2023 IEEE International Conference on Cloud Computing Technology and Science (CloudCom) | 979-8-3503-3982-6/23/$31.00 ©2023 IEEE | DOI: 10.1109/CloudCom59040.2023.00047

Undoing CRDT Operations Automatically

Provakar Mondal and Eli Tilevich
Software Innovations Lab, Virginia Tech, USA
{provakar, tilevich}@cs.vt.edu

***Abstract*—In a distributed replicated data system, Conflict-free Replicated Data Types (CRDTs) keep data replicas consistent on different nodes, while providing intuitive programming abstractions for accessing and modifying the replicas. Due to user errors, program bugs, or hardware malfunctions, a CRDT can be updated incorrectly, so the effect of the executed CRDT update operations needs to be undone. However, because CRDT libraries rarely include the undo capability, adding it requires modifying source code by hand, a task that is hard to accomplish in a modular and reusable fashion. As a result, programmers end up adding this advanced functionality in an ad-hoc fashion, with the resulting code being hard to understand, maintain, and reuse. To address this problem, this paper presents AUTO-UNDO, an automatic approach that generates and actuates undo functionality for existing CRDT libraries, based on simple configurations and without modifying the library code by hand. The configurations specify which CRDT operations undo each other and the conditions that trigger the execution of undo procedures. Based on the configuration, AUTO-UNDO generates and actuates a sequence of update operations that undo the specified updates on a given replica. We have implemented and evaluated AUTO-UNDO in JavaScript, a popular CRDT language, demonstrating our approach's effectiveness, flexibility, and efficiency. Our experiences show that AUTO-UNDO effectively provides the undo capability for CRDT-based applications, thus streamlining the complexity of adding features to distributed programming frameworks.**

***Index Terms*—Distributed Computing, Fault Tolerance, Conflict-free Replicated Data Types, Undo, Code Generation**

I. Introduction

Modern distributed applications commonly replicate data across distributed execution sites. The resulting replication is required to reduce access latency, increase availability, and provide fault tolerance [1]. Because individual replicas can be modified independently, their states need to be synchronized across the distributed system to ensure consistency. Eventual consistency protocols have proven themselves as an effective means of synchronizing the states of distributed replicas in various application scenarios [2], [3].

Conflict-free replicated Data Types (CRDTs) have become a popular approach for providing eventual consistency. A CRDT is an abstract data type that provides an interface that exposes update and access methods, keeping the synchronization functionality out of the programmer's purview [4]. The CRDT runtime propagates the updates of an individual replica to the remaining replicas. The eventual consistency of CRDTs ensures their resilience against partial failure, caused by temporary network disconnection [5] or replica nodes going up and down [6]. Eventually, all updates are propagated to the participating replicas, so their states are synchronized. However, eventual consistency is no remedy for application errors, caused by incorrect user actions, system malfunction, or program bugs [7]. Modern replicated data systems commonly integrate volatile hardware components, such as sensors and actuators [8]. These realities often lead to incorrect updates to the distributed replicated data, with the erroneous states unrelated to the correct execution of eventual consistency protocols, thus propagating wrong data across the system.

To make it possible for developers to fix such incorrect updates, some CRDT libraries may provide the *undo* functionality, which restores the distributed state to some prior execution point [9], [10]. Despite the potential of the undo functionality as a powerful building block for error-handling strategies, established CRDT libraries rarely include it as part of their built-in functions. As a result, when needing to undo CRDT operations, developers end up implementing this non-trivial functionality by modifying the CRDT library code by hand. Even if the CRDT library code can be modified, adding the undo functionality this way can be laborious, time-consuming, and error-prone, with the resulting ad-hoc modifications being hard to understand and reuse [11]. Furthermore, a recent study reveals that application programmers face difficulties with using CRDT libraries correctly and find some CRDT functionalities confusing [12].

We address this problem with a generative approach that introduces the undo functionality to existing CRDT libraries based on declarative metadata specifications. In many CRDT libraries, some update operations form so-called *undo pairs*, with one operation canceling the effect of another operation. For example, `increment(x)` and `decrement(x)` form an undo pair. Additionally, only those update operations that change the CRDT state need to be undone. For example, if `increment(x)` ends up having no effect (e.g., as the data type has reached the specified upper bound), then undoing its intended effect would be erroneous. With the developer declaratively providing this information about the CRDT, our approach then automatically generates a sequence of update operations that we refer to as an `Undo Script`.

Our approach's implementation, AUTO-UNDO, adds the undo functionality to CRDT libraries in JavaScript, a widely used language in this domain. AUTO-UNDO makes use of the JSON format as its metadata and relies on JavaScript's advanced code adaptation techniques, such as monkey patching, to introduce the required additional functionalities [13].

This paper makes the following contributions:

2380-8004/23/$31.00 ©2023 IEEE
DOI 10.1109/CloudCom59040.2023.00047

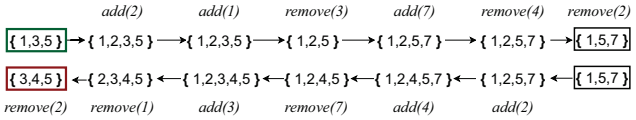


Fig. 1: Incorrect Undo Procedure

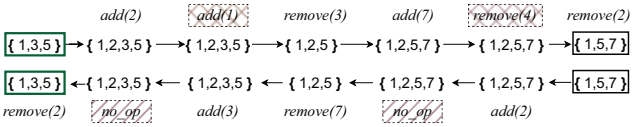


Fig. 2: `no_op` Undo Procedure

1) We present a novel approach for automatically generating and actuating the undo functionality for existing CRDT libraries without having to modify their source code by hand; parameterized via declarative metadata, the approach provides flexible options for configuring whether to actuate the generated Undo Scripts.
2) We provide a concrete implementation of our approach as AUTO-UNDO, which adds the required undo functionality to existing CRDT libraries implemented in JavaScript, configured via JSON metadata.
3) We report on the results of applying AUTO-UNDO to add the undo functionality to different CRDTs in third-party libraries; our evaluation assesses the applicability of our approach and its performance characteristics.

II. Motivating Example

Consider an Add-Remove Set CRDT, a replicated set data structure, whose functionality ensures the mathematical definition of a set: all elements are unique. Set elements can be added and removed. Once removed, elements can be added to the same set at a later point. Imagine an application scenario requiring that developers undo the effect of the last n update operations of this set. However, the CRDT library provides no built-in undo functionality.

One can come up with a seemingly straightforward solution. For each executed update operation, identify its counter-operation that cancels out the resulting effect (i.e., add for $remove$ and vice versa). Having executed an operation, record its counter-operation, and then replay the recorded counter-operations in reverse order.

Figure 1 depicts the aforementioned approach. The figure's upper part presents a set in the initial state of $\{1, 3, 5\}$, followed by six update operations, whose execution leads to the state of $\{1, 5, 7\}$. The figure's lower part presents the counter-operations for the executed update operations. Notice that the counter-operations are meant to be executed in reverse order. However, this undo strategy fails to correctly restore the set's initial state. Indeed, the resulting state arrives at $\{3, 4, 5\}$ rather than the expected $\{1, 3, 5\}$.

This undo strategy is flawed, as it fails to consider the data structure disallowing the execution of operations that violate its properties. In particular, adding an existing element to a set would have no effect. Similarly, removing a non-existing element would have no effect either. Hence, blindly recording the counter-operations may lead to invalid undo procedures.

To correct this strategy, one can check whether an update operation actually ends up updating the data structure's state, and only then record the corresponding counter-operation. The problem is that it might be prohibitively expensive to determine if a data structure's state has changed, particularly if it needs to be done for each operation. However, our approach efficiently determines whether a data structure's state has changed by leveraging domain-specific information provided by a developer, which we refer to as `no_op`. For example, if the `size` of a set before and after executing an operation remains the same, the operation is a `no_op`.

By exploiting this insight, our approach automatically generates a Undo Script. Notice how in our example, $add(1)$ and $remove(4)$ do not change the set's state, and we can determine that by comparing the set's size before and after executing these operations. Figure 2 depicts the execution and its undo procedure, cognizant of the `no_op` property. The $add(1)$ and $remove(4)$ appear as `no_op`, and as such are excluded from the generated Undo Script. As a result, the undo procedure correctly restores the set to its initial state of $\{1, 3, 5\}$.

III. Key Components

Figure 3 gives an overview of our approach as realized in AUTO-UNDO, whose programming model is developer-provided declarative metadata. The provided metadata describes the target CRDT data structures, their update operations, the counter operations for the updates, and the undo actuation conditions. Developers also mark the update operations that will need to be monitored and then undone, if required. We use a higher-order function for that purpose as explained below. AUTO-UNDO consists of four key components: Update Function Interceptor, Update Manager, Undo Script Generator, and Undo Script Actuator. The following subsections elaborate on the functionality of the corresponding components.

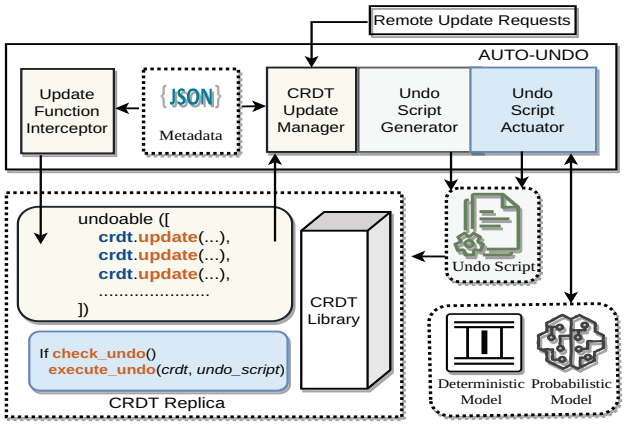


Fig. 3: Undo Generation and Actuation with AUTO-UNDO

A. Update Function Interceptor

This component intercepts the specified update operations of the target CRDT library, being invoked in the application.

AUTO-UNDO provides the undo functionality by proxying each update operation, removing the need to manually modify the CRDT library source code. Via proxying, AUTO-UNDO injects additional functionality that interacts with the runtime. Update Function Interceptor generates proxy functions that provide the necessary interception. Procedure 1 explains the functionality of the proxy functions. The main additional functionality is adding the corresponding undo action to the Undo Script for the intercepted update operation.

Procedure 1 Intercept Update Functions

```
1: procedure INTERCEPT_UPDATE_FUNCTIONS
2:     crdt_functions ← read_functions(metaData)
3:     for all func ∈ crdt_functions do
4:         params ← metaData[func.params]
5:         func ← generate_proxy(func, params)
6:     end for
7: end procedure
8:
9: procedure GENERATE_PROXY (func, params)
10:    origFunc ← func
11:    generate_undo_action(func, params)
12:    origFunc(params)
13: end procedure
```

B. Update Manager

The Update Manager component tracks the specified update operations of the CRDT library, as they are being invoked in the application. To keep track of which update operations to monitor, AUTO-UNDO provides a higher-order function `undoable([...])`, which accepts the invocations of the marked update operations as parameters. Additionally, the Update Manager queues up the remote update requests received from other replicas while tracking the marked update operations. If the execution of an Undo Script is allowed to be interleaved with the processing of external update requests, the resulting CRDT state may deviate from that under which the counter operations were determined. To prevent this issue, the Update Manager queues up all incoming update requests issued by the other replicas in the order of arrival. After completing the execution of `undoable([...])` (having actuated an Undo Script or not), it then processes the queued update requests. Because AUTO-UNDO proxies all update functions, it has complete control over their execution.

C. Undo Script Generator

The Undo Script Generator component automatically generates the required sequence of undo actions for the invoked update operations. By executing the resulting Undo Script, AUTO-UNDO brings the CRDT's state to the point before the first monitored update operation was executed. Procedure 2 details the generation procedure for a single monitored update operation. Notice how the generation logic makes use of the `no_op` information, also specified declaratively in metadata.

D. Undo Script Actuator

The Undo Actuator (1) checks the current state of the CRDT to determine if an Undo Script needs to be executed by applying a decision-making model, and (2) executes the Undo Script if needed. Procedure 3 describes the logic used to determine whether an undo is needed. For its decision-making, AUTO-UNDO provides deterministic and probabilistic models, as configured in metadata. The deterministic model makes use of standard statistical functions and static thresholds. The outcome of a deterministic model is solely determined by the specified statistical functions, their thresholds, and the CRDT data. If checking whether an undo is needed by a deterministic model becomes computationally prohibitive, AUTO-UNDO provides an alternative probabilistic model that takes advantage of Machine Learning (ML). It uses the AUTO-UNDO's execution history to train itself (i.e., among the previous checks, what were the states of the CRDT, which led to the positive and negative outcomes). This trained model is then used to compute the probability that the undo is needed at a given point. Developers also configure the probability threshold under which an Undo Script is to be executed. Finally, to specify whether AUTO-UNDO should utilize the deterministic or probabilistic model, developers can use metadata to provide the corresponding conditions. For added flexibility, AUTO-UNDO makes it possible for developers to express their own custom decision-making logic as a function parameter. Finally, the Undo Actuator automatically actuates the undo procedure once it is triggered by the specified conditions, as explained in Procedure 4.

Procedure 2 Generate Undo Action

```
1: procedure GENERATE_UNDO_ACTION (funcName, params)
2:     no_op ← is_no_op(funcName, params)
3:     if not no_op then
4:         undoFuncName ← metaData[funcName]
5:         undo_stack.push(undoFuncName, params)
6:     end if
7: end procedure
```

Procedure 3 Check Undo

```
1: procedure UNDO_CHECK (crdt)
2:     params ← read_params(crdt)
3:     option ← read_options(crdt)
4:     if option is in metaData then
5:         model ← read_model(crdt, metaData)
6:         if model is deterministic then
7:             res ← deterministic_model(params)
8:             return res > metaData[det_threshold]
9:         else
10:            res ← probabilistic_model(params)
11:            return res > metaData[prob_threshold]
12:        end if
13:    else
14:        accept_custom_predicate(customLogic)
15:    end if
16: end procedure
```

IV. Evaluation

Our evaluation is driven by the following questions:

1) **Q1:** What is the performance overhead of applying AUTO-UNDO to add the undo capability in existing CRDT libraries under different workloads?

Procedure 4 Execute Undo Actions

```
1: procedure EXECUTE_UNDO_ACTIONS (crdt)
2:     if undo_check(crdt) then                    ▷ Undo Required
3:         for all action ∈ undo_stack do
4:             func ← action.crdtFuncName
5:             params ← action.crdtparams
6:             func(crdt, params)
7:         end for
8:     end if
9: end procedure
```

TABLE I: Latency Overhead (%)

Number of Updates	`js-delta-crdts`			`crdts`	
	Counter	Set	Map	Counter	Set
100	2.93	3.87	3.36	2.64	3.08
200	2.19	3.98	4.18	3.13	3.45
300	3.27	4.32	4.37	3.49	4.07
500	3.61	4.94	4.78	4.33	4.56

2) **Q2:** What is the accuracy/performance trade-off between the deterministic and probabilistic models for determining whether to execute an undo procedure?

In our evaluation, we applied AUTO-UNDO to two existing third-party CRDT libraries, `js-delta-crdts` [14] and `crdts` [15]. From `js-delta-crdts`, we used three CRDT data structures: PNCounter, Add-Remove Set, and Map; from `crdts`, we used PNCounter and Add-Remove Set, as `crdts` has no map data structure.

The following subsections describe the evaluation tasks we conducted to answer the aforementioned questions. To answer **Q1:**, we benchmarked the performance impact of applying AUTO-UNDO to the target CRDT libraries. Finally, to answer **Q2:**, we correlated the execution time and the accuracy rate of the deterministic/probabilistic models to determine their correctness/performance trade-off.

A. Q1: Performance

To understand the performance costs of applying AUTO-UNDO to add the undo capability, we compared the overall latencies of executing different numbers of CRDT update operations with the same operations enhanced with the undo capability by AUTO-UNDO. Recall that it is the provided decision-making model (deterministic or probabilistic) that determines whether to actuate a Undo Script. Therefore, the total additional latency incurred by AUTO-UNDO includes the costs of generating Undo Script, consulting the model, and actuating the undo procedure if required. In our experiments, we applied AUTO-UNDO to the aforementioned two existing third-party CRDT libraries and measured the performance under different workloads under two settings: (1) baseline CRDT update operations (no AUTO-UNDO present); (2) the same operations but with AUTO-UNDO. Figures 4 and 5 present the experimental results for three CRDTs in the first library and two CRDTs in the second one respectively. Furthermore, we averaged the imposed overhead as percentage points, detailed in Table I.

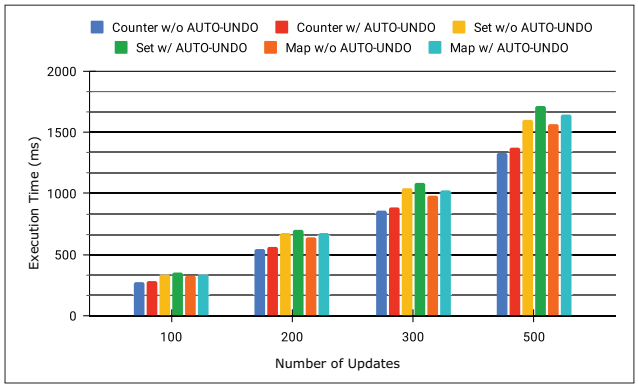


Fig. 4: The overhead for the `js-delta-crdts` library

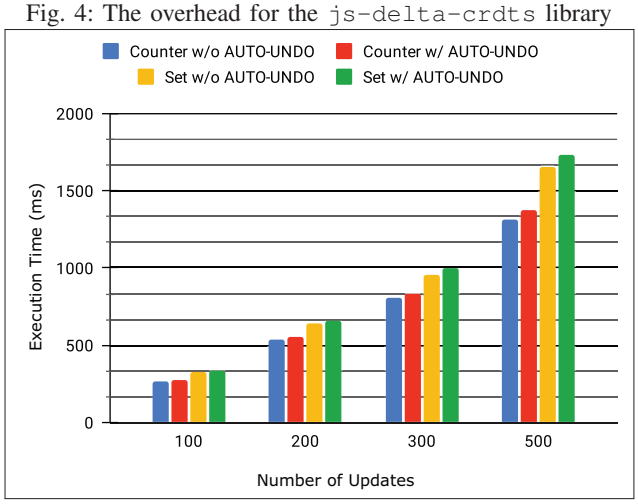


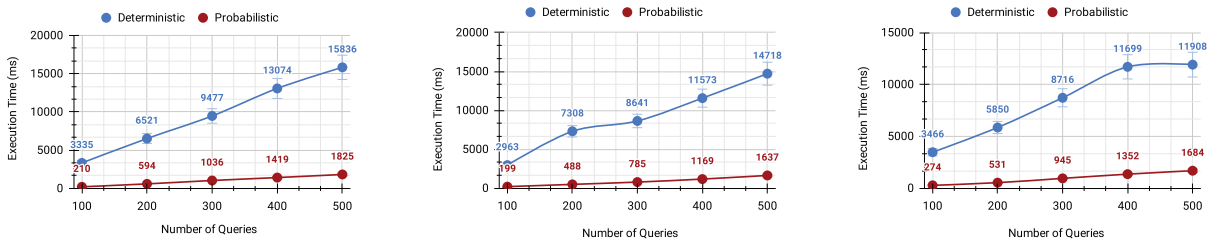
Fig. 5: The overhead for the `crdts` library

a) Experimental Setup: We created a virtual distributed environment on a 64-bit Ubuntu 20.04 computer with 32 GB of RAM and an Intel Core i7 Processor to conduct our performance evaluation. Having a virtual environment makes it possible to avoid the inherent volatility of distributed execution to be able to isolate AUTO-UNDO's performance overhead. We deployed three virtual nodes to measure the overhead generated by the target CRDT libraries.

With the original CRDT deployment establishing the performance baseline, we observed that AUTO-UNDO's usage incurs a constant overhead, unaffected by the number of update operations. Furthermore, the overhead never exceeds 5% in the worst-case scenario of tracking 500 updates operation of Set CRDT in `js-delta-crdts` library. The observed constant overhead is reasonable, given the saved programming effort of AUTO-UNDO, which provides versatile undo functionality without modifying the CRDT source code by hand, based on declarative metadata input.

B. Q2: Decision-Making Accuracy/Performance Trade-off

To check whether to execute an undo procedure, AUTO-UNDO takes advantage of two decision-making models: deterministic and probabilistic. These models possess dissim-



(a) Add-Remove Set (`js-delta-crdts`) (b) Map (`js-delta-crdts`) (c) Add-Remove Set (`crdts`)

Fig. 6: Latency Comparison of Deterministic vs Probabilistic Model

(a) Add-Remove Set (js-delta-crdts) (b) Map (`js-delta-crdts`) (c) Add-Remove Set (`crdts`)

Fig. 7: Correct/Incorrect Ratio for the Probabilistic Model

ilar accuracy performance characteristics. The deterministic model applies the specified statistical procedures to accurately determine whether to execute an undo. However, statistical computations over large data sets can become prohibitively expensive. In contrast, the probabilistic model approximates the decision-making process using an ML procedure (the reference implementation used `Logistic Regression`). Although beating in performance the deterministic model for large data sets, the probabilistic model lacks accuracy. However, its accuracy keeps growing with usage in line with the training feature of ML.

Figure 6 compares the two models in terms of their latency characteristics, without taking the corresponding accuracy into account. To isolate the performance, the experiments were run on a single replica. In this experiment, we added more than 1,000 elements to the Set and Map CRDTs. Then we trained the Probabilistic Model on the responses produced by executing the Deterministic Model 1000 times. The Deterministic Model was configured to compute the Standard Deviation of the contained elements. Since model training can be performed offline, our measurements exclude the time it took to train the ML routine of the Probabilistic Model. As one can see, the performance benefits of the Probabilistic Model increase their prominence both with the size of the training dataset and usage frequency.

However, the superior performance of the Probabilistic Model comes at the price of correctness, as demonstrated in Figure 7. The number of correct responses appears in green, while incorrect ones are in red. The ML property manifests itself in the ratio of $correct : incorrect$ responses increasing with the number of queries, as the model keeps learning from the previous interactions.

It is ultimately up to the developer to determine which model to use in different application scenarios. Irrespective of the choice made, it takes a simple metadata directive to specify which model to use. We plan to explore the suitability of different models in scenarios as a future work direction.

V. Related Work

Because the undo functionality serves as a useful component for constructing strategies for handling errors and system faults, enhancing CRDTs with undo has gained prominence as a research topic. The research literature primarily contains examples of highly customized CRDTs with built-in undo capabilities. Some of the recent examples introduce undo-enhanced CRDTs as a distributed key-value database with reversible operations [16] as well as providing robustness and uniformity for reversing CRDT operations [17]. The ability to undo CRDT operations has been found particularly beneficial in the domain of collaborative editing. Representative examples focus on different issues that include: undo causing undesirable effects that are addressed by the selective undo of string-wise operations [18]; applying undo functionality to any number of operations rather than only the last ones [19]; enabling to “undo anywhere, anytime” with low-cost and formally proved [20]; presenting undo algorithms that satisfy the conditions of neutrality and forward transposition [21]; supporting selective undo with an algorithm that provides operational transformation consistency control, with causality

and admissibility preservation [22]; providing multiple undo with a generic algorithm that generates valid undo operations [23]; presenting a JavaScript framework, named Yjs for collaborative editing of arbitrary data types with an Undo/Redo manager, able to undo/redo selective updates in a domain-specific fashion, designed to work specifically with collaborative editing [24]. A related approach that provides undo for CRDTs out-of-the-box, albeit for state-based CRDTs, defines a formal semantics of concurrent undo and redo operations and logs state deltas [9].

The presence of such an extensive prior body of work concerned with adding the undo capability to CRDT confirms the necessity of efforts such as ours that target this problem from the perspective of novel programming approaches. Our work explores solutions to this problem that exploits the conventions of CRDT libraries to provide concise yet effective techniques for automating the process of generating and actuating versatile undo procedures. AUTO-UNDO's declarative programming model can be potentially beneficial for some of the aforementioned state-of-the-art approaches, an angle that we plan to pursue as a future work direction.

VI. Conclusion

We have presented AUTO-UNDO, an automatic approach that enhances CRDTs with the undo capability. The required extra functionalities are added without having to modify the CRDT library code by hand, based on input expressed as declarative metadata. The reference implementation of AUTO-UNDO targets JavaScript, a popular language for CRDT libraries. AUTO-UNDO decides whether to actuate the generated Undo Script based on decision-making models, both deterministic driven by statistical functions and probabilistic driven by ML.

Our evaluation results demonstrate the feasibility of declarative meta-programming for generating and actuating undo in realistic CRDT scenarios. The introduced undo functionality incurs a reasonable performance overhead at runtime. Finally, AUTO-UNDO's decision-making process offers flexibility in its correctness/performance tradeoff between the deterministic/probabilistic models.

Acknowledgments

This research is supported by NSF via Grant #2232565.

References

[1] B. Krishnamachari and S. Kapadia, *"Data replication and scheduling for content availability in vehicular networks"*. University of Southern California, 2007.

[2] X. Zhao and P. Haller, "Replicated data types that unify eventual consistency and observable atomic consistency," *Journal of logical and algebraic methods in programming*, vol. 114, p. 100561, 2020.

[3] V. Balegas, D. Serra, S. Duarte, C. Ferreira, M. Shapiro, R. Rodrigues, and N. Preguiça, "Extending eventually consistent cloud databases for enforcing numeric invariants," in *2015 IEEE 34th Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 2015, pp. 31–36.

[4] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," in *Stabilization, Safety, and Security of Distributed Systems: 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings 13*. Springer, 2011, pp. 386–400.

[5] Y.-W. Kwon, E. Tilevich, and T. Apiwattanapong, "DR-OSGi: Hardening distributed components with network volatility resiliency," in *Middleware 2009*, J. M. Bacon and B. F. Cooper, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 373–392.

[6] M. Kleppmann and H. Howard, "Byzantine eventual consistency and the fundamental limits of peer-to-peer databases," *arXiv preprint arXiv:2012.00472*, 2020.

[7] P. Bailis and A. Ghodsi, "Eventual consistency today: Limitations, extensions, and beyond: How can applications be built on eventually consistent infrastructure given no guarantee of safety?" *Queue*, vol. 11, no. 3, pp. 20–32, 2013.

[8] M. Capra, R. Peloso, G. Masera, M. Ruo Roch, and M. Martina, "Edge computing: A survey on the hardware requirements in the internet of things world," *Future Internet*, vol. 11, no. 4, p. 100, 2019.

[9] W. Yu, V. Elvinger, and C.-L. Ignat, "A generic undo support for state-based crdts," in *OPODIS 2019-Proceedings of 23rd International Conference on Principles of Distributed Systems*, 2019.

[10] E. Brattli and W. Yu, "Supporting undo and redo for replicated registers in collaborative applications," in *18th International Conference on Cooperative Design, Visualization, and Engineering*, ser. CDVE 2021. Springer LNCS volume 12983, Oct. 2021, pp. 195–205.

[11] J. Rubin, *Cloned product variants: From ad-hoc to well-managed software reuse*. University of Toronto (Canada), 2014.

[12] Y. Zhang, M. Weidner, and H. Miller, "Programmer Experience When Using CRDTs to Build Collaborative Webapps: Initial Insights," 3 2023. [Online]. Available: https://kilthub.cmu.edu/articles/conference_contribution/Programmer_Experience_When_Using_CRDTs_to_Build_Collaborative_Webapps_Initial_Insights/22277341

[13] Y. Wang, K. S. Cheng, M. Song, and E. Tilevich, "A declarative enhancement of javascript programs by leveraging the java metadata infrastructure," *Science of Computer Programming*, vol. 181, pp. 27–46, 2019.

[14] P. Teixeira, "Delta state-based crdts in javascript," 2018, https://github.com/peer-base/js-delta-crdts.

[15] R. Littauer, "A library of conflict-free replicated data types for javascript," 2016, https://github.com/orbitdb/crdts.

[16] Y. Mao, Z. Liu, and H.-A. Jacobsen, "Reversible conflict-free replicated data types," in *Proceedings of the 23rd ACM/IFIP International Middleware Conference*, 2022, pp. 295–307.

[17] N. Saquib, C. Krintz, and R. Wolski, "Log-based crdt for edge applications," in *2022 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2022, pp. 126–137.

[18] W. Yu, L. André, and C.-L. Ignat, "A crdt supporting selective undo for collaborative text editing," in *Distributed Applications and Interoperable Systems: 15th IFIP WG 6.1 International Conference, DAIS 2015, Held as Part of the 10th International Federated Conference on Distributed Computing Techniques, DisCoTec 2015, Grenoble, France, June 2-4, 2015, Proceedings 15*. Springer, 2015, pp. 193–206.

[19] S. Martin, P. Urso, and S. Weiss, "Scalable xml collaborative editing with undo: (short paper)," in *On the Move to Meaningful Internet Systems: OTM 2010: Confederated International Conferences: CoopIS, IS, DOA and ODBASE, Hersonissos, Crete, Greece, October 25-29, 2010, Proceedings, Part I*. Springer, 2010, pp. 507–514.

[20] S. Weiss, P. Urso, and P. Molli, "Logoot-undo: Distributed collaborative editing system on p2p networks," *IEEE transactions on parallel and distributed systems*, vol. 21, no. 8, pp. 1162–1174, 2010.

[21] J. Ferrié, N. Vidot, and M. Cart, "Concurrent undo operations in collaborative environments using operational transformation," in *On the Move to Meaningful Internet Systems 2004: CoopIS, DOA, and ODBASE: OTM Confederated International Conferences, CoopIS, DOA, and ODBASE 2004, Agia Napa, Cyprus, October 25-29, 2004. Proceedings, Part I*. Springer, 2004, pp. 155–173.

[22] B. Shao, D. Li, and N. Gu, "An algorithm for selective undo of any operation in collaborative applications," in *Proceedings of the 2010 ACM International Conference on Supporting Group Work*, 2010, pp. 131–140.

[23] C. Sun, "Undo as concurrent inverse in group editors," *ACM Transactions on Computer-Human Interaction (TOCHI)*, vol. 9, no. 4, pp. 309–361, 2002.

[24] K. Jahns, "Yjs: A crdt framework with a powerful abstraction of shared data," 2014, https://github.com/yjs/yjs.