Brief Announcement: Semantics of Eventually Consistent Replicated Sets *

Annette Bieniusa[1], Marek Zawirski[1,2], Nuno Preguiça[3,1], Marc Shapiro[1], Carlos Baquero[4], Valter Balegas[3], and Sérgio Duarte[3]

[1] INRIA/LIP6, Paris, France
[2] UPMC, Paris, France
[3] CITI, Universidade Nova de Lisboa, Portugal
[4] HASLab, INESC Tec and Universidade do Minho, Portugal

This paper studies the semantics of sets under eventual consistency. The set is a pervasive data type, used either directly or as a component of more complex data types, such as maps or graphs. Eventual consistency of replicated data supports concurrent updates, reduces latency and improves fault tolerance, but forgoes strong consistency (e.g., linearisability). Accordingly, several cloud computing platforms implement eventually-consistent replicated sets [2,4].

The sequential semantics of a set are well known, and are defined by individual updates, e.g., $\{\mathsf{true}\}add(e)\{e \in S\}$ (in "{pre-condition} computation {post-condition}" notation), where S denotes its abstract state. However, the semantics of concurrent modifications is left underspecified or implementation-driven.

We propose the following *Principle of Permutation Equivalence* to express that concurrent behaviour conforms to the sequential specification: *"If all sequential permutations of updates lead to equivalent states, then it should also hold that concurrent executions of the updates lead to equivalent states."* It implies the following behavior, for some updates u and u':

$$\{P\}u; u'\{Q\} \wedge \{P\}u'; u\{Q'\} \wedge Q \Leftrightarrow Q' \;\; \Rightarrow \;\; \{P\}u \parallel u'\{Q\}$$

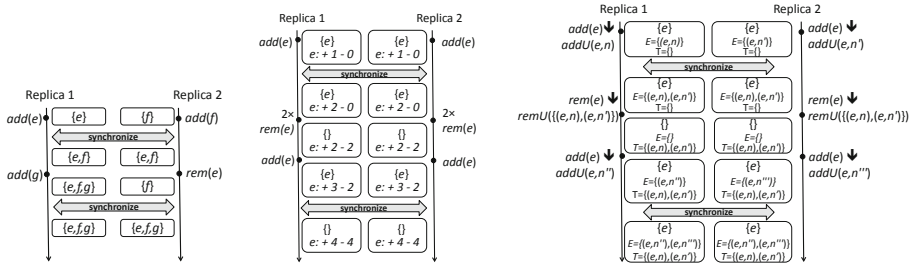
Specifically for replicated sets, the Principle of Permutation Equivalence requires that $\{e \neq f\}add(e) \parallel remove(f)\{e \in S \wedge f \notin S\}$, and similarly for operations on different elements or idempotent operations. Only the pair $add(e) \parallel remove(e)$ is unspecified by the principle, since $add(e); remove(e)$ differs from $remove(e); add(e)$. Any of the following post-conditions ensures a deterministic result:

$$\begin{array}{cc} \{\bot_e \in S\} & \text{– Error mark} \\ \{e \in S\} & \text{– } add \text{ wins} \\ \{e \notin S\} & \text{– } remove \text{ wins} \\ \{add(e) >_{\mathsf{CLK}} remove(e) \Leftrightarrow e \in S\} & \text{– Last Writer Wins (LWW)} \end{array}$$

where $<_{\mathsf{CLK}}$ compares unique clocks associated with the operations. Note that

* This research is supported in part by ANR project ConcoRDanT (ANR-10-BLAN 0208), by ERDF, COMPETE Programme, by Google European Doctoral Fellowship in Distributed Computing received by Marek Zawirski, and FCT projects #PTDC/EIA-EIA/104022/2008 and #PTDC/EIA-EIA/108963/2008.

M.K. Aguilera (Ed.): DISC 2012, LNCS 7611, pp. 449–450, 2012.



(a) Dynamo shopping cart (b) C-Set (c) OR-Set

Fig. 1. Examples of anomalies and a correct design

not all concurrency semantics can be explained as a sequential permutation; for instance no sequential execution ever results in an error mark.

A Study of Existing Replicated Set Designs. In the past, several designs have been proposed for maintaining a replicated set. Most of them violate the Principle of Permutation Equivalence (Fig. 1). For instance, the Amazon Dynamo shopping cart [2] is implemented using a register supporting *read* and *write* (assignment) operations, offering the standard sequential semantics. When two *write*s occur concurrently, the next *read* returns their union. As noted by the authors themselves, in case of concurrent updates even on unrelated elements, a *remove* may be undone (Fig. 1(a)).

Sovran et al. and Asian et al. [4,1] propose a set variant, C-Set, where for each element the associated *add* and *remove* updates are counted. The element is in the abstraction if their difference is positive. C-Set violates the Principle of Permutation Equivalence (Fig. 1(b)). When delivering the updates to both replicas as sketched, the add and remove counts are equal, i.e., e is not in the abstraction, even though the last update at each replica is $add(e)$.

Shapiro et al. propose a replicated set design, called OR-Set, [3] that ensures that concurrent *add*/*remove* operations commute. Unlike the others, it satisfies the Principle of Permutation Equivalence, as illustrated in Figure 1(c). Hidden unique tokens distinguish between different invocations of *add*, making it possible to to precisely track which *add* operations are affected by a *remove*.

References

1. Aslan, K., Molli, P., Skaf-Molli, H., Weiss, S.: C-Set: a commutative replicated data type for semantic stores. In: Int. W. on REsource Discovery, RED (2011)
2. DeCandia, G., Hastorun, D., et al.: Dynamo: Amazon's highly available key-value store. In: Symp. on Op. Sys. Principles, SOSP (2007)
3. Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M.: Conflict-Free Replicated Data Types. In: Défago, X., Petit, F., Villain, V. (eds.) SSS 2011. LNCS, vol. 6976, pp. 386–400. Springer, Heidelberg (2011)
4. Sovran, Y., Power, R., Aguilera, M.K., Li, J.: Transactional storage for geo-replicated systems. In: Symp. on Op. Sys. Principles, SOSP (2011)