Eventually Consistent Register Revisited

Marek Zawirski, Carlos Baquero, Annette Bieniusa, Nuno Preguiça, Marc Shapiro

Eventually Consistent Register Revisited

Marek Zawirski*[1], Carlos Baquero[2], Annette Bieniusa[3], Nuno Preguiça[4], and Marc Shapiro[1]

[1]Inria & Sorbonne Universités, UPMC Univ Paris 06, LIP6
[2]HASLab, INESC Tec & Universidade do Minho
[3]U. of Kaiserslautern
[4]NOVA LINCS, DI, FCT, Universidade NOVA de Lisboa

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.

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,

*Now at Google.

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 [1]. 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 [5]. 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 [4, 5]. For the multi-value register that stores values from a domain V, the register value is specified by a function $\mathcal{F}_{\texttt{mvr}}$ that produces a subset of values from V:

$$\mathcal{F}_{\texttt{mvr}}(E, \mathsf{hb}) = \{v \mid \exists e \in E : e = \texttt{write}(v) \\ \wedge \nexists e' \in E : e' = \texttt{write}(\_) \wedge e \xrightarrow{\mathsf{hb}} e'\},$$

where E is a set of events observed by read operation, and hb 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 [1].

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 $\mathcal{F}_{\texttt{mvrr}}$:

$$\mathcal{F}_{\texttt{mvrr}}(E, \mathsf{hb}) = \mathsf{resolve}(\mathcal{F}_{\texttt{mvr}}(E, \mathsf{hb})),$$

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

2.1 Partially Ordered Values

Let $\prec$ be a strict partial order predefined on values V by the application, embedded in the object type. We define $\mathsf{resolve}_{\prec}$ based on this order as:

$$\mathsf{resolve}_{\prec}(V) = \{v \in V \mid \nexists v' \in V : v \prec v'\}.$$

The register eliminates concurrently written values that are dominated in $\prec$. The result is the set of maximal values acording to the order on values. This reduces the number of conflicts that the user, or the application, need to resolve. When the order $\prec$ is not provided (empty), $\mathsf{resolve}_{\prec}$ behaves as the identity function, as in the classical multi-value register.

2.2 Totally Ordered Values

A special case of partial order is a total order. Under total order, $\mathsf{resolve}_{\prec}$ ensures that the register presents at most one value to the application, i.e., $\forall X : |\mathsf{resolve}(X)| \leq 1$. This is a desirable property, since applications and users are often expecting to read a single value, as in the sequential register.

3 Use Cases

Instantiations of our construction can be applied to a number of use cases.

3.1 Semantics-based Ordering

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

For example, consider a software bug tracking system. A register may store priority level of a bug, from a predefined and totally-ordered domain of priority levels. Our construction provides a reasonable convergent behavior: concurrent assignments of different levels should converge to the highest one. Nevertheless, it allows to decrease the level again, with a later assignment.

A bug tracker may use another register to store status of a bug. Consider the following status options: *open*, *assigned*, *closed-fixed*, and *closed-irreproducible*. In this case, the application can specify a partial order on statuses, e.g., *assigned* dominates *open*, dominated in turn by the two incomparable variants of *closed*. Using this order with our construction, concurrent modifications of the status converge to a single value, except when the bug is both marked as irreproducible and fixed, which requires user intervention.

3.2 Runtime-based Ordering

Although the order of values $\prec$ is static, it can be also based on runtime-provided information, such as replica ID or timestamp. In particular, our construction can achieve behavior similar to the last-writer-wins policy

(LWW) [2], provided every $\texttt{write}(v)$ operation is augmented with a timestamp t at the time of write, becoming effectively $\texttt{write}((v,t))$, and pairs (v,t) are totally-ordered according to their t.

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.

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.

Replica states	Σ	$=$	$\mathcal{P}(\mathbb{I} \times \mathbb{N} \times V) \times (\mathbb{I} \to \mathbb{N})$
Initial state	σ_i^0	$=$	$(\{\}, \{\})$
Write at replica i	$\texttt{write}_i\big(v, (s,c)\big)$	$=$	$\big(\{(i, c[i]+1, v)\}, c[i \mapsto c[i]+1]\big)$
Read at replica i	$\texttt{read}_i\big((s,c)\big)$	$=$	$\{v \mid (\_,\_,v) \in s\}$
Merge replica states	$\mathsf{deliver}\,\big((s,c),(s',c')\big)$	$=$	$\mathsf{resolve}_\prec\big((s \cap s') \cup \{(i,n,v) \in s \mid n > c'[i]\}$ $\cup\{(i',n',v') \in s' \mid n' > c[i']\}, c \sqcup c'\big)$
where	$\mathsf{resolve}_\prec\big((s,c)\big)$	$=$	$\big(\{(i,n,v) \in s \mid \nexists (\_,\_,v') \in s \cdot v \prec v'\}, c\big)$

Figure 1: Optimized implementation of register with $\mathsf{resolve}_\prec$, replica i.

The register implementation in Figure 1 uses an implementation of $\mathsf{resolve}_\prec$ 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 $\texttt{write}_i(v, \sigma)$ 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 $\texttt{read}_i(\sigma)$ 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 $\mathsf{resolve}_{\prec}$ for possible further reduction on resulting set. The implementation in deliver detects values that have been observed and later overwritten by checking that the scalar cloks 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 $\mathsf{resolve}_{\prec}$ 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 work was partially supported by EU FP7 SyncFree project (609551), FCT/MCT projects PEST-OE/EEI/UI0527/2014 and UID/EEA/50014/2013, and by a Google Faculty Research Award 2013.

References

[1] S. Burckhardt, A. Gotsman, H. Yang, and M. Zawirski. Replicated Data Types: Specification, verification, optimality. In *POPL*, San Diego, CA, USA, Jan. 2014.

[2] P. R. Johnson and R. H. Thomas. The maintenance of duplicate databases. Internet Request for Comments RFC 677, Information Sciences Institute, Jan. 1976.

[3] K. Kingsbury. The trouble with timestamps. `https://aphyr.com/posts/299-the-trouble-with-timestamps`, Oct. 2013.

[4] D. S. Parker, Jr., G. J. Popek, G. Rudisin, and A. Stoughton et al. Detection of mutual inconsistency in distributed systems. *IEEE ToSE*, SE-9(3):240–247, May 1983.

[5] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free Replicated Data Types. In *SSS*, volume 6976 of *LNCS*, Grenoble, France, Oct. 2011. Springer Verlag.

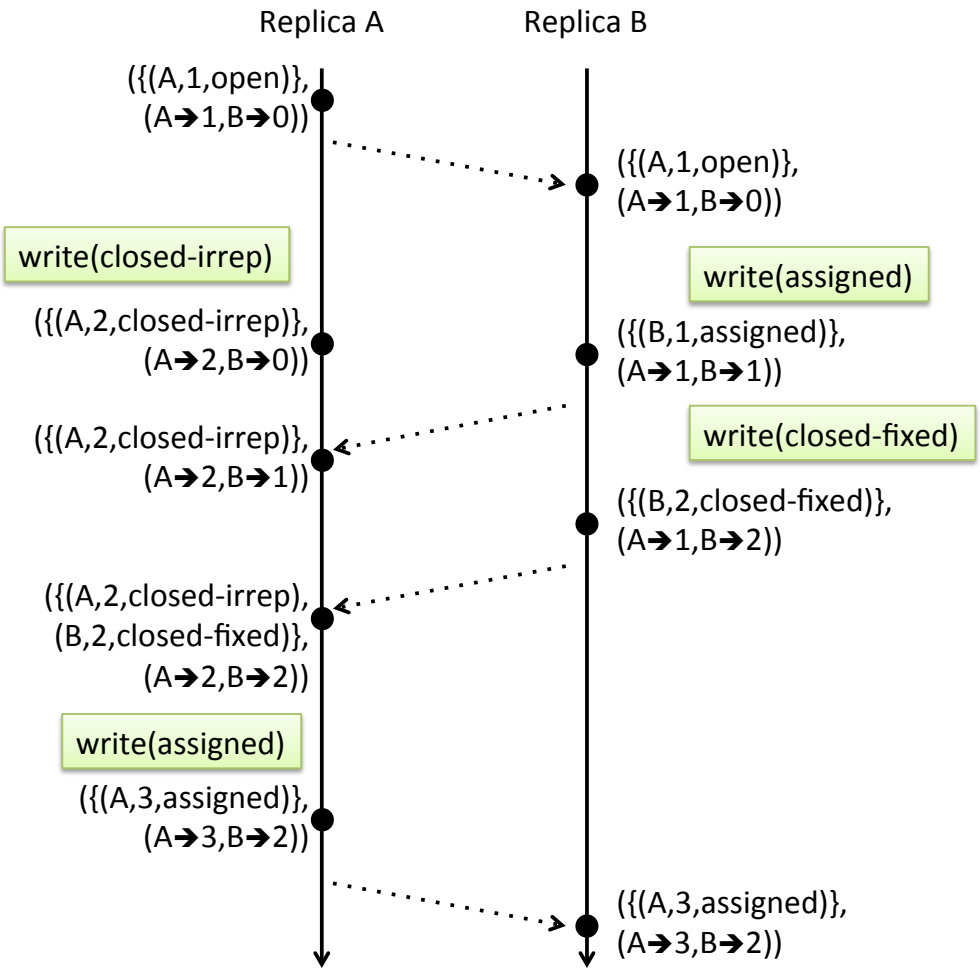

Figure 2: Bug tracking run with two register replicas; dashed arrow represents a message, merged at the receiver replica; solid box indicates a client operation.

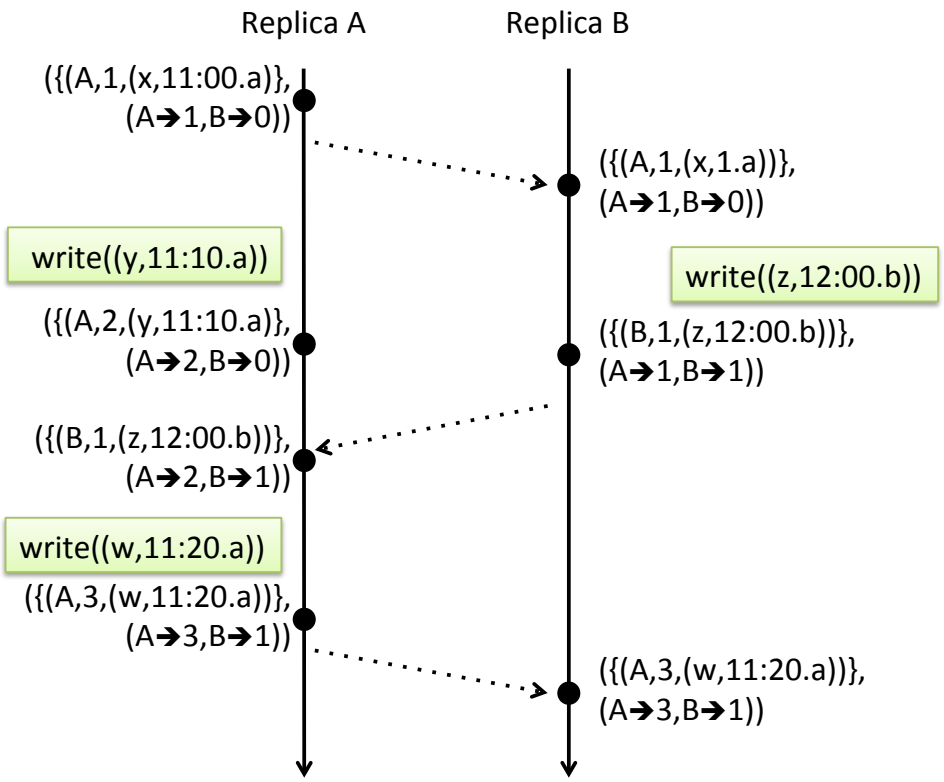

Figure 3: Last-writer-wins run with two register replicas; dashed arrow represents a message, merged at the receiver replica; solid box indicates a client operation.