Eventually Consistent Register Revisited

Marek Zawirski∗
Inria & Sorbonne Universités,
UPMC Univ Paris 06, LIP6
marek.zawirski@gmail.com

Carlos Baquero
HASLab, INESC TEC &
Universidade do Minho
cbm@di.uminho.pt

Annette Bieniusa
Technische Universität
Kaiserslautern
bieniusa@informatik.uni-kl.de

Nuno Preguiça
NOVA LINCS, DI, FCT,
Universidade NOVA de Lisboa
nuno.preguica@fct.unl.pt

Marc Shapiro
Inria & Sorbonne Universités,
UPMC Univ Paris 06, LIP6
marc.shapiro@acm.org

ABSTRACT
In order to converge in the presence of concurrent updates, modern eventually consistent replication systems rely on causality information and operation semantics. It is relatively easy to use semantics of high-level operations on replicated data structures, such as sets, lists, etc. However, it is difficult to exploit semantics of operations on registers, which store opaque data. In existing register designs, concurrent writes are resolved either by the application, or by arbitrating them according to their timestamps. The former is complex and may require user intervention, whereas the latter causes arbitrary updates to be lost. In this work, we identify a register construction that generalizes existing ones by combining runtime causality ordering, to identify concurrent writes, with static data semantics, to resolve them. We propose a simple conflict resolution template based on an application-predefined order on the domain of values. It eliminates or reduces the number of conflicts that need to be resolved by the user or by an explicit application logic. We illustrate some variants of our approach with use cases, and how it generalizes existing designs.

CCS Concepts
• Theory of computation → Distributed algorithms;

Keywords
Multi-Value-Register, Eventual Consistency, CRDTs.

1. BACKGROUND
An eventually-consistent replication system accepts updates concurrently at different replicas. The challenge is to ensure convergence of values at all replicas under absence of a common execution order of updates. To this end, replicas need to interpret delivered updates into a value without relying on execution order. Formally, the intended value of an object can be specified in this manner as a function on the set of delivered updates partially ordered by causality $\mathcal{E}$. Value of abstract data types, such as set, list or counter, can be easily expressed in this way with the help of their method semantics or causality relation $\mathcal{E}$. This is harder for a low-level register data type with write and read operations, which provide little semantics to make use of.

A classical approach is the multi-value register that uses causality information to provide all concurrent writes to the application $\mathcal{E}$. For the multi-value register that stores values from a domain $V$, the register value is specified by a function $\mathcal{F}_{\text{mv}}$ that produces a subset of values from $V$:

$$\mathcal{F}_{\text{mv}}(E, \mathcal{E}) = \{ v | \exists e \in E : e = \text{write}(v) \land \exists e' \in E : e' = \text{write}(_{\perp}) \land e \xrightarrow{\mathcal{E}} e' \} ,$$

where $E$ is a set of events observed by read operation, and $\mathcal{E}$ is a causality partial order on $E$. Provided all replicas eventually observe the same set of updates, and always observe restriction of a common causality relation, the register converges $\mathcal{E}$.

When more than one value appears in the set returned by the multi-value register, it indicates concurrent updates, called a conflict. Conflicts are undesirable, since either the application or the user need to resolve them, which is complex and may in turn cause another conflict.

2. REGISTER WITH DATA-DRIVEN CONFLICT RESOLUTION
We propose a simple template for conflict resolution based on a predefined order of values. This approach reduces or even eliminates the number of conflicts that need to be resolved by an explicit logic or by the user.

We define a generalization of the classical multi-value register as $\mathcal{F}_{\text{mv}}$:

$$\mathcal{F}_{\text{mv}}(E, \mathcal{E}) = \text{resolve}(\mathcal{F}_{\text{mv}}(E, \mathcal{E})),$$

where resolve : $\mathcal{P}(V) \rightarrow \mathcal{P}(V)$ is a function that can resolve some or all of the conflicts. Hereafter, we identify some simple yet useful classes of resolve.

2.1 Partially Ordered Values

∗Now at Google.
The register eliminates concurrently written values that are variants of \( \forall \) pairs (write at the time of write, becoming effectively write ior similar to the last-writer-wins policy (LWW) [2], provided based on runtime-provided information, such as replica ID or fixed, which requires user intervention.

Concurrent modifications of the status converge to a single open, and totally-ordered domain of priority levels. Our construction may store priority level of a bug, from a predefined semantics of stored values.

3.1 Semantics-based Ordering

An application can define the order according to the semantics of stored values.

A special case of partial order is a total order. Under total ordering – all values are concurrent, and thus not order reducible; Partial order – one or more maximal values are kept after resolve; Total order – a single maximal value is kept after resolve. The algorithm includes an optimization that allows storing a single scalar logical clock to identify each written value, complemented by a version vector for the whole register. The classical multi-value register implementation stores a version vector per value [4, 5].

The state is composed by a set of values, tagged by scalar clocks, and by a common version vector. The scalar clocks are locally generated by using a replica id \( i \in \mathbb{I} \) and a monotonic counter per replica. A write operation \( \text{write}(v, \sigma) \) is depicted as a state transforming function, tagged with the replica id \( i \), and supplying a value \( v \) and the current state \( \sigma = (s, c) \), where \( s \) is the set and \( c \) is the “causal context” version vector. Each write uses the version vector to create a new scalar clock and derives a new set with a single value tagged by the scalar clock, as well as an updated version vector that includes the new scalar. The read operation \( \text{read}(\sigma) \) keeps the state unchanged and replies with a set comprising all values present in the multi-value register, stripped of clock metadata.

Since writes always derive a set with a single value, the set will only have multiple values as a result of a merge that gathers concurrently assigned values, written in different replica states. The merge collects concurrently assigned values that have not been overwritten and supplies these values to the \( \text{resolve}_\prec \), for possible further reduction on resulting set.

An advantage of our approach compared to the classical LWW register is that the timestamps are used to arbitrate the concurrent values only, avoiding some of the arbitration anomalies caused by physical clocks [3]. For instance, it is no longer possible to timestamp a write, with a far future time, and prevent later writes to appear. Any write that observes this write will be, in our construction, ordered after that write, regardless of the timestamp that is present for the tagging.

4. IMPLEMENTATION

We illustrate an implementation of the proposed register in the state-based eventually-consistent replication model [5]. In this model, replicas opportunistically exchange their complete states via message passing.

The register implementation in Figure 1 uses an implementation of \( \text{resolve}_\prec \) to reduce any concurrently assigned values according to the partial order \( \prec \) defined by the application on those values. The order among values can range from: No ordering – all values are concurrent, and thus not order reducible; Partial order – one or more maximal values are kept after resolve; Total order – a single maximal value is kept after resolve. The algorithm includes an optimization that allows storing a single scalar logical clock to identify each written value, complemented by a version vector for the whole register. The classical multi-value register implementation stores a version vector per value [4, 5].

The state is composed by a set of values, tagged by scalar clocks, and by a common version vector. The scalar clocks are locally generated by using a replica id \( i \in \mathbb{I} \) and a monotonic counter per replica. A write operation \( \text{write}(v, \sigma) \) is depicted as a state transforming function, tagged with the replica id \( i \), and supplying a value \( v \) and the current state \( \sigma = (s, c) \), where \( s \) is the set and \( c \) is the “causal context” version vector. Each write uses the version vector to create a new scalar clock and derives a new set with a single value tagged by the scalar clock, as well as an updated version vector that includes the new scalar. The read operation \( \text{read}(\sigma) \) keeps the state unchanged and replies with a set comprising all values present in the multi-value register, stripped of clock metadata.

Since writes always derive a set with a single value, the set will only have multiple values as a result of a merge that gathers concurrently assigned values, written in different replica states. The merge collects concurrently assigned values that have not been overwritten and supplies these values to the \( \text{resolve}_\prec \), for possible further reduction on resulting set.

The implementation in \( \text{deliver} \) detects values that have been observed and later overwritten by checking that the scalar clocks associated to those values are included in the version vector \( c \) while those entries are no longer present in the set \( s \). Values still present on both sets, or newly written values are
kept. This detects and keeps all concurrently assigned values, but when resolve is finally called some of these values can be removed if the order information on values indicates that they are dominated by a higher value.

Figure 2 shows a run of a system with two replicas for the bug tracking example mentioned before. After the first synchronization from replica B to replica A, the state will be closed-irrep, as this value is greater than assigned in the order of values. After the second synchronization, the register will maintain two values as closed-irrep and closed-fixed are incomparable. Later, these values are replaced by a new write with value assigned.

Figure 3 shows a run with the last-writer-wins behavior. In this example, we assume that replica B has a local clock at a higher value. We can see that after the first write in replica B is propagated to replica A, the following write in A will overwrite the value previously written by replica B, although the new timestamp is smaller. The reason for this is that the timestamp is only used to arbitrate among concurrent values.

Acknowledgments.

This project was supported in part by: Project Norte-01-0145-FEDER-000020 under the North Portugal Regional Operational Programme (Norte 2020), under the Portugal 2020 Partnership Agreement, and through the European Regional Development Fund (ERDF); EU FP7 SyncFree project (609551); FCT/MCT projects SwiftComp (PTDC/ EEI-SCR/1837/2012) and NOVA LINCS (UID/CEC/04516/2013); and a Google Faculty Research Award 2013.

References


