT operation, the server acknowledges the client or server request.

4.2 RWF Level Implementation – Data Element Definition

In this section, we discuss the data element definition in the RWF level, which extracts the common characteristics of the data container type we focus on. We first discuss the innate and acquired values of concrete data. Then we discuss the metadata for the remove-win conflict resolution.

4.2.1 Innate and Acquired Values

Each RWF-DT shares the common nature of being a container of data elements. Each data element has its ID, which identifies the existence of the element. How the ID is defined, e.g. using a 64 bit string or a long integer, will be decided in the DT Level.

Each element in the container has its value. The value is initialized by *add* operations, and is then updated by the *upd* operations (together with the conflict resolution logics). However, one important common pattern is that the value of each data element has two different types of constituents, with different intentions behind (see detailed discussions in Section 3.3). One is innate value. It is often associated with initialization. The intention behind the initialization is value assignment. The new initialization should overwrite the old one. However, the concrete definition of 'old' and 'new' is user-specified since there may be concurrent initializations (later in the DT level). The other one is acquired value. The conflict resolution logic could be arbitrary and user-defined. However, it is often different from the conflict resolution logic for innate values. The classification of innate and acquired values further simplifies the development of conflict resolution logics.

4.2.2 Metadata for Conflict Resolution

The conflict resolution is based on the pre-defined remove-win strategy. Thus each element has pid and $current$. The pid is the ID of the replica which accepts the request from the client. This pid info identifies each replica. This info is leveraged to break the symmetry between concurrent (conflicting) updates.

The $current$ is the rh-vec timestamp. As in the RWF-Skeleton, the rh-vec is encoding of the remove history, which is essential to the conflict resolution following RWF. We define the struct `Rwf_element_header` containing this metadata, as shown in Fig. 4. All RWF-DT metadata structs extend the header to contain specific (innate and acquired) values.

4.2.3 Data Organization on the Server Replica

Here we use a hash map to store the metadata of a data type following RWF. This hash map can be used to get the element in the container by its key. Other data structures can be used for the RWF-DT if needed. For example, our exemplar RWF-RPQ implementation additionally uses a skiplist [18] to maintain the order of elements.

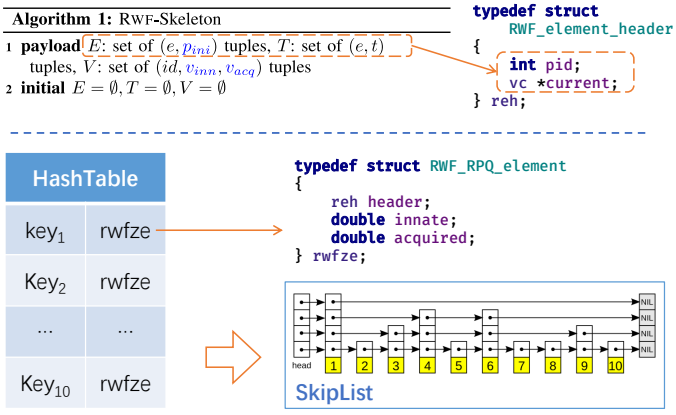

Fig. 4. The data storage implementation of RWF-RPQ.

4.3 RWF Level Implementation – Conflict Resolution

In the RWF level, the CRDT template (in Fig. 3) is further extended to include the data definitions and conflict resolution operations which are pertinent to the remove-win resolution strategy, as highlighted in Fig. 5. Here we use the *add* command as an example to illustrate the RWF level implementation.

4.3.1 Prepare

In the local processing of a client request, we first need to get the element in the hash table. Though the specific data element type may vary, getting the handler of one data element in the hash table has the generic pattern. Specifically, we first get the correct data container in the data store (we may have multiple data containers working in the data store). We then get the element by its key. We also need to get the handler of the local data structure for maintaining the structure among data elements. In the RWF level, we provide tool function `rehHTGet(···)`[6], and the user further provides parameters as required in the DT level.

Before doing the actual processing, we need to first guarantee that certain precondition holds. In the RWF level, we implement two common precondition checking functions widely used in data container types. Specifically, data container operations often need to ensure that the current element is or is not in the container. We implement two tool functions for these two types of checking. Other user-defined precondition checking can be supplemented by the user in the DT level.

In the end of the local processing, the remove history of data element needs to be updated, which is essential to the remove-win conflict resolution. The tool function for updating the remove history is implemented in the RWF level.

6. See detailed comments of the "`rehHTGet`" function in "redis-6.0.5/src/RWFramework.h" at the repository [17].

4.3.2 Effect

To conduct remove-win conflict resolution, the replica should first get the remove history (rh-vec) of the element under processing. The tool functions/macros of getting and deleting the rh-vec is provided in the RWF level. Given the rh-vec of the remote replica, the current replica needs to get the element from the hash table. This is principally the same with the `rehHTGet(···)` operation in the prepare part.

As discussed in Section 3.2.3, to cope with the late arrival of messages, the replica should check the remove history and do the missing remove operation first. Note that the remove operation not only eliminates the current element. It also needs to update the data structure after the delete operation. This update is provided in the DT level.

Before doing the actual processing, the replica needs to check whether the remove operation and the current replica are in the same phase, by comparing the rh-vecs (see Line 11 of Algorithm 1). After the checking, the actual processing can be conducted. In our example, we provide the "`addCheck`" function. Similarly for rmv and upd operations, we provide the corresponding "`rmvCheck`" and "`updCheck`".

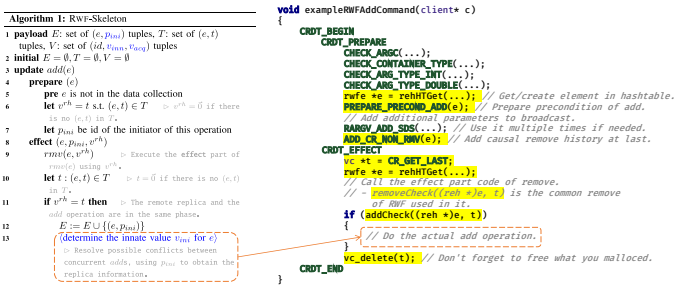

Fig. 5. Implement the add operation of a RWF CRDT using the framework.

4.4 DT Level Implementation – an RPQ Example

Here we give an example of how to implement a replicated priority queue, denoted as RWF-RPQ, using the RWF. We first "inherit" the `Rwf_element_header` to define the metadata struct of elements `rwfze`, as shown in Fig. 4. As an RWF-RPQ element, it further contains the innate and acquired values. Each data element has its ID and value (defined in `rwfze`). The key-value pairs (ID, `rwfze`) are stored in the hash table. For the priority queue, each server uses the skip list to organize the elements with their priorities. Local organization of data elements is orthogonal to the design of the RWF-DT.

The users provide parameters to the tool functions. They may also implement the concrete "`removeFunc`" for deleting an element from a data structure. Finally the users provide the logics for conflict resolution.

The development task is greatly simplified. The user only needs to adopt the template, choose the tool functions, and provide parameters to the functions. The user-defined logics are then supplemented in the indicated places.

5 Experimental Evaluation

In this section, we first present the experiment setup and design. Then we discuss the evaluation results.

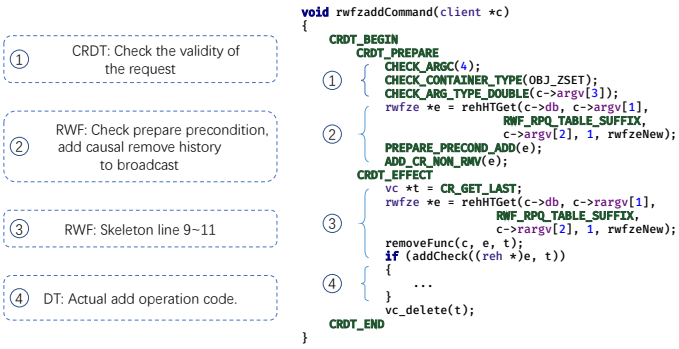

Fig. 6. The implementation of add operation of RWF-RPQ in Redis.

5.1 Experiment Setup

The experiment is conducted on a workstation with an Intel i9-9900X CPU (3.50GHz), with 10 cores and 20 threads, and 32GB RAM, running Ubuntu Desktop 16.04.6 LTS. We run all server nodes and client nodes on the workstation. Logically we divide the Redis servers into 3 data centers as shown in Fig. 7. Each data center has 3 instances of Redis. We use traffic control (TC) [19] to control the network delay among Redis instances. The default inter-data center communication delay follows $\mathcal{N}(50, 10)$[7], while the default intra-data center delay follows $\mathcal{N}(10, 2)$ (the time unit is ms). We use this set of network delay based on our experience.

The clients obtain when and what operations to issue to the servers from the workload module. This module generates workloads of different patterns. The clients record 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, to obtain current status of the CRDT. 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 CRDT.

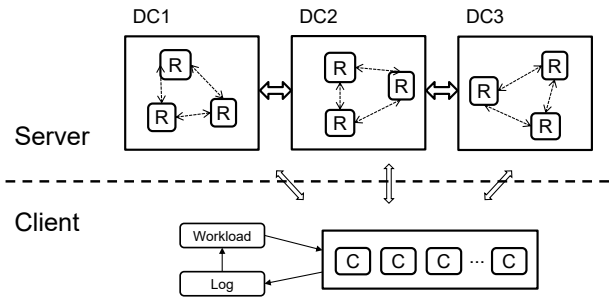

Fig. 7. Experiment setup.

5.2 Experiment Design

We design replicated priority queue and replicated list, using both the existing remove-win strategy [11] and our RWF design framework (namely the Remove-Win RPQ, the RWF-RPQ, the Remove-Win List and the RWF-List). The

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

design and implementation of the data types used in the experiments are all available online[8].

The key space for elements in the RPQ has size 200,000. The workload module randomly chooses elements to be added from all possible ones. The inc and rmv operations are conducted on random elements in the RPQ. The initial values of elements are randomly chosen from integers ranging from 0 to 100. The value increased is randomly chosen from -50 to 50.

Because the key space of RPQ in our experiment is relatively large, the probability of generating conflicting operation pairs containing add on the same element is low, as we randomly choose elements from the key space for add. We intentionally create such conflict operation pairs to evaluate the performance of an RPQ. 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 have 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, as they are conducted only on the elements that are already in the RPQ. All workloads we consider have 59%–89% operations which are inc or rmv.

The replicated lists are targeted at strings of text chars in collaborative editing scenarios. We use $(clientID, num)$ pairs as the keys of the elements in lists. We generate a new key for each add operation, and all *undo* and *redo* operations are translated into add and rmv operations. To exercise the conflict resolution strategies, 50% add operations will add previously removed elements, and the rest of add operations will add new elements. There are 6 properties for elements in the list: font(0-9), size(0-99), color(24 bits), bold(Y/N), italic(Y/N), underline(Y/N). The upd operation randomly chooses one property to update. Both the initial properties and the upd operation parameters are chosen at random. The upd and rmv operations are conducted on random elements which are currently in the list. We do not need to intentionally create conflicting operations for lists, as the probability of conflict is fairly high.

Since the CRDTs serve operations instantly by design, they have statistically the same performance in terms of query / update delay. However, there is the intrinsic trade-off 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 obtain the performance in response delay. As for the priority queue, we measure the difference between the return value of get_max and the real max value. The read-time 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 order to decide the status of the priority queue and calculate the correct max values. As for the list, we also use the real-time order on the client side to obtain the linearized list. We measure the edit distance between the list on the server and the list linearized on the client side.

8. See detailed discussions on the design in Appendix A-D of [14]. The source codes are also available in the repository [17].

TABLE 1
Data inconsistency on average. '*r*' means Remove-Win CRDT, and '*rwf*' means RWF-DT. 'upd-*dom*' stands for the upd-dominant pattern, and 'a/r-*dom*' stands for the add/rmv-dominant pattern.

	RPQ (Fig.8)		List-local (Fig.9)		List-replica (Fig.10)	
	r	rwf	r	rwf	r	rwf
upd-dom	14.8	4.0	449.8	301.0	7.8	7.5
a/r-dom	31.4	38.4	15416.1	11725.2	12.4	14.8

We further measure the edit distance between lists from different servers. Also we record the metadata overhead for resolving conflicts by the CRDTs under evaluation. The metadata overhead is averaged among all elements in the data container.

We use two types of workload patterns for both RPQs and Lists. First, we have the *add-rmv dominant* pattern where 41% operations are add, 39% operations are rmv and 20% operations are upd. Second, we have the *upd dominant* pattern where 80% operations are $update$, 11% operations are add and 9% operations are rmv[9]. We generate 4,000,000 operations in total for RPQs, 10,000 operations per second. As for lists, the number of operations generated is 400,000, 1000 operations per second.

5.3 Evaluation Results

We list the average performance in terms of data inconsistency of all data types in Table 1.

Then we discuss the evaluation results for the priority queues and lists in detail. Please note that, more evaluation results and the corresponding discussions are provided in Appendix E of [14], due to the limit of space.

5.3.1 Replicated Priority Queue

We first compare the return value of get_max from server, and the max value of the centrally linearized queue. As shown in Fig. 8, the difference vibrates mostly between -100 and 100. This is relatively small, considering the increase value we generate are chosen randomly between -50 and 50. According to evaluation results in Fig. 8 and Table 1, two RPQs act similarly considering the read max difference. The add/rmv-dominant workload pattern causes more differences. This is mainly because, in the add/rmv-dominant workload, data items enter and leave the queue more frequently, while in the upd-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[10].

As for the metadata overhead of two RPQs, it slowly increases as more operations are executed. We do not have garbage collection for the removed elements, thus needing to store their tombstones. Such removed elements require

9. We make the add operations appear slightly more than rmv to prevent the RPQ from being often empty.

10. We also compare the difference between two queues. The results are principally the same with those by comparing the replicated queue and the linearized queue. The results are shown in Appendix E in [14].

more storage as more rmv are executed. The metadata overhead is higher in the add/rmv-dominant pattern, because the RPQ needs to store more conflict resolution data for add/rmv operations than for inc operations. The RWF-RPQ has less metadata overhead than the Remove-Win RPQ, mainly because the latter needs more space to guarantee the causal delivery of messages.

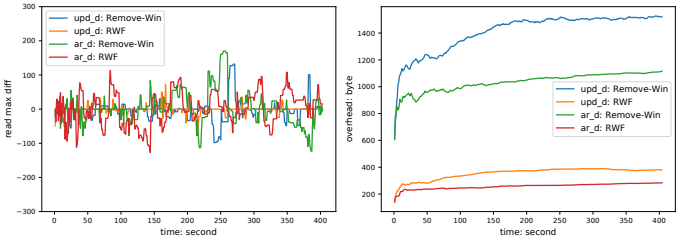

Fig. 8. The performance of RPQs, comparing max value read from the server with the max value of local linearized queue.

5.3.2 Replicated List

We first compare the list on the replicas with the list linearized on the client side. The results are shown in Fig. 9. The edit distance increases as more operations are executed. This is because the CRDTs only guarantee eventual convergence. The replica is not guaranteed to be the same with (or similar to) the linearized one. The edit distance of the upd-dominant pattern is relatively small. This is because upd operation does not affect the order of elements. Less add/rmv operations mean that the server will execute add/rmv in a more sequential manner, and need less conflict resolution.

We then compare the lists on different servers at the same time instant. As shown in Fig. 10 and Table 1, both Remove-Win List and RWF-List perform well. The distances of two lists are mostly within 50, and two lists quickly converge. The distance of the upd-dominant pattern is slightly small, as shown in Table 1. This is also because less add/rmv operations induce less divergence between the replicas.

As for the metadata cost, the overhead slowly increases as we need to store the tombstone of the removed elements. The overhead is much lower in the experiment of comparison between replicas (Fig 10), because here we make 50% add to add previously removed elements, causing their tombstones to be efficiently reused. The metadata overhead is much lower in the upd-dominant pattern. Similar to the RPQ case, conflict resolution data needed for upd is much less for that of add/rmv operations. Moreover, the Remove-Win List needs to maintain causal message delivery, which causes higher metadata overhead.

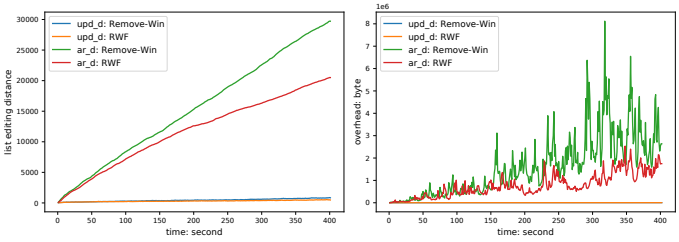

Fig. 9. The performance of lists, comparing the edit distance between the list read from the server and the local linearized list.

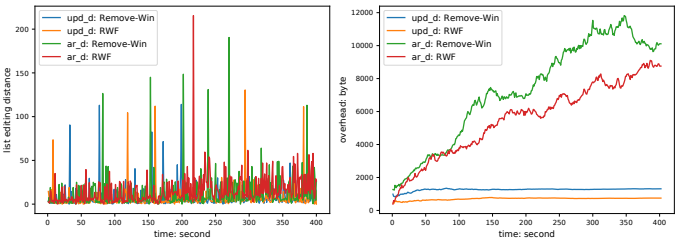

Fig. 10. The performance of lists, comparing the edit distance between two lists read from different servers at the same time.

6 Related Work

Conflict resolution is the essential issue in the design of CRDTs. For data container types, the dual add-win and remove-win strategies are intuitive and widely used. The Add-Win Set proposed in [1] lets each rmv operation record all add operations it has seen. The effect of a rmv operation is limited to the add operations it has seen, which makes the add operation win over the concurrent rmv. The design of the Remove-Win Set proposed in [11] 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 win over the concurrent add. In existing add-win and remove-win sets, all operations are recorded in the execution and a total order among all operations is derived to interpret the state of each replica. In our RWF design framework, non-remove operations which are concurrent with a remove operation are pruned from the execution under concern. Thus no conflict will occur concerning remove operations. The remove-win strategy used in RWF further utilizes the potential of the remove-win strategy, thus better supporting a design framework. Experiments show that the semantics of RWF-DTs are statistically similar to CRDTs using the existing remove-win strategy.

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 [20]. Multiple improved designs following WOOT were proposed, including WOOTO and WOOTH [21]. In the area of computational CRDTs, for a class of CRDTs whose state is the result of a computation over the executed updates, a brief study is presented in [22] 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 [23]. Though existing derivations of CRDT designs are mainly driven by the application scenarios, our RWF design framework focuses on the data type itself. RWF focuses on the widely-used data collection type and can be used in a variety of application scenarios.

7 Conclusion

In this work, we propose the RWF design framework to guide the design of CRDTs. RWF leverages the remove-win strategy to resolve conflicting updates pertinent to remove operations, and provides generic design for a variety of data

container types. Exemplar implementations over the Redis data type store show the effectiveness of RWF. Performance measurements show the efficiency of CRDT implementations following RWF.

In our future work, we will design more CRDTs using RWF. We will also formally specify and verify the designs and implementations following RWF. More comprehensive experimental evaluations under various workloads are also necessary.

REFERENCES

[1] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "A comprehensive study of Convergent and Commutative Replicated Data Types," Inria – Centre Paris-Rocquencourt ; INRIA, Research Report RR-7506, Jan. 2011. [Online]. Available: https://hal.inria.fr/inria-00555588

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

[3] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Stronger semantics for low-latency geo-replicated storage," in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, ser. nsdi'13. Berkeley, CA, USA: USENIX Association, 2013, pp. 313–328. [Online]. Available: http://dl.acm.org/citation.cfm?id=2482626.2482657

[4] L. Gondelman, S. O. Gregersen, A. Nieto, A. Timany, and L. Birkedal, "Distributed causal memory: Modular specification and verification in higher-order distributed separation logic," *Proc. ACM Program. Lang.*, vol. 5, no. POPL, Jan. 2021. [Online]. Available: https://doi.org/10.1145/3434323

[5] E. A. Brewer, "Towards robust distributed systems (abstract)," in *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC'00. New York, NY, USA: ACM, 2000, pp. 7–. [Online]. Available: http://doi.acm.org/10.1145/343477.343502

[6] S. Gilbert and N. A. Lynch, "Perspectives on the cap theorem," *Computer*, vol. 45, no. 2, pp. 30–36, 2012.

[7] 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*, ser. SSS'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 386–400. [Online]. Available: http://dl.acm.org/citation.cfm?id=2050613.2050642

[8] 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*, 2018, pp. 12:1–12:16. [Online]. Available: https://doi.org/10.4230/LIPIcs.OPODIS.2018.12

[9] 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.

[10] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Coordination avoidance in database systems," *Proc. VLDB Endow.*, vol. 8, no. 3, p. 185–196, Nov. 2014. [Online]. Available: https://doi.org/10.14778/2735508.2735509

[11] M. Zawirski, "Dependable Eventual Consistency with Replicated Data Types," Theses, Universite Pierre et Marie Curie, Jan. 2015. [Online]. Available: https://tel.archives-ouvertes.fr/tel-01248051

[12] F. Mattern, "Virtual time and global states of distributed systems," in *Proc. International Workshop on Parallel and Distributed Algorithms*, Holland, 1989, pp. 215–226.

[13] K. Birman, A. Schiper, and P. Stephenson, "Lightweight causal and atomic group multicast," *ACM Trans. Comput. Syst.*, vol. 9, no. 3, pp. 272–314, Aug. 1991. [Online]. Available: http://doi.acm.org/10.1145/128738.128742

[14] Remove-win: a design framework for conflict-free replicated data types. 01-15-2021. [Online]. Available: https://github.com/anonymous2159-sys/CRDT-Redis/blob/master/document/rwf-tr.pdf

[15] 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*, ser. PODC '16. New York, NY, USA: ACM, 2016, pp. 259–268. [Online]. Available: http://doi.acm.org/10.1145/2933057.2933090

[16] "Riak distributed database," https://riak.com/, 2019.

[17] "Conflict-free replicated data type implementations based on redis," https://github.com/anonymous2159-sys/CRDT-Redis.

[18] W. Pugh, "Skip lists: A probabilistic alternative to balanced trees," *Commun. ACM*, vol. 33, no. 6, p. 668–676, Jun. 1990. [Online]. Available: https://doi.org/10.1145/78973.78977

[19] M. A. Brown, "Traffic control howto," http://tldp.org/HOWTO/Traffic-Control-HOWTO/index.html, 2020, accessed: 09-30-2020.

[20] 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*, ser. CSCW '06. New York, NY, USA: ACM, 2006, pp. 259–268. [Online]. Available: http://doi.acm.org/10.1145/1180875.1180916

[21] 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*, ser. DocEng '11. New York, NY, USA: ACM, 2011, pp. 103–112. [Online]. Available: http://doi.acm.org/10.1145/2034691.2034717

[22] 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*, ser. PaPoC '15. New York, NY, USA: ACM, 2015, pp. 1:1–1:4. [Online]. Available: http://doi.acm.org/10.1145/2745947.2745948

[23] G. M. Cabrita, "Non-uniform replication for replicated objects," *Master thesis, Universidade Nova de Lisboa*, 2017.

APPENDIX A
RWF-RPQ DESIGN

We design and implement a Replicated Priority Queue (RPQ), under the guidance of the Remove-Win Framework. The RPQ 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 RPQ 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 RPQ supports the query operations below to better illustrate our RPQ design:

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

Following the RWF-Skeleton, design of the RPQ is obtained by instantiating the RWF-Skeleton and develop RPQ-specific stubs, as detailed below.

A.1 RPQ Design

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

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 RPQ 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 adds are overwritten. In our exemplar design, we can simply specify "largest replica id wins" (assuming that the ids 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 RPQ design is presented in Algorithm 2 and Algorithm 3.

Algorithm 2: RWF-RPQ (payloads and queries)

```
1  payload E: set of (e, p_ini) tuples, T: set of (e, t)
     tuples, V: set of (e, v_inn, v_acq) tuples
2  initial E = ∅, T = ∅, V = ∅
3  query empty(): boolean
4      return E ≠ ∅
5  query lookup(e): boolean
6      return ∃p_ini : (e, p_ini) ∈ E
7  query get_pri(e): integer
8      pre lookup(e)
9      let x, δ : (e, x, δ) ∈ V
10     return x + δ
11 query get_max(): id, integer
12     pre ¬empty()
13     let e : lookup(e) ∧ ∀o : lookup(o) ∧ get_pri(o) ≤
         get_pri(e)
14     return e, get_pri(e)
```

A.2 Illustrating Examples

We use three examples to better illustrate the design of our RPQ. 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 shows that we don't need causal delivery for phases because we redo rmv operations in non-remove operations using the rh-vec they carry.

In the remove-win example in Figure 11, 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 rh-vec of e in T is set to $v_1 = [0, 1]$, which is larger than the rh-vecs 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 rh-vec v_1.

In the example of conflict resolution among non-remove operations in Figure 14, 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 rh-vec $v_0 = [0, 0]$. 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 rh-vec 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 Figure 13, we show the reason why we don't need causal delivery. The rmv initiated by p_0 is visible to the inc initiated by p_2, not directly but via the add operation initiated by p_1. The rh-vec is initially $v_0 = [0, 0, 0]$. The rmv on p_0 updates the rh-vec 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, and the missing rmv operation is redone at p_2, updating the rh-vec of p_2 to v_1. Thus when the rmv operations arrives late at p_2 (bringing with it the rh-vec v_1), it will be safely ignored since p_2 has already obtained the rh-vec v_1 before. Without the redo of the rmv triggered by add that update the rh-vec on p_2, 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.

Algorithm 3: RWF-RPQ (updates)

```
update add(e, x)
    prepare (e, x)
        pre ¬lookup(e)
        let v^rh = t s.t. (e, t) ∈ T        ▷ v^rh = 0⃗ if there is no (e, t) in T.
        let p_ini be id of the initiator of this operation
    effect (e, x, p_ini, v^rh)
        rmv(e, v^rh)        ▷ Execute the effect part of rmv(e) using v^rh.
        let pid : (e, pid) ∈ E        ▷ pid = −1 if there is no (e, pid) in E.
        let t : (e, t) ∈ T        ▷ t = 0⃗ if there is no (e, t) in T.
        if v^rh = t ∧ p_ini > pid then ▷ Larger replica id wins.
            E := E \ {(e, pid)} ∪ {(e, p_ini)}
            let x′, δ : (e, x′, δ) ∈ V        ▷ x′ = 0 and δ = 0 if there is no (e, x′, δ) in V.
            V := V \ {(e, x′, δ)} ∪ {(e, x, δ)}
update inc(e, i)        ▷ i ∈ ℤ, i < 0 means 'decrease'.
    prepare (e, i)
        pre lookup(e)
        let v^rh = t s.t. (e, t) ∈ T        ▷ v^rh = 0⃗ if there is no (e, t) in T.
    effect (e, i, v^rh)
        rmv(e, v^rh)        ▷ The same as the effect part of add.
        let t : (e, t) ∈ T        ▷ t = 0⃗ if there is no (e, t) in T.
        if v^rh = t then
            let x, δ : (e, x, δ) ∈ V        ▷ x = 0 and δ = 0 if there is no (e, x, δ) in V.
            V := V \ {(e, x, δ)} ∪ {(e, x, δ + i)}
update rmv(e)
    prepare (e)
        pre lookup(e)
        let v^rh = t s.t. (e, t) ∈ T        ▷ v^rh = 0⃗ if there is no (e, t) in T.
        let p_ini be id of the initiator of this operation
        v^rh[p_ini] := v^rh[p_ini] + 1
    effect (e, v^rh)
        let t : (e, t) ∈ T ▷ t = 0⃗ if there is no (e, t) in T.
        if ∃k : t[k] < v^rh[k] then
            let pid : (e, pid) ∈ E        ▷ pid = −1 if there is no (e, pid) in E.
            E := E \ {(e, pid)}
            let x, δ : (e, x, δ) ∈ V        ▷ x = 0 and δ = 0 if there is no (e, x, δ) in V.
            V := V \ {(e, x, δ)}
            let t′ : ∀k : t′[k] := max(v^rh[k], t[k])
            T := T \ {(e, t)} ∪ {(e, t′)}
```

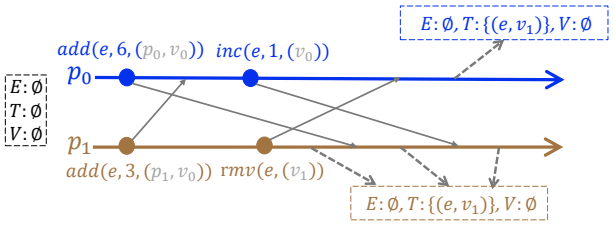

Fig. 11. An example showing how an rmv wins, where $v_0 = [0, 0]$, $v_1 = [0, 1]$.

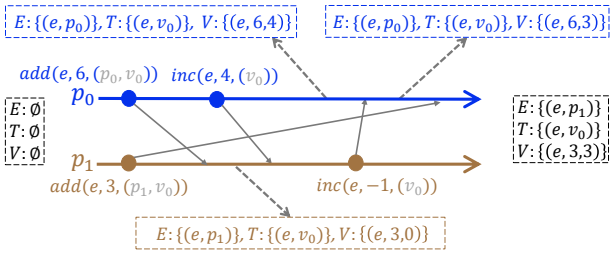

Fig. 12. Conflict resolution among non-remove operations, where $v_0 = [0, 0]$.

APPENDIX B
REMOVE-WIN RPQ DESIGN

Here we try to design a Remove-Win RPQ without our RWF-Skeleton.

Note that the classic remove-win doesn't mean that the remove operation simply kills all other concurrent non-remove operations. Sometimes these concurrent non-remove operations will still take effect. See the example in figure 14. There are no communication between two processes. Therefor r_1 and a_1 are concurrent with r_2 and a_2. Although both a_1 and a_2 has a concurrent remove operation (r_2 and r_1) that may kill them due to the remove-win semantics, combined they win over the remove operations. Then the element is in the RPQ rather than removed. This is reasonable, because if you linearize the causal order of these four operations, the last operation will always be an add.

The detailed design is shown in Algorithm 4. Here we assume that causal delivery is provided by the underlying network. We can use $now()$ function to get the vector clock of the current operation. We denote $v_1 \parallel v_2$ as two vector clocks v_1 and v_2 are parallel, and $v_1 < v_2$ means v_1 is less than v_2. Note that this vector clock indicates the visible relation between operations: $op_1 \xrightarrow{vis} op_2 \iff op_1.vec < op_2.vec$.

We first discuss the existence of elements. Firstly the Remove-Win specification: $e \in RPQ \iff \exists add(e) \wedge \forall rmv(e). \exists add(e). rmv(e) \xrightarrow{vis} add(e)$. We notice that to decide if an element e is in the RPQ, we only need to store all the $add(e)$ and $rmv(e)$ operations that may be effective, which means there is no $add(e)$ or $rmv(e)$ operation that happen after them. Then we decide if the element e is in the RPQ strictly by the Remove-Win specification.

Then the value of elements, we resolve the conflicts of initial value brought by concurrent add operations with the process id. Here we let the add with larger process id win, and yet we store all the value records of these add operations. As for the inc operations, we let it only increase

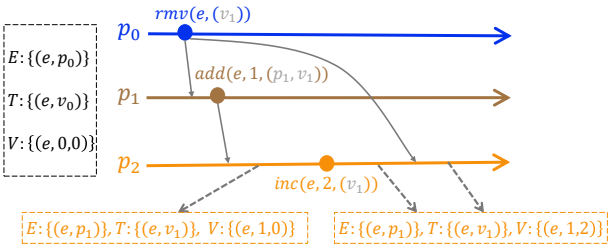

Fig. 13. No need for causal delivery for phase, where $v_0 = [0,0,0]$, $v_1 = [1,0,0]$.

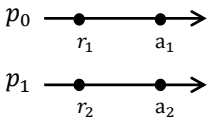

Fig. 14. The case of remove-win.

the value records brought by the add operations that are visible to it. We use the vector clock to identify this. Because of causal delivery, the inc operation will be correctly applied at all replicas.

Appendix C
RWF-List Design

We design and implement a Replicated List under the guidance of the Remove-Win Framework. The List is a container of elements of the form $e = (id, content, properties)$. Elements are totally ordered. An element has its unique ID, the content (letter, word, or paragraph...), and properties (font, size, color, shape...). The content of one element will not be changed in co-editing scenario. Clients can modify the list by the following update operations:

- $add(e, e_p, P)$: add the element e after e_p with initial properties P.
- $upd(e, p)$: update the element e with some new property p.
- $rmv(e)$: remove the element e.

Additionally, we assume that the List supports the query operations below:

- $empty()$: returns $true$ if the List is empty.
- $lookup(e)$: returns $true$ if e is in the List.
- $properties(e)$: returns the properties of e.
- $read\_list()$: returns the list of elements with its content and properties, totally ordered.

Following the RWF-Skeleton, design of the RWF-List is obtained by instantiating the RWF-Skeleton and develop List-specific stubs.

The detailed RWF-List design is presented in Algorithm 5 and Algorithm 6.

Here we use the Logoot ID to identify the position of the element. The Logoot ID is unique, totally ordered and dense. Hence the list is transformed into the ordered set whose elements are ordered by the Logoot ID. By using the RWF-Skeleton, the existence of elements is properly handled. The order of elements is identified by Logoot IDs. Now we only need to care about the consistence of values of elements.

Algorithm 4: Remove-Win RPQ

```
1  payload A: set of (e,t,id,x,δ) tuples, R: set of (e,t) tuples
2  initial A = ∅, R = ∅
3  query empty(): boolean
4      return ∀e : (e,t,id,x,δ) ∈ A → ¬lookup(e)
5  query lookup(e): boolean
6      return ∃t : (e,t,id,x,δ) ∈ A ∧ ∄t' : (e,t') ∈ R
7  query get_pri(e): integer
8      pre lookup(e)
9      let id,x,δ : ∀(e,t,id',x',δ') ∈ A : id' < id
10     return x + δ
11 query get_max(): id, integer
12     pre ¬empty()
13     let e : lookup(e) ∧ ∀o : lookup(o) ∧ get_pri(o) ≤
         get_pri(e)
14     return e, get_pri(e)
15 update add(e,x)
16     prepare (e,x)
17         pre ¬lookup(e)
18         let t = now()
19         let id: the id of the process
20     effect (e,x,t,id)
21         A := A ∪ {(e,x,t,id,0)}
22         foreach (e,x,t,id,δ) ∈ A do
23             if t' < t then A := A \ {(e,x,t,id,δ)}
24         end
25         foreach (e,t') ∈ R do
26             if t' < t then R := R \ {(e,t')}
27         end
28 update rmv(e)
29     prepare (e)
30         pre lookup(e)
31         let t = now()
32     effect (e,t)
33         R := R ∪ {(e,t)}
34         foreach (e,x,t,id,δ) ∈ A do
35             if t' < t then A := A \ {(e,x,t,id,δ)}
36         end
37 update inc(e,i)
38     prepare (e,i)
39         pre lookup(e)
40         let t = now()
41     effect (e,i,t)
42         foreach (e,x,t',id,δ) ∈ A do
43             if t' < t then A :=
               A \ {(e,x,t',id,δ)} ∪ {(e,x,t',id,δ+i)}
44         end
```

Moreover, the innate value of elements brought by add operations are handled by the RWF-Skeleton. The add-add conflict resolution is done by using the pid of the initiating process. As for the add-upd conflict, we let the $update$ operations win over add operations if they are in the same phase. As for the upd-upd conflict, we attach a totally-ordered lamport-clock generated by $now()$ function to each $update$ operation. Then we adopt the last-write-win policy for conflicting $update$ operations in the same phase.

Algorithm 5: RWF-List (payloads and queries)

payload E: set of (e, p_{ini}, pos) tuples, T: set of (e, t) tuples, V: set of (e, I, A) tuples ▷ pos: Logoot ID, I: set of $property_{inn}$, A: set of $(property_{acc}, t)$ tuples
initial $E = \emptyset, T = \emptyset, V = \emptyset$
query $empty()$: boolean
  return $\nexists e : lookup(e)$
query $lookup(e)$: boolean
  return $\exists p_{ini} : (e, p_{ini}, pos) \in E \wedge p_{ini} \neq -1 \wedge \nexists A : (e, \emptyset, A) \in V$
query $properties(e)$: properties
  pre $lookup(e)$
  let $I, A : (e, I, A) \in V$
  return for each kind of property, the value in A with max t, or the value in I if no such property in A
query $read\_list()$: list
  pre $\neg empty()$
  let $R = (e, pos) | (e, p_{ini}, pos) \in E \wedge lookup(e)$
  return the list of e in R, sorted by pos

APPENDIX D
REMOVE-WIN LIST DESIGN

Here we try to design a Remove-Win List without our RWF-Skeleton. The detailed design is shown in Algorithm 7. The same as Remove-Win RPQ, here we assume that causal delivery is provided by the underlying network. And we can use $now()$ function to get the vector clock of the current operation. Like RWF-List, we use Logoot ID to identify the position of an element in the list. Then the consistency of element order is guaranteed.

We use the same technique of the Remove-Win RPQ to ensure the consistency of the existence of elements and the remove-win semantics, which is to store the effective add and rmv operations, and then decide if the element e is in the list by the Remove-Win specification.

Then the consistency of the element value. We store all the initial value brought by add operations that are still effective, together with the process id of the replica that generated the add, as value records. The $update$ operations, like it is in Remove-Win RPQ, will update all the value records of add operations that is visible to it. The $update$ operations adopt a last-write-win strategy if two $update$ want to update the same value record simultaneously. Finally, the value record that is read by clients is that with the highest process id.

Algorithm 6: RWF-List (updates)

update $add(e, e_p, P)$ ▷ add e after e_p, or at the beginning if $e_p = null$, P: initial properties
  prepare (e, e_p, P)
    pre $\neg lookup(e) \wedge (lookup(e_p) \vee e_p \; is \; null)$
    let $v^{rh} = t$ s.t. $(e, t) \in T$ ▷ $v^{rh} = \vec{0}$ if there is no (e, t) in T.
    let p_{ini} be id of the initiator of this operation
    let $pos : (e, p_{ini}, pos) \in E$ if there is such tuple, or otherwise the proper Logoot ID after e_p and before the next element of e_p
  effect $(e, pos, P, p_{ini}, v^{rh})$
    $rmv(e, v^{rh})$ ▷ Execute the **effect** part of $rmv(e)$ using v^{rh}.
    let $pid : (e, pid, pos) \in E$ ▷ $pid = -1$ if there is no (e, pid, pos) in E.
    let $t : (e, t) \in T$ ▷ $t = \vec{0}$ if there is no (e, t) in T.
    if $v^{rh} = t \; \wedge \; p_{ini} > pid$ **then** ▷ Larger replica id wins.
      $E := E \setminus \{(e, pid, pos)\} \cup \{(e, p_{ini}, pos)\}$
      let $(e, I, A) \in V$ ▷ $I = \emptyset$ and $A = \emptyset$ if there is no (e, I, A) in V.
      $V := V \setminus \{(e, I, A)\} \cup \{(e, P, A)\}$
update $upd(e, p)$ ▷ p is some property
  prepare (e, p)
    pre $lookup(e)$
    let $v^{rh} = t$ s.t. $(e, t) \in T$ ▷ $v^{rh} = \vec{0}$ if there is no (e, t) in T.
    let $t_u = now()$ ▷ lamport clock
  effect (e, p, t_u, v^{rh})
    $rmv(e, v^{rh})$ ▷ The same as the effect part of add.
    let $t : (e, t) \in T$ ▷ $t = \vec{0}$ if there is no (e, t) in T.
    if $v^{rh} = t$ **then**
      let $(e, I, A) \in V$ ▷ $I = \emptyset$ and $A = \emptyset$ if there is no (e, I, A) in V.
      $V := V \setminus \{(e, I, A)\} \cup \{(e, I, A \cup (p, t_u))\}$
update $rmv(e)$
  prepare (e)
    pre $lookup(e)$
    let $v^{rh} = t$ s.t. $(e, t) \in T$ ▷ $v^{rh} = \vec{0}$ if there is no (e, t) in T.
    let p_{ini} be id of the initiator of this operation
    $v^{rh}[p_{ini}] := v^{rh}[p_{ini}] + 1$
  effect (e, v^{rh})
    let $t : (e, t) \in T$ ▷ $t = \vec{0}$ if there is no (e, t) in T.
    if $\exists k : t[k] < v^{rh}[k]$ **then**
      let $pid : (e, pid, pos) \in E$ ▷ $pid = -1$ if there is no (e, pid, pos) in E.
      $E := E \setminus \{(e, pid, pos)\} \cup \{(e, -1, pos)\}$
      let $(e, I, A) \in V$ ▷ $I = \emptyset$ and $A = \emptyset$ if there is no (e, I, A) in V.
      $V := V \setminus \{(e, I, A)\}$
      let $t' : \forall k : t'[k] := \max(v^{rh}[k], t[k])$
      $T := T \setminus \{(e, t)\} \cup \{(e, t')\}$

Algorithm 7: Remove-Win List

payload L: set of (e, pos) tuples, A: set of (e, t, id, P) tuples, R: set of (e, t) tuples ▷ P: set of (property, t, id) tuples
initial $L = \emptyset, A = \emptyset, R = \emptyset$
query $empty()$: boolean
  return $\nexists e : lookup(e)$
query $lookup(e)$: boolean
  return $\exists t : (e, t, id, P) \in A \wedge \nexists t' : (e, t') \in R$
query $properties(e)$: properties
  pre $lookup(e)$
  let $id, P : \forall (e, t, id', P') \in A : id' < id$
  return properties in P
query $read\_list()$: list
  pre $\neg empty()$
  let $R = (e, pos) | (e, pos) \in L \wedge lookup(e)$
  return the list of e in R, sorted by pos
update $add(e, e_p, P)$ ▷ add e after e_p, or at the beginning if $e_p = null$, P: initial properties
  prepare (e, e_p, P)
    pre $\neg lookup(e) \wedge (lookup(e_p) \vee e_p \; is \; null)$
    let $t = now()$
    let id: the id of the process
    let $pos : (e, pos) \in L$ if there is such tuple in L, or otherwise the proper logoot ID after e_p and before the next element of e_p
  effect (e, P, t, id, pos)
    $L := L \cup \{(e, pos)\}$
    $A := A \cup \{(e, t, id, P \times \{(t, id)\})\}$
    foreach $(e, t, id, P) \in A$ **do**
      if $t' < t$ **then** $A := A \setminus \{(e, t, id, P)\}$
    end
    foreach $(e, t') \in R$ **do**
      if $t' < t$ **then** $R := R \setminus \{(e, t')\}$
    end
update $rmv(e)$
  prepare (e)
    pre $lookup(e)$
    let $t = now()$
  effect (e, t)
    $R := R \cup \{(e, t)\}$
    foreach $(e, t, id, P) \in A$ **do**
      if $t' < t$ **then** $A := A \setminus \{(e, t, id, P)\}$
    end
update $upd(e, p)$ ▷ p is some property
  prepare (e, p)
    pre $lookup(e)$
    let $t = now()$
    let id: the id of the process
  effect (e, p, t, id)
    foreach $(e, t', id', P) \in A$ **do**
      if $t' < t$ **then**
        let $p', t_u, id_u : (p', t_u, id_u) \in P$ and p' is the same type of p
        if $t_u < t \vee (t_u \parallel t \wedge id_u < id)$ **then**
          $P := P \setminus \{(p', t_u, id_u)\} \cup \{(p, t, id)\}$
      end
    end

TABLE 2
Data inconsistency on average.

	RPQ-replica (Fig.15)	
	r	rwf
upd-dom	8.7	9.2
a/r-dom	33.2	21.4

APPENDIX E
EXPERIMENT RESULT

In this section, we provide more evaluation results and discussions.

E.1 RPQ max difference between replicas

Here we compare the max read from two different replicas at the same time. The experiment settings are the same with the previous RPQ experiment. The statistics are shown in Table 2, and the result is shown in Fig. 15.

The results are principally the same with those by comparing the replicated queue and the linearized queue. The difference vibrates mostly between -100 and 100. And the add/rmv-dominant workload pattern causes more differences. As for metadata overhead, it slowly increases, the add/rmv-dominant pattern causes higher overhead, and the RWF-RPQ has less metadata overhead than the Remove-Win RPQ. The reasons are discussed in the previous sections.

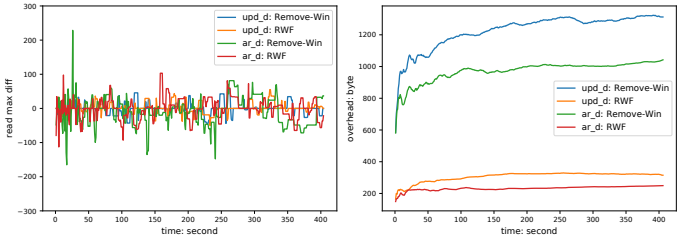

Fig. 15. The performance of RPQs. Compare max read form different servers at the same time.

E.2 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 10,000 ops/s for RPQ, and from 50 to 1,000 ops/s for list. To control the concurrency in the space dimension, we change the network delay and the number of replicas. We tune the inter-data center delay from $\mathcal{N}(20, 4)$ms to $\mathcal{N}(380, 76)$ms, and tune the intra-data center delay from $\mathcal{N}(4, 0.8)$ms to $\mathcal{N}(76, 15.2)$ms. As for the number of replicas, we increase the number of Redis instances from 1 to 5 in every data center, and fix the operation generation speed for each Redis instance.

After the discussion of the former experiment, we here focus on comparing the max value between server and local record for RPQ, and comparing the list edit distance between lists read from two replicas.

As for the data consistency, we find that the average error $\bar{x}$ of $read\_max$ for both RPQs, and the list edit distance for both lists increases linearly with the concurrency among operations, as shown in Fig. 16, 17 and 18 for RPQ, and Fig. 19, 20 and 21 for List. This is mainly because the CRDT 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 the read differences increase linearly.

As for the metadata overhead, at the end of each run of the experiment, we measure the average total metadata overhead during this run. We find that the Remove-Win CRDTs have more metadata overhead as the operation speed increases, as shown in Fig. 16 and 19. This is because the Remove-Win CRDTs require causal delivery. And as the operation speed increases, there are more causally unready operations that need more memory to deal with. And our RWF CRDTs do not need to deal with causally unready operations. They do not require causal delivery. As long as the number of operations conducted on the queue is statistically similar, the metadata overhead is also similar.

The message delay has less impact on the data consistency and the metadata overhead, as shown in Fig. 17 and 20. We think this is because the message delay has less influence on the concurrency among operations than the other two factors in our experiment setups.

The metadata overhead of our CRDTs increases as there are more replicas in Fig. 18 and 21. Not only because the concurrency among operations increases as the number of replica increases, since we fix the operation generation speed for each replica, but also the dimension of both vector clock and rh-vec get increased, as they are equal to the number of replicas on the server side. Thus the metadata overhead (for recording the vector) increases linearly as the number of replicas increases.

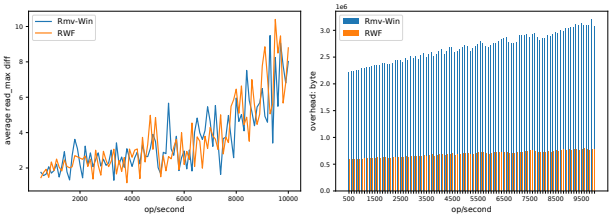

Fig. 16. The performance of RPQs over different operation speed.

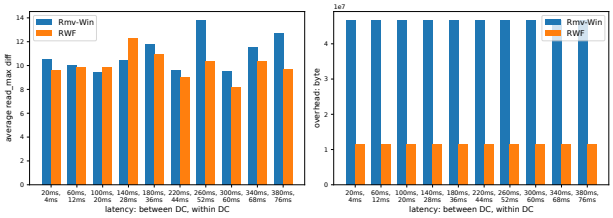

Fig. 17. The performance of RPQs over different network delay.

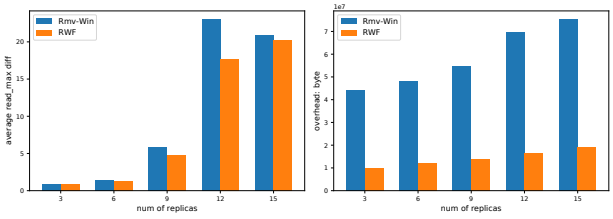

Fig. 18. The performance of RPQs over different number of replicas.

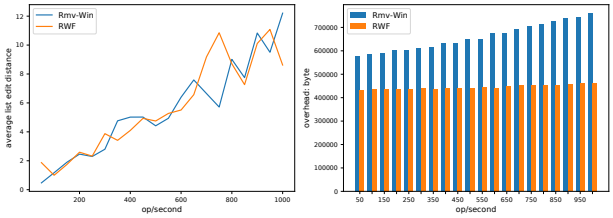

Fig. 19. The performance of Lists over different operation speed.

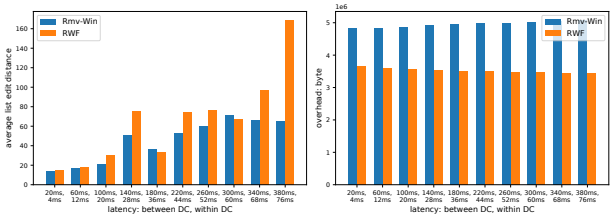

Fig. 20. The performance of Lists over different network delay.

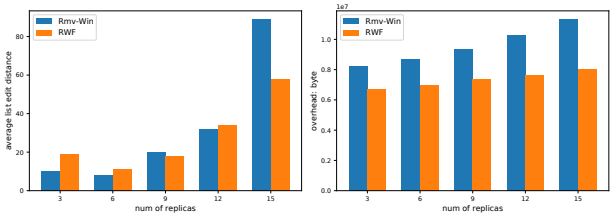

Fig. 21. The performance of Lists over different number of replicas.