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

Yuqi Zhang, Hengfeng Wei*, Yu Huang*
National Key Laboratory for Novel Software Technology, Nanjing University
Nanjing, Jiangsu Province, 210023, China
cs.yqzhang@gmail.com, {hfwei, yuhuang}@nju.edu.cn

***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**

I. 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]. 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 [4], [5]. 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 [6]. Eventually consistent replicated data types are widely used in scenarios where responsiveness is critical, e.g. in collaborative editing [7], distributed caching [8] or coordination-avoidance in databases [9]. 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].

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

* Corresponding author.

This work is supported by the National Natural Science Foundation of China (No. 61772258).

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 [10]. 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 II, we overview our design framework. In Section III, we present the generic design of RWF-DTs. Section IV presents the implementation and the performance evaluation results. Section V discusses the related work. In Section VI, we summarize our work and discuss the future work.

II. 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.

A. *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 III.

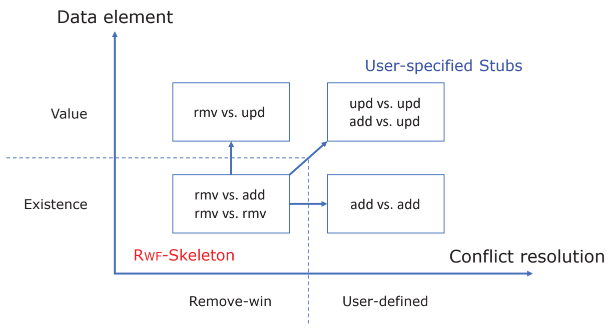

Fig. 1. Two dimensions in RWF-DT design.

B. *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 [6]. 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.

III. 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.

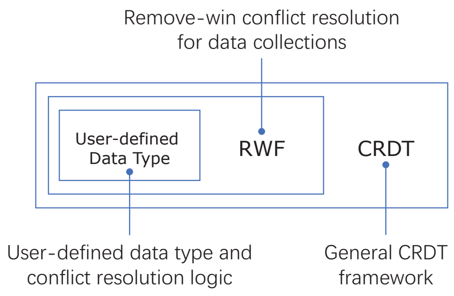
Remove-win conflict resolution for data collections
User-defined Data Type
RWF
CRDT
User-defined data type and conflict resolution logic
General CRDT framework

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

A. 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.

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.

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 $\mathcal{H}_r(a) \subset \mathcal{H}_r(b)$.

B. 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.

1) Encoding of 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.We only need to record the last remove operation initiated on p_i.

The encoding/decoding scheme we use is principally the vector clock [11]. 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: $\forall j \in [1..n] : v_i[j] = \max(t_i[j], t[j])$.

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) 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 [12]. 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.

C. 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 [10], 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 increases the complexity of designing the container type CRDT with it because you need to additionally consider value of elements. 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. The implementation of RWF-Skeleton is presented in Section IV-A. We have some exemplar designs presented in Appendix A-D in [13].

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 $E = \emptyset, T = \emptyset, V = \emptyset$
3  update $add(e)$
4      prepare $(e)$
5          pre $e$ is not in the data collection
6          let $v^{rh} = t$ s.t. $(e, t) \in T$      ▷ Default $v^{rh} = \vec{0}$.
7          let $p_{ini}$ be id of the initiator of this operation
8      effect $(e, p_{ini}, v^{rh})$
9          $rmv(e, v^{rh})$      ▷ The effect part of $rmv(e)$.
10         let $t : (e, t) \in T$      ▷ Default $t = \vec{0}$.
11         if $v^{rh} = t$ then      ▷ The remote replica and the
             $add$ operation are in the same phase.
12             $E := E \cup \{(e, p_{ini})\}$
13             ⟨determine the innate value $v_{ini}$ for $e$⟩
14 update $rmv(e)$
15     prepare $(e)$
16         pre $e$ is in the data collection
17         let $v^{rh} = t$ s.t. $(e, t) \in T$      ▷ Default $v^{rh} = \vec{0}$.
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 $t : (e, t) \in T$      ▷ Default $t = \vec{0}$.
22         if $\exists k : t[k] < v^{rh}[k]$ 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 $t' : \forall k : t'[k] := \max(v^{rh}[k], t[k])$
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 $v^{rh} = t$ s.t. $(e, t) \in T$      ▷ Default $v^{rh} = \vec{0}$.
31     effect $(e, v^{rh})$
32         $rmv(e, v^{rh})$      ▷ The effect part of $rmv(e)$.
33         let $t : (e, t) \in T$      ▷ Default $t = \vec{0}$.
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 $e$⟩
```

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.

2) User-specified Resolution: Now the user only needs to care about the conflicts of add and upd operations within the same phase. There are three different types of possible conflicts, 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. Here we suggest using the "larger replica ID wins" strategy, keeping the value whose add operation carries larger p_{ini}, the unique ID of initiating replica.

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). This is simplified because you do not need to consider the influence of add and rmv operations now.

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. 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. And the acquired value stores the relative change of the value from the innate value brought by upd operations.

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.

IV. Implementation and Experimental Evaluation

In this section, we first present our RWF-DT implementation. Then we present the experiment setup and design. Finally we discuss the evaluation results.

A. RWF-*DT Implementation*

We implement two exemplar data types over Redis: RWF-RPQ and RWF-List. 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. More details can be found in Section 4, Appendix A and Appendix C of [13].

B. *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. 3. Each data center has 3 instances of Redis. We use traffic control (TC) [14] to control the network delay among Redis instances. The default inter-data center communication delay follows $\mathcal{N}(50, 10)$[5], 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.

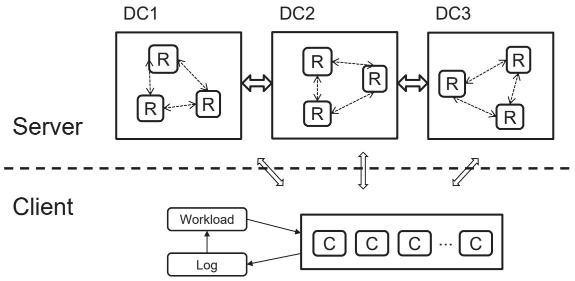

Fig. 3. Experiment setup.

C. *Experiment Design*

We design replicated priority queue and replicated list, using both the existing remove-win strategy [10] and our RWF design framework (namely the Remove-Win RPQ, the RWF-RPQ, the Remove-Win List and the RWF-List). The design and implementation of the data types used in the experiments are all available online[6].

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. We intentionally create additional add-add and add-rmv conflict for RPQ, with 15% of the total add/rmv operations to execute on the same data element concurrently. Because the probability of randomly generate such operations for RPQ is too low. 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.

Since the CRDTs serve operations instantly by design, we only measure the data consistency, to show how much data consistency is sacrificed to obtain the high performance. We have logged the read-time order of all client requests, using the result it may produce on a linearized server as the oracle of our consistency measurement. We measure the difference between the linearized result and the actual server result. For priority queue we measure the max value, and for list the edit distance. We further measure the edit distance between lists from different servers. Also we measure the metadata overhead, which 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[7]. 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]$\mathcal{N}(\mu, \sigma)$ stands for the normal distribution, where μ is the mean and σ is the standard deviation.

[6]See detailed discussions on the design in Appendix A-D of [13]. The source codes are also available in the repository [15].

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

TABLE I

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.4)		List-local (Fig.5)		List-replica (Fig.6)	
	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

D. Evaluation Results

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

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 [13], due to the limit of space.

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. 4, 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. 4 and Table I, 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[8].

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 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.

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. 5. 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.

[8]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 [13].

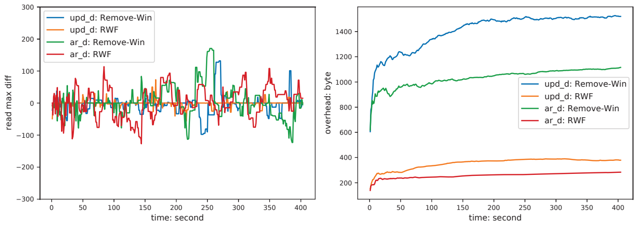

Fig. 4. The performance of RPQs, comparing max value read from the server with the max value of local linearized queue.

We then compare the lists on different servers at the same time instant. As shown in Fig. 6 and Table I, 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 I. 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 6), 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.

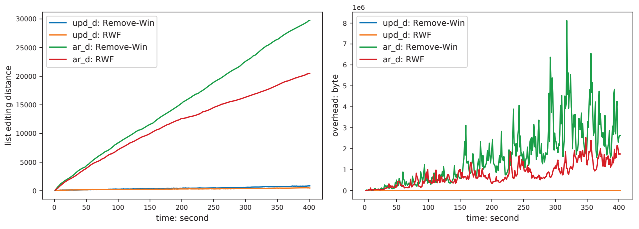

Fig. 5. The performance of lists, comparing the edit distance between the list read from the server and the local linearized list.

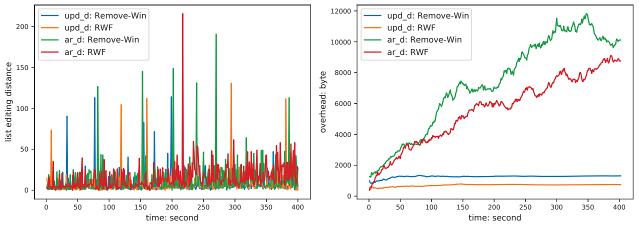

Fig. 6. The performance of lists, comparing the edit distance between two lists read from different servers at the same time.

V. 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 [10] 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 [16]. Multiple improved designs following WOOT were proposed, including WOOTO and WOOTH [17]. 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 [18] 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 [19]. 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.

VI. 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] 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

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

[6] 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

[7] 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

[8] 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.

[9] 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

[10] 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

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

[12] 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

[13] Y. Zhang, H. Wei, and Y. Huang. Remove-win: a design framework for conflict-free replicated data types. 01-15-2021. [Online]. Available: https://github.com/elem-azar-unis/CRDT-Redis/blob/master/document/rwf-tr.pdf

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

[15] "Conflict-free replicated data type implementations based on redis," https://github.com/elem-azar-unis/CRDT-Redis.

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

[17] 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

[18] 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

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