SYNCOPE: Automatic Enforcement of Distributed Consistency Guarantees

Kia Rahmani[1], Gowtham Kaki[1], and Suresh Jagannathan[1]

Purdue University, West Lafayette IN 47906, USA
{rahmank,gkaki,suresh}@purdue.edu

Abstract. Designing reliable and highly available large-scale applications, unavoidably requires replication of data over geo-distributed systems, which compromises the simplicity of developing libraries, because it forces designers to weigh the correctness benefits of guaranteeing strong data consistency, against a potentially significant cost in performance. Unfortunately, anomalies possible under weak consistency guarantees can endanger operational correctness of the libraries and forces designers to either implement ad-hoc mechanisms for every operation to avoid these anomalies, or choose to run applications on store-offered stronger levels of consistency, that may well enforce stronger guarantees than required. The former approach introduces unwanted complexity, while the latter sacrifices performance.
In this paper, we describe a lightweight runtime system that is based on declarative axiomatic specifications, which reflect the necessary constraints any correct implementation of anomaly avoidance mechanism must satisfy. Our runtime enforcement mechanism realizes a *provably optimal* approach that imposes no additional communication or blocking overhead beyond what is required to satisfy the specification and allows each operation to run in a *provably safe* environment without witnessing any of the specified anomalies. Experimental results show that the performance of our automatically derived mechanisms are better than both specialized hand-written protocols, and common store-offered consistency guarantees, providing strong evidence of its practical utility.

Keywords: computational geometry, graph theory, Hamilton cycles

1 Introduction

Modern web-based applications are typically implemented as multiple agents simultaneously serving clients, operating over shared data objects replicated across geographically distributed machines. Historically, replication transparency (i.e. requiring distributed systems to *appear* as a single compute and storage server to users), has been the *de facto* system abstraction used to program in these environments. This abstraction has resulted in the development of standardized implementation and reasoning techniques around *strongly consistent* (SC) distributed stores. Although strong notions of consistency, such as *linearizability* and *serializability*, are ideal to develop and reasoning about distributed applications, they come at the price of availability and low-latency. Extensive synchronization overhead often necessary to realize strong consistency is unacceptable for web-scale applications that wish to be "always-on" despite network partitions. Such applications are therefore usually designed to tolerate certain inconsistencies, allowing them to adopt weaker notions of consistency that impose less synchronization overhead. An extreme example is *eventual consistency* (EC), where the application responds to user requests using just the local state of the server to which the client connects; this state is *some* subset of the global state (i.e., it includes an unspecified subset of writes submitted to the application in an unspecified order). Applications that may not tolerate the level of inconsistency imposed by EC strengthen it as needed, resulting in various instantiations that are stronger than EC, but weaker than SC. The term *weak consistency* is a catch-all term used to refer to such application-specific weak consistency guarantees.

Unfortunately, the *ad hoc* nature of weak consistency confounds standardization, with different implementations defining different mechanisms for achieving weakly consistent behavior. Oftentimes, implementations are closely tied to application logic, complicating maintainability and reuse. To illustrate, consider a web application that stores user passwords (encrypted or otherwise) in an off-the-shelf EC data store (e.g., Cassandra [**?**]). The application allows an authenticated user to change her password, following which the current authentication expires, and the user is required to re-login. Now, consider the scenario shown in Fig. **??** where Alice changes her password, and subsequently tries to login with the new password. This involves a write of a new password to the store, followed by a read during authentication. However, because of transient system properties (e.g., load balancing, or network partitions), Alice's write and the read could be served by different replicas of the store, say R_1 and R_2 (resp.), where R_2 may not (yet) contain the latest writes from R_1. Consequently, Alice login attempt fails, even though she types the correct password.

To preempt the scenario described above, applications might want to enforce a stronger consistency guarantee that ensures reads from a client session witness previous writes from the same session. The consistency guarantee, known as *Read-My-Writes/Read-Your-Writes* (RMW/RYW), is one of several well-understood session guarantees [**?**], yet the methods used for its enforcement are often store -and application- dependent. For instance, Oracle's replicated implementation of Berkeley DB suggests application developers implement RMW by querying various metadata associated with writes [**?**]. Each successful write to the store returns a commit token, which is then passed with the subsequent reads to help the store identify the last write preceding the read. The read succeeds only if the write is present at the replica serving the read, failing which the application has to retry the read, preferably after some delay. Fig. **??** illustrates this idea.

The RMW implementation described above already requires considerable re-engineering of the application (to store and pass commit tokens for each object accessed), and conflates application logic with concerns orthogonal to its semantics. On stores that do not admit metadata queries (e.g., Cassandra), the implementation is even more complicated as we describe in Sec. **??**. Moreover, applications sometimes require different consistency guarantees for different objects. In such cases different enforcement mechanisms must be developed, forcing developers to simultaneously reason about their respective properties *in conjunction with* the application state. This is clearly an onerous task. Other alternatives such as forgoing application integrity, or resorting to strong consistency, sacrifice correctness or availability, both unappealing options.

In this paper, we propose an alternative to the aforementioned approaches that overcomes their weaknesses. SYNCOPE is a lightweight runtime system for Haskell that allows application developers to take advantage of weak consistency without having to re-engineer their code to accommodate consistency enforcement logic. The key insight that drives SYNCOPE's design is that the hardness of reasoning about the integrity of a distributed application stems from conflating application logic with the consistency enforcement logic, reasoning about both *operationally*. By separating application semantics from consistency enforcement semantics, admitting operational reasoning for the former, and declarative reasoning for the latter frees programmers from having to worry about implementation details of consistency guarantees, and instead focus on reasoning about application semantics under the assumption that specified consistency guarantees are automatically enforced by the data store runtime. Our approach admits declarative reasoning for consistency enforcement via a specification language that lets programmers formally specify the consistency requirements of their application. The design of our specification language is based on the observation that various forms of weak consistency guarantees differ only in terms of how and what they mark as *dependencies* among operations. When all dependencies are present on a replica to an operation, then the effect of the operation is guaranteed to be correct with respect to the specification. For example, RMW marks all previous writes from the same session as dependencies of subsequent read operations, so an RMW read succeeds only if all the previous writes are visible (i.e., present on the replica on which the read is performed) . A different consistency guarantee (e.g., *Monotonic Reads* (MR)) imposes a different set of dependencies, as do various combinations of (e.g., MR+RMW). By allowing a runtime system to monitor dependencies defined by consistency specifications, we realize a generic weak consistency enforcement mechanism framework. Such a runtime, working in tandem with a consistency specification language, contributes to the novelty of our approach.

A summary of our contributions is given below:

– We propose a consistency specification language that lets programmers express the consistency requirements of their applications in terms of the dependencies between operations.
– We describe a generic consistency enforcement runtime that analyzes an operation's consistency specification, and ensures that its dependencies are satisfied before it is executed. We formalize the operational semantics of the

runtime, and prove its correctness and optimality guarantees. Optimality includes *minimum wait*, which guarantees that an operation waits (on arriving communication) only until its dependencies are satisfied, and *minimum staleness*, which guarantees that among various states that satisfy an operation's dependencies, operation witnesses the latest state.

– We describe an implementation of our specification language and consistency enforcement runtime in a tool called SYNCOPE, which works on top of an off-the-shelf EC data store. We evaluate SYNCOPE over realistic applications and microbenchmarks, and present results that demonstrate the performance benefits of making fine-grained distinctions between consistency guarantees, and the ease of doing so via our specification language.

The remainder of the paper is organized as follows. The next section presents a system model that describes key notions of consistency and replication. Sec. **??** provides additional motivation. Sec. **??** introduces an abstract store model that forms the basis of our specification language, along with the language itself. We describe the semantis of our consistency enforcement runtime, and formalize its correctness and optimality guarantees in Sec. **??**. Sec. **??** elaborates on the algorithmic aspects of our runtime that is key to its efficient realization. Sec. **??** describes SYNCOPE, an implementation of the specification language and enforcement mechanism, and evaluates its applicability and practical utility. Related work and conclusions are presented in Sec. **??**.

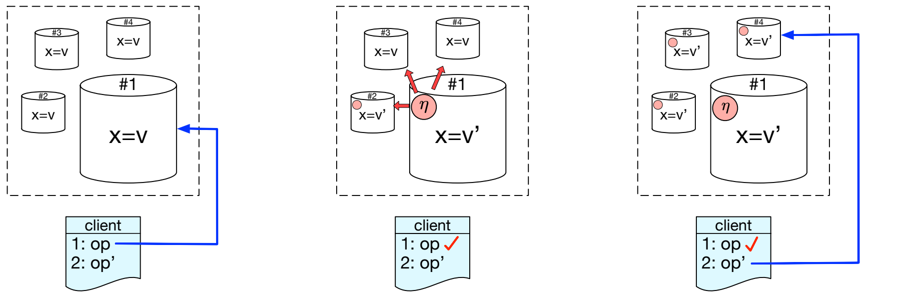
2 System Model

A data store in our system model is a collection of *replicas* (#1,#2,...), each of which maintains an instance of a set of replicated *data object* (x,y,...). These objects, which are defined by application developers, contain a *state* (v,v',...) and are equipped with a set of *operations* (op,op',...), each of which are designated as either read-only or effectful. The former characterizes operations that just read from the store, and receive an instance of an object's state, while the latter characterizes operations that modify an object's state by generating *update effects* (η,η',...). Update effects or simply effects are associated with an *apply* function that executes asynchronously, and modifies object state on different replicas. An effectful operation is handled by a replica that updates the object's state, and guarantees eventual delivery of the effect to all other replicas in the system. Each recipient uses the apply function to modify its local instance of the object; Fig.1 illustrates this behavior.

Observe there is no direct synchronization between replicas when an operation is executed, which means there can be conflicting updates on an object, that are generated at different replicas concurrently. We do not bound the system to a particular conflict resolution strategy Consequently, this model admits all inconsistencies and anomalies associated with eventual consistency [**?**]; our goal is equip applications and implementations with mechanisms to specify and avoid such inconsistencies.

Clients interact with the store by invoking operations on objects. A *session* is a sequence of operations invoked by a particular client. Consequently, operations (and update effects) can be uniquely identified by their invoking *session id* and their *sequence number* in that particular session. The data store is typically accessed by a large number of clients concurrently and as a result of the load balancing regulations of the store, operations might be routed to different replicas, even if they are from the same session (Figures 1a and 1c). Operations belonging to a given session are not required to be handled by the same replica.

We new define two relations over effects created in the store. *Session order* ($\xrightarrow{\mathsf{so}}$) is an irreflexive, transitive relation that relates effects from the same session following the integer *smaller than* relation over their sequence numbers. *Visibility* ($\eta \xrightarrow{\mathsf{vis}} \eta'$) is an irreflexive and assymetric relation that holds if effect η has been applied to a replica R before η' is applied to R. In Fig.1c for example, η' (the effect of executing op') will witness the updates associated with effect η ($\eta \xrightarrow{\mathsf{vis}} \eta'$), since η is already present at the replica4, when op' is submitted).



(a) A client submits an operation op to the store, which is routed to the replica1.

(b) The state of the object at replica1 is updated, an effect is created and is being propagated.

(c) second operation op' is submitted to the store, which is this time routed to the replica4.

Fig. 1: system model of SYNCOPE

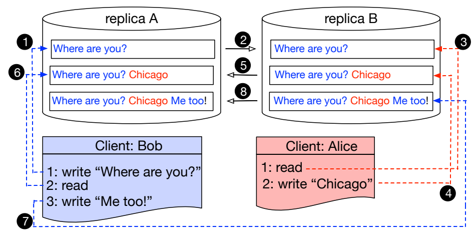
```haskell
type Effect = String
type State =  String

read :: State -> (String,Maybe Effect)
read s = (s,Nothing)

write :: String -> ((),Maybe Effect)
write comment = ((),comment)

apply :: State -> Effect -> State
apply s eff = in s ++ " - " ++ comment
```

(a) A simple implementation



(b) Example execution

Fig. 2: A distributed application for comment section management

3 Motivation

3.1 Replicated Data Types (RDTs) in Eventually Consistent Stores

To provide further motivation, consider a highly available (low latency) application that manages the comment section of posts, as part of a photo sharing web site. Fig. 2a presents a simple Haskell implementation of such an application cognizant of our system model. The implementation treats `Effect` strings as the text of a comment, and `State` strings as the concatentation of all visible comments associated with a post. A new `Effect` is generated every time a user wants to comment on a post by calling the `write` function, and a `read` call simply returns the `State` of the object at the serving replica.

The `apply` function given an effect pastes the included comment to the end of the current `State`. For perspicuity, we omit any conflict resolution strategy in the code; however, developers (using roll-backs, etc) can design the `apply` function to resolve conflicting concurrent updates as they desire.

Fig. 10a presents an example of how users interact with the application. The example shows two clients, Bob and Alice, that invoke operations on the same

```
 1 data Sess = Bob | Alice
 2 type ID = (Sess,Int)
 3 type Effect= (ID,String)
 4 type State = (String,Int,Int)
 5
 6 read :: ID -> State -> String
 7 read (sess,seq) (st,sq1,sq2) =
 8  case sess of
 9   Bob ->    if (seq==sq1+1) then st
10             else read (sess,seq)(st,sq1,sq2)
11   Alice -> if (seq==sq2+1) then st
12             else read (sess,seq)(st,sq1,sq2)
13 apply :: State -> Effect -> State
14 apply (st,sq1,sq2) ((sess,seq),cm) =
15   case sess of
16     Bob ->    if (sq1==seq-1)
17               then (st++cm,sq1+1,sq2)
18               else (st,sq1,sq2)
19     Alice -> if (sq2==seq-1)
20               then (st++cm,sq1,sq2+1)
21               else (st,sq1,sq2)
```

Fig. 3: Guarded Application to Prevent Lost-updates Anomaly When Serving Bob and Alice

object. In the beginning Bob writes a comment, which is routed to replica A (❶), whose effect is then propagated and delivered to replica B (❷); Alice's first read operation is routed to next (❸). Alice and Bob then keep talking through more read and write events while updates are propagated between replicas, whose order is marked in the figure.

Assume Bob's read operation, instead of being sent to replica A, was routed to another replica C, where the update from his first operation was not present. This is known as a *lost-updates* anomaly, a very well-understood albeit undesirable behavior that is admitted by eventually consistent stores. Preventing this sort of anomaly requires subtle reasoning and may entail sophisticated restructuring of application logic.

3.2 Ad-hoc Anomaly Prevention

One way to prevent a lost-update anomaly is to *tag* effects and operations with unique identifiers; such identifiers consist of an originating session (Alice/Bob), and an integer showing a sequence number in the session ($line : 1, 2, 3$). This is used by replicas to record the set of effects they have witnessed. Tracking dependencies in this way prevents the anomaly. Operations that are routed to a replica that have not witnessed their dependencies are postponed.

Another technique called **filtration** is also used to further realize the above idea, by separating the set of effects that have arrived to the replica (available effects), and the set of effects that have arrived and also been applied to the state (filtered effects). Replicas can filter the set of available effects before applying them, so that effects are applied only when all previous effects, as determined by session order, have been applied. In order to maintain the set of all effects applied to the state, replicas should only record the highest sequence numbers from each session since it is guaranteed that smaller ones are also already applied ($line : 4$).

Fig. 3 represents the blocking technique in the modified `read` operation, where the result is only returned if the sequence number of the operation is one larger than the previously seen sequence number from that session. ($line : 9, 11$). Otherwise, the function is blocked by calling itself recursively ($line : 10, 12$).

The above approach obviously requires fundamental and pervasive changes to the original code. Additionally, modifications are heavily tangled with application logic complicating reasoning and hampering correctness arguments.

Another major drawback of this approach is that it requires constant alteration to the state of the application when sessions come and go. Applications are now required to make sure that a new field is created locally *and* globally when new sessions are connected. This requires modifying object state in the data store, and making sure that the global information is updated before allowing local sessions to start. This requires direct synchronization between replicas, degrading application performance and availability.

To make the matter worse, new anomalies are constantly found in systems requiring developers to develop new non-trivial solutions. For example, in the above application, another type of anomaly can occur when a third user Chris, uses the application and submits a read, which is routed to a replica D, that only contains the last write from Bob. Then Chris sees a window containing "Me too!", which is an undesirable behavior. Developers must now either find another ad-hoc solution, further polluting application logic, or execute the application using a stronger form of consistency, compromising performance.

3.3 An Alternative

To overcome these issues, we consider the design of a generic consistency management tool. Our goal is to have developers define a consistency level for each operation *a priori*, ensuring their satisfiability at runtime. We have realized this idea, following our observation that preventing anomalies can be expressed in terms of filtration and blocking.

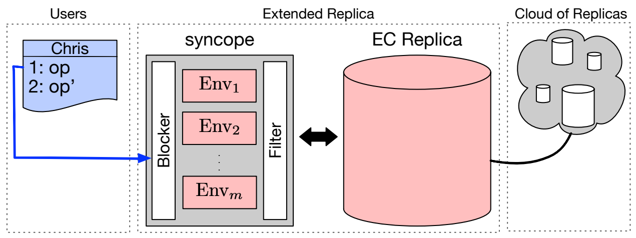


Fig. 4: SYNCOPE's outline

We therefore propose equipping a distributed data store with a filtration mechanism that regulates the effects an operation witnesses (e.g., preventing Chris from seeing Bob's last comment without including the causal history that preceded it), and a blocking mechanism that allows operations to execute only when all its dependent effects have been recorded (e.g., thereby preventing the lost updates anomaly). We refer to the view an operation has of a replica's state as its *environment*.

Our implementation (called SYNCOPE) requires developers to specify constraints on read operations that can be used to synthesize appropriate filtration and blocking mechanisms. The specification language is seeded with so vis relations and allows users to proscribe different anomalous behaviors. For example, the followings are the two constraints that prevent the anomalies described above: the two anomalies mentioned in this section:

$$\psi_1 : \forall(\eta,\eta').\ \eta \xrightarrow{\mathsf{so}} \eta' \quad \Rightarrow \eta \xrightarrow{\mathsf{vis}} \eta'$$
$$\psi_2 : \forall(\eta,\eta').\ \eta \xrightarrow{\mathsf{vis;vis}} \eta' \Rightarrow \eta \xrightarrow{\mathsf{vis}} \eta'$$

The specification of the first constraint eliminates the possibility of lost updates by mandating that any operation *op* with effect η that precedes another *op'* with effect η' in session order must also be visible (i.e., η must be witnessed on any replica that *op'* executes on). The second specification prevents causality violations by demanding that if an effect η is visible to another η' , then η' should also witness any effects η has witnessed.

4 Specification Language

The formal syntax of our specification language is presented in Fig. 5. The language allows the definition of propositions, FOL formulae that establish the conditions under which one effect may become visible to another. The dependencies that must hold for a particular visibility effect to be valid is given in terms of a composition of

$$
\begin{array}{lll}
\mathtt{r} \in \mathtt{rel.seed} & := \mathsf{vis} \ \mid \ \mathsf{so} \ \mid \ \mathtt{r} \cup \mathtt{r} \\
\mathsf{R} \in \mathtt{relation} & := \mathtt{r} \ \mid \ \mathsf{R};\mathtt{r} \ \mid \ \mathtt{null} \\
\pi \in \mathtt{prop} & := \forall a.\ a \xrightarrow{R} \hat{\eta} \ \Rightarrow \ a \xrightarrow{\mathsf{vis}} \hat{\eta} \\
\psi \in \mathsf{spec} & := \pi \ \mid \ \pi \ \wedge \ \pi
\end{array}
$$

Fig. 5: Syntax of the Specification Language

The type `relation` which is used to define dependencies between effects in `prop`s, is a sequence of `rel.seed`s where each of them is a vis or so relation or the union of them both over the set of effects. In order to configure each environment, the developers are required to write a `spec` for each of them, that is consisted of conjunctions of `prop`s, which simply allows developers to eliminiate multiple types of anomalous behavior from each environment. For example, the developers of the comment management application, can simply write $\psi_1 \wedge \psi_2$ for their `read` operation and prevent *both* anomalies explained in the previous section.

Now we syntactically classify propositions into *lower bound* (LB) and *upper bound* (UB) and hybrid types, and show that they completely align with types of anomalies previously mentioned.

- **LB**: We define a `prop` to be of type LB, if its dependency relation R, ends with an so relation, i.e. is of the following from: $\forall a.a \xrightarrow{r_1;r_2;\ldots;\mathsf{so}} \hat{\eta} \Rightarrow a \xrightarrow{\mathsf{vis}} \hat{\eta}$. Observe that these contracts simply put a lower bound on the set of effects each operation must witness, by defining a certain set of dependency for each effect, that must be visible to it, which makes the blocking technique very suitable for easily maintaining them.
- **UB**: These propositions are similarly defined as `prop`s with dependency relations ending with vis, or of the following form: $\forall a.a \xrightarrow{r_1;r_2;\ldots;\mathsf{vis}} \hat{\eta} \Rightarrow a \xrightarrow{\mathsf{vis}} \hat{\eta}$. This type of `props`, put an upper bound on the set of effects a replica should

make visible to each operation when executing it, by enforcing that if an effect is made visible, certain set of dependent effects must also be made visible. This clearly resembles the filteration technique, where only a subset of available effects can enter the consistent environments.
- **Hybrid**: The `prop`s whose dependency relation ends with $\mathsf{vis} \cup \mathsf{so}$, which require both blocking and filteration to be maintained.

We can extend the above definitions to `spec`s also, by defining an LB (UB) `spec` to contain only LB (UB) `prop`s. Hybrid `spec`s are also defined as the ones that are neither LB or UB.

We finish this section by presenting weak consistency guarantees from Terry et. al. in figure **??**, which shows the generality of our simple specification language. We also present a simple way of representing contracts as graphs where the left-hand-side of the contracts are depicted as the sequence of edges relating two effects, and the right-hand-sinde (or what must be enforced by replicas at the execution time) is represented as the single dashed vis edge.

5 Semantics

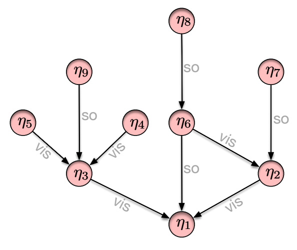
In this section we formalize our consistency enforcement algorithm with an operational semantics, which is also a high-level abstraction of our tool SYNCOPE. Our approach is complete for the specification language defined in section 4, however, here for simplicity reasons we present an operational semantics parametrized over a non-hybrid contract with a single prop. As we will explain in section 5.5, the rules can be easily generalized to cover multiple consistency levels, each specified by any given contract. Therefore, in the rest of this section we will assume a given contract ψ of the following form:

$$\psi = \forall a.a \xrightarrow{R} \hat{\eta} \Rightarrow a \xrightarrow{\mathsf{vis}} \hat{\eta} \qquad\qquad R = r_1; r_2; ...; r_k$$

The operational semantics is defined via a small-step relation over *execution states*, which are tuples of the form $\mathsf{E} = (\mathsf{A}, \mathsf{vis}, \mathsf{so})$. The *effect soup* A, represents the set of all effects produced in the system, and $\mathsf{vis}, \mathsf{so} \subseteq \mathsf{A} \times \mathsf{A}$, respectively stand for the visibility and session order relations among such effects. Figures 6a and 6b represent a simple execution state of a system consisting of 9 effects with associated primitive relations, where we ommited drawing the transitive so edge between η_8 and η_1, for better readability. We use notation $\mathsf{A}_{(condition)}$ to represent a subset of A consisting of effects that satisfy the specified condition. Note that SYNCOPE's contracs are in fact constraints over execution states, where the domain of quantification is fixed to the effect soup A, and interpretation for so and vis relations (which occur free in the contract formulae) are also provided. Thus, execution states are potential models for any first-order formula expressable in the contract language. If an execution state E is in fact a valid model for a contract ψ, we say that E satisfies ψ, written as $\mathsf{E} \models \psi$.

The samentics' reduction step is of the form

$$(\mathsf{E}, \mathsf{op}_{<s,i>}) \xrightarrow{\mathsf{V}} (\mathsf{E}', \eta),$$



(a) An execution state E

$$
\begin{aligned}
\mathsf{E.A} &= \{\eta_1, \eta_2, \eta_3, \eta_4, \eta_5, \\
&\quad \eta_6, \eta_7, \eta_8, \eta_9\} \\
\mathsf{E.vis} &= \{(\eta_5, \eta_3), (\eta_4, \eta_3), \\
&\quad (\eta_3, \eta_1), (\eta_2, \eta_1), \\
&\quad (\eta_6, \eta_2)\} \\
\mathsf{E.so} &= \{(\eta_9, \eta_3), (\eta_8, \eta_6), \\
&\quad (\eta_6, \eta_1), (\eta_8, \eta_1), \\
&\quad (\eta_7, \eta_2)\}
\end{aligned}
$$

(b) Effect soup and primitive relations

$$
\begin{aligned}
\mathsf{vis}^{-1}(\eta_1) &= \{\eta_2, \eta_3\} \\
\mathsf{so}^{-1}(\eta_1) &= \{\eta_6, \eta_8\} \\
(\mathsf{so} \cup \mathsf{vis})^{-1}(\eta_1) &= \{\eta_2, \eta_3, \eta_6, \eta_8\} \\
(\mathsf{vis}^*)^{-1}(\eta_1) &= \{\eta_2, \eta_3, \eta_4, \eta_5, \eta_6\} \\
(\mathsf{so}; \mathsf{vis})^{-1}(\eta_1) &= \{\eta_7, \eta_9\}
\end{aligned}
$$

(c) Relation inverse examples

Fig. 6: A simple execution state

which can be interpreted as a reduction of the initial execution state E, initiated by a replica with a local set of effects V, after it executes an operation op, which is the i^{th} request from the session s. During this reduction step a new effect η is produced and added to the system, resulting a new execution state E' with updated effect soup and primitive relations.

Before introducing the operational semantics, we will first formally present the required definitions in the next section. Namely, we will define notions of the *inverse* of a given relation R, and the *maximally closed subset* of a given set of effects V under a contract ψ.

5.1 Preliminaries

We start by formally defining the inverse of a seed relation ($\mathtt{r} \in \mathtt{rel.seed}$) given an execution state E:

$$
\mathtt{r}^{-1}(\mathsf{S}) = \begin{cases} \bigcup\limits_{b \in \mathsf{S}} \{a | (a, b) \in \mathsf{E}.\mathtt{r}\} & \mathtt{if}\ \mathtt{r} \in \{\mathsf{so}, \mathsf{vis}\} \\ \mathtt{r}_1^{-1}(\mathsf{S}) \cup \mathtt{r}_2^{-1}(\mathsf{S}) & \mathtt{if}\ \mathtt{r} = \mathtt{r}_1 \cup \mathtt{r}_2 \end{cases} \tag{1}
$$

Note that when the input of an inversed relation is a singleton $\{\eta\}$, we drop the brackets and simply write it as $\mathtt{r}^{-1}(\eta)$. We now present the definition of the inverse of sequences (of size larger than 1) of $\mathtt{rel.seed}$ as follows:

$$
b \in (R'; r)^{-1}(a) \iff \exists c. c \in r^{-1}(a) \wedge b \in (R')^{-1}(c) \tag{2}
$$

Inverse of sequences of length 1 is also implicitely defined as the inverse of the enclosed $\mathtt{rel.seed}$.

Following definitions (1) and (2), since $\mathtt{relation}$ in our specification language is defined as either a $\mathtt{rel.seed}$, or a sequence of them, we are now ready to formally define the inverse of any given relation $R \in \mathtt{relation}$. However, note that the definition (2) fails to capture the reality of distributed systems, where all computations are done locally by replicas, which might have access to only a *subset of all effects* at any given moment. For example, consider $(\mathsf{so}; \mathsf{vis})^{-1}(\eta_1)$ of the execution state in figure 6. In order to compute this set, based on the recursive defintion of (2) we have:

$$
b \in (\mathsf{so}; \mathsf{vis})^{-1}(\eta_1) \iff \exists c. c \in \mathsf{vis}^{-1}(\eta_1) \wedge b \in (\mathsf{so})^{-1}(c)
$$

Since there exist *mid-level* effects η_2 and η_3, such that satisfy the above definition respectively for $b = \eta_7$ and $b = \eta_9$, we have: $(\mathsf{so};\mathsf{vis})^{-1}(\eta_1) = \{\eta_7, \eta_9\}$. Now assume a replica only contains $\{\eta_1, \eta_6, \eta_7, \eta_8, \eta_9\}$ and wants to check if the dependencies of η_1 are locally present or not. Even though based on the above definitions the answer is yes (since the replica does contain $\{\eta_7, \eta_9\}$), but in reality the replica would not be able to verify that, since the mid-level effects η_2 and η_3 are not present at the replica yet.

To capture the above property, we now present partial definition of the inverse of a given relation $R \in \texttt{relation}$ *according to a set of available effects* V. We define the inverse, only if all the required mid-level effects are present in V using definition (1) and a slightly different version of the definition (2).

$$b \in R_V^{-1}(a) \iff \begin{cases} \bot & \texttt{if} \;\; R = \texttt{null} \\ b \in \texttt{r}^{-1}(a) & \texttt{if} \;\; R = \texttt{r} \\ \exists c.c \in \texttt{r}^{-1}(a) \wedge b \in (R')^{-1}(c) \wedge \texttt{r}^{-1}(a) \subseteq V & \texttt{if} \;\; R = R';\texttt{r} \end{cases} \tag{3}$$

Note that the only difference between the third case in above definition and the definition (2), is the last conjunct which is added to ensure the presence of mid-level effects before performing the next recursive call.

Now, we define $\mathsf{trunc}()$ as a function that given $R \in \texttt{relation}$, removes the last element from the sequence (if there is any) in R, i.e.

$$\mathsf{trunc}(R) = \begin{cases} \texttt{null} \;\texttt{if}\; R = \texttt{r} \quad \texttt{or} \quad R = \texttt{null} \\ R' \quad \texttt{if}\; R = R';\texttt{r} \end{cases} \tag{4}$$

Finally, we define *closed subsets* of a given set of effects V under the contract ψ, which the maxiamal element among such subsets is also defined next[1]:

$$\begin{array}{rl} \texttt{closed subsets} : V' \in \lfloor V \rfloor & \iff V' \subseteq V \quad \wedge \; (\mathsf{trunc}(R))_V^{-1}(V') \subseteq V' \\ \texttt{maximally closed subset} : V' = \lfloor V \rfloor_{\texttt{max}} & \iff V' \in \lfloor V \rfloor \; \wedge \; \nexists V'' \in \lfloor V \rfloor \,.|V''| > |V'| \end{array} \tag{5}$$

5.2 Core Operational Semantics

In this part we present the reduction rules, representing our consistency preservation approach. Figure 7 presents the set of rules defining the auxiliary relation ($\hookrightarrow$) and small-step reduction relation ($\rightarrow$) over executions. The latter relation is parametrized over a set V, that represents the set of effects that are available at the replica taking the step. Obviously V must be a subset of the effect soup of the initial execution, however, there is no other restrictions on V, since we only assume eventual consistency at the underlying store.

The rule [OPER] represtns the procedure of producing a new effect η, by witnessing a set of effects S. An effect is formally defined as a tuple $\eta = (s, op, v)$, representing the session and the operation name whose execution created η, and the value that the replica returns as the response to that operation. The rule explains how the execution state changes after producing an effect at a replica.

[1] We abuse the previously defined notation slightly and use a *set* of effects as the input to the inverse of $R \in \texttt{relation}$, which simply means the union of the results of apply the function for all the effects in the input set

Auxiliary Definitions

$op \in$ `Operation Name`	$v \in$ `Return Value`	$s \in$ `Session Id`	
η	$\in$ `Effect`	$:=$	(s, op, v)
F_{op}	$\in$ `Op. Def.`	$:=$	$\mathcal{P}(\eta) \mapsto v$
A	$\in$ `Eff Soup`	$:=$	$\mathcal{P}(\eta)$
$\mathsf{vis}, \mathsf{so}$	$\in$ `Relations`	$:=$	$\mathcal{P}((\eta, \eta))$
E	$\in$ `Exec State`	$:=$	$(\mathsf{A}, \mathsf{vis}, \mathsf{so})$

Auxiliary Reduction

$$\boxed{S \vdash (\mathsf{E}, op_{<s,i>}) \hookrightarrow (\mathsf{E}', \eta)}$$

[Oper]

$$\frac{\begin{array}{c} S \subseteq \mathsf{A} \qquad F_{op}(S) = v \qquad \eta \notin S \qquad \eta = (s, op, v) \\ \mathsf{A}' = \mathsf{A} \cup \{\eta\} \qquad \mathsf{vis}' = \mathsf{vis} \cup S \times \{\eta\} \qquad \mathsf{so}' = \mathsf{so} \cup \{(\eta', \eta) \mid \eta' \in \mathsf{A}_{(\mathsf{SessID}=s)}\} \end{array}}{S \vdash ((\mathsf{A}, \mathsf{vis}, \mathsf{so}), op_{<s,i>})) \hookrightarrow ((\mathsf{A}', \mathsf{vis}', \mathsf{so}'), \eta)}$$

Operational Semantics

$$\boxed{(\mathsf{E}, op_{<s,i>}) \xrightarrow{V} (\mathsf{E}', \eta)}$$

[UB Exec]

$$\frac{\begin{array}{c} \mathsf{vis} \subseteq r_k \quad V \subseteq E.A \quad V' = \lfloor V \rfloor_V \\ V' \vdash (E, op_{<s,i>})) \hookrightarrow (E', \eta) \end{array}}{(\mathsf{E}, op_{<s,i>}) \xrightarrow{V} (\mathsf{E}', \eta)}$$

[LB Exec]

$$\frac{\begin{array}{c} \mathsf{vis} \nsubseteq r_k \quad V \subseteq E.A \quad R_V^{-1}(\eta) \subseteq V \\ V \vdash (E, op_{<s,i>})) \hookrightarrow (E', \eta) \end{array}}{(\mathsf{E}, op_{<s,i>}) \xrightarrow{V} (\mathsf{E}', \eta)}$$

Fig. 7: Core Operational semantics of a replicated data store.

Specifially, in the new state, the effect soup A' contains the newly created effect η, and the relations vis' and so' capture the fact that all effects in the set S were made visible to η, and all effects from the same session that were already presenet in the intial execution state, should be in session order with η in the final execution state.

Now we explain the rules for reduction relation ($\xrightarrow{V}$), starting with [ub exec], which represents the execution of operations in a replica that updates the global state and produces a new effect under a UB contract. The rule requires operations witnessing only the maximally consistent subset V' of the local set of available effects V. In other words, the rule filters out the effects that may result anomalies and shows the safe environment to the operation.

The next rule, [lb exec], represents the step taken when an operation is performed under an LB contract. The precondition $R_V^{-1}(\eta) \subseteq V$ in the rule, ensures that the reduction happens only if the effects necessary to avoid the specified anomaly are present in V. The operations performing under an LB contract must be blocked, untill all the necessary effects (and possibly required mid-level effects) become available in the locally available set of effects V. Note that in this case effects are not filtered out, and the operation witnesses all effects in set V.

5.3 Soundness

In order to prove a meta-theoretic correctness property for our semantics, we first define a ψ-consistent set of effects S given a execution state E as follows:

$$S \texttt{ is } \psi\texttt{-consistent} \iff \forall(\eta \in S).\forall(a \in \mathsf{E.A}).R(a,\eta) \Rightarrow a \in S \tag{6}$$

Where the definition of R is based on the definition of R^{-1} from subsection 5.1:

$$R(a,b) \iff a \in R^{-1}_{E.A}(b) \tag{7}$$

Theorem 1. *For all reduction steps* $(\mathsf{E}, op_{<s,i>}) \xrightarrow{V} (\mathsf{E}', \eta)$*, if* V *is* ψ*-consistent under* E*, then:*

$$\begin{array}{ll}(i) & V \cup \{\eta\} \texttt{ is } \psi\texttt{-consistent under } E' \\ (ii) & E' \models \psi[\eta/\hat{\eta}]\end{array}$$

Proof. Appendix A.1

5.4 Optimality

Now we will present two theorems, regarding the optimality of our approach. Firstly, we show that there cannot be a larger subset of local effects in a replica to be made visible to an operation, than what our reduction rules specify. Next, we will prove the liveness property of our semantics, which guarantees that all replicas will take a step, given the proper set of local effects. Considering the eventual delivery of all updates, which is guaranteed by the underlying store, this theorem guarantees that our system would never get permanently stuck.

Theorem 2. *For all operation executions* $(\mathsf{E}, op_{<s,i>}) \xrightarrow{V} (\mathsf{E}', \eta)$*, the set of effects made visible to* η *is maximal. i.e. for all execution states* E'' *such that:* $(\mathsf{E}, op_{<s,i>}) \xrightarrow{V} (\mathsf{E}'', \eta)$ *then* $(\mathsf{E}''.\mathsf{vis}^{-1}(\eta)) \subseteq (\mathsf{E}'.\mathsf{vis}^{-1}(\eta))$

Proof. The proof is directly followed by the fact that the subset of effects made visible to each operation is $\lfloor V \rfloor_V$ which is by definition maximal. The detailed proof can be found in appendix A.2.

Theorem 3. *For all execution states* E*, if there exists* $(S \subseteq E.A)$ *such that:*

$$S \vdash (\mathsf{E}, op_{<s,i>}) \hookrightarrow (\mathsf{E}', \eta) \quad \wedge \quad (\mathsf{S} \cup \{\eta\} \texttt{ is } \psi\texttt{-consistent under } E')$$

then there exist E''*,* η' *and* $V \subseteq E.A$ *such that:* $((\mathsf{E}, op_{<s,i>}) \xrightarrow{V} (\mathsf{E}'', \eta'))$

Proof. The proof is given by choosing set V to be equal to S, and then considering two cases, where either S or $\lfloor S \rfloor_S$ are made visible to operations, if the contract is respectively LB and UB (or hybrid). In both cases, all premises of taking an step are satisfied. A detailed proof can be found in appendix A.3.

5.5 Generalization

We finish this section by extendeding our ideas in two dimentions. We will first explain how to handle an arbitrary contract ψ of the following form:

$$\psi = \pi_1 \wedge \pi_2 \wedge \ldots \wedge \pi_m \qquad\qquad \pi_i = \forall(a,b).a \xrightarrow{R_i} b \Rightarrow a \xrightarrow{\mathsf{vis}} b$$

Later, we will explain how to maintain multiple levels of consistency simultaneously, each of which is defined for a different operation name. We will assume an arbitrary contract ψ_{op} for every user-defined operation op, and explain how to modify our system model to preserve them all.

To begin with, as we mentioned earlier, all propositions in our specification language, either put a maximal or a minimal bound on the subset of local effects to be made visibe to each opreation. This simply means that when the system is given a conjunction of propositions, it should define the such subsets in a way, so it would not violate *any* of them. Therefore, by a few modifications we can extend the system to support all contracts. Firstly, the single premise $R_V^{-1}(\eta) \subseteq V$ in the reduction rule should be replaced with the following conjunction:

$$\bigwedge_{1 \le i \le m} (R_i)_V^{-1} \subseteq V$$

Secondly, the definition of the maximal closed subset of local effects must also be modified to a subset that is closed under *all* given relations:

$$\lfloor S \rfloor_V = S' \iff S' \subseteq S \;\wedge\; \bigwedge (R_i)_V^{-1}(S') \subseteq S' \;\wedge\; \nexists S''.(\bigwedge((R_i)_V^{-1}(S'')) \subseteq S'' \wedge |S''| > |S'|)$$

Moreover, for modifying the system to handle multiple contracts simultaneously, we can extend the local effect set V, to a sequence of sets V_{op_i}, each maintaning the consistency level for an operation type op_i. Now we define the modified form of execution steps as follow:

$$(\mathsf{E}, \mathsf{op}_{<s,i>}) \xrightarrow{V_{\mathsf{op}}} (\mathsf{E}', \eta)$$

The local effect set V must also be replaced with V_{op_i} in the premises of the reduction rules, so each operation of type op_i would only witness the associated subset for its own consistency requirements. This abstractly represents our implementation, in the sense that all operations work only on a specific subset of available effects at any replica. The subset, is maintained according to the contract assosiated with each operation, and is guaranteed to preserve the consistency requirements following the theorems of sections 5.3 and 5.4.

6 Algorithm

In this section, we present a detailed and practical implementation strategy of the operational semantics presented in section 5, where we introduced an abstract outline of our consistency preservation technique. Here we realize our ideas by equipping each replica with a *cache*, that is guaranteed to preserve a specified consistency level. Here we assume a contract of the form $\psi = \forall(a,b).a \xrightarrow{R} b \Rightarrow$

$a \xrightarrow{\mathsf{vis}} b$ and a replica containg a local set V and explain SYNCOPE's behavior in more detail.

Let's first define a *truncated relation* as the relation derived by removing the last element from a given relation:

$$\mathsf{trunc}(r_1; r_2; ...; r_k) = r_1; r_2; ...; r_{k-1}$$

We now extend the above definition to the given contract ψ, by replacing R with $\mathsf{trunc}(R)$ and argue that an effect η can only enter the cache, if its presence would not violate the truncated contract in the replica, i.e. η's dependency set $(\mathsf{trunc}(R))_V^{-1}(\eta)$ is already present in the cache (replica) for a UB (LB) contract. Now we consider two possible types of contracs and explain the replicas' behavior when an operation is submitted:

1. LB contracts: In this case, the replica makes sure that the operation is blocked until effects of earlier operations from the same session, are already present in the cache. This guarantees the presence of all the dependencies of the current operation, accoring to the *original* contract. Note that in this case, the operation can witness *all* effects at the replica.
2. UB contracts: In this case, operations are not blocked, however, they should only witness the effects that are already present in the cache. This guarantees the preservation of the original contract, which puts a maximal bound on the set of effects to be made visible to an operation.
3. Hybrid Contracts: Here, the oeprations should both be blocked similar to the LB case, and also witness only the effects in the cache.

6.1 Degree of Dependency Presence

In the above description of our consistency management tool, the notion of *the presence of the depency set* is treated as a true/false property, that is checked before allowing an effect enter the cache. However, as an astute reader might have noticed, a naive implementation of this idea, could result in poor performance. That is because contracts in our specification language can be arbitrarily large and might contain closures of relations, computing the inverse of which can become very large. A naive implementation that drops all the computations done for a failed dependency check of an effect, results in redunencies the next time the same property is being validated, which is not practically reasonable.

To address the mentioned difficulties, we introduce the *Degree of Dependency Presence*, *DDP*, that extends the above binary property, by marking the effects with a number, that represents how far the presence of their dependencies have been checked so far. The *DDP* of an effect according to a relation $R = r_1; r_2; ...; r_k$ and a given set of effects V is defined as the length of the longest prefix of R, under which $V \cup \{\eta\}$ is consistent, that is,

$$DDP_V(\eta) = m \iff (r_1; r_2; ...; r_m)^{-1}(\eta) \subseteq V$$

For example, $DDP_V(\eta) = 0$ means that η has just arrived to the replica and no degree of its dependencies are checked yet, and $DDP_V(\eta) = k - 1$ means that

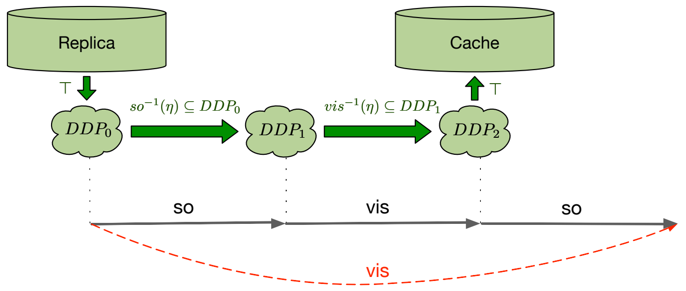

Replica
Cache
⊤
⊤
$so^{-1}(\eta) \subseteq DDP_0$
$vis^{-1}(\eta) \subseteq DDP_1$
DDP_0
DDP_1
DDP_2
so
vis
so
vis

Fig. 8: Representation of the stepwise process where effects become closer to the cache, before actually entering it

all of η's dependencies according to the truncated relation are present and it is safe now to add it to the cache. We use the notation DDP^i to refer to the set of all effects whose DDP value is equal to i.

This way, by periodically refreshing the DDP of effects, the porcess of moving effects from the replica to the cache is recorded while the the dependencies arrive, and as we will explain shortly, we can totally avoid computing closures of relations and redundant computations at replicas by a simple memoization technique. Finally, note that (following the discussion in the previous section) we require the dependencies to be looked for, in the replica and the cache respectively, for the LB and non LB contracts. i.e. in the case of LB contracts $DDP_{replica}$ should be computed and DDP_{cache} for UB and hybrid contracts.

6.2 Example

In this part, we will explain the behavioral outline of our algorithm using an example. The formal operational semantics of this approach can be found in appendix B.

Let's assume we are given a contract $\psi = \forall(a,b).a \xrightarrow{\text{so;vis;so}} b \Rightarrow a \xrightarrow{\text{vis}} b$, and we want replicas to maintain consistent caches according to this contract. We explain our approach by explaining a replicas' behavior when certain events occur:

- **Remote Effect Arrival:** When a new remote effect arrives to the replica, it is simply added to the set of local effects and its DDP is initially set to 0. Since the length of the given contract is 3, as we will see shortly the effect requires two steps of DDP refreshes, before it can enter the cache.
- **Operation Submission:** Since the given contract is an LB type, the replica must now make sure that all effects from ealier operations of the same session, are present in the *cache* and if not, block the operation temporarily. The operation can proceed and wtiness *all* effects at the replica, after the mentioned effects enter the cache.
- **Cache Refresh:** The replica, must periodically perform cache refreshes and move effects from DDP_2 to the cache. As explained we know that the complete set of dependencies for these effects are already present.

– **DDP Refresh:** At this periodic step, DDP of effects are updated by checking if they can be given a larger one. At each step the DDP of an effect η is increased from i to $i+1$ if all effects in $r_{i+1}^{-1}(\eta)$ already have DDP value at least equal to i. In this example, an effect η that has the initial DDP value 0, can get the value 1, only if all effects in $\mathsf{so}^{-1}(\eta)$ are already present at the replica which means they have DDP value of at least 0. Similarly, effects that have the DDP value of 1 can get the value 2, if all effects in their vis^{-1} set, have the DDP value of minimum 1. At this point, effects have reached the value 2, which means they can be now moved to the cache at the next cache refresh (Figure 8).

Note that, in case one of the elements of the given relation is a closure, for example assume the given relation is $\xrightarrow{so;vis^*;so}$, we can avoid all recursive computations, by allowing an effect η to go from DDP_1 to DDP_2, only if $vis^{-1}(\eta)$ is present in DDP_1 **and** in DDP_2. This way, we are sure that the same condition was also checked for all effects in $vis^{-1}(\eta)$ before they entered DDP_2, which brings us the desired behavior, without any computations involving closures.

Benchmark	LoC	Consistency Reqs.	Description
Counter	65	MR	Monotonicly increasing counter, e.g. YouTubes' watch count
DynamoDB	126	RMW	An integer register that allos different types of (conditional) puts and gets
Online Store	236	RMW	Online store with shopping carts and modifiable item prices
Bankaccount	85	2VIS ∧ RMW	A bankaccount application with deposit, withdraw and get balance operatio
Shopping List	140	MW ∧ RMW	A shopping list with concurrent adds and deletes functionality
Microblog	395	MW, RMW	A Twitter-like application modeled after Twissandra, with posting and reply to tweets, following, unfollowing and blocking users, etc.
Rubis	417	RMW, RMW∧2VIS	eBay-like application with browsing, bidding and making payments from a v

Fig. 9: Usage of weak consistency requirements in benchmark applications

7 Evaluation

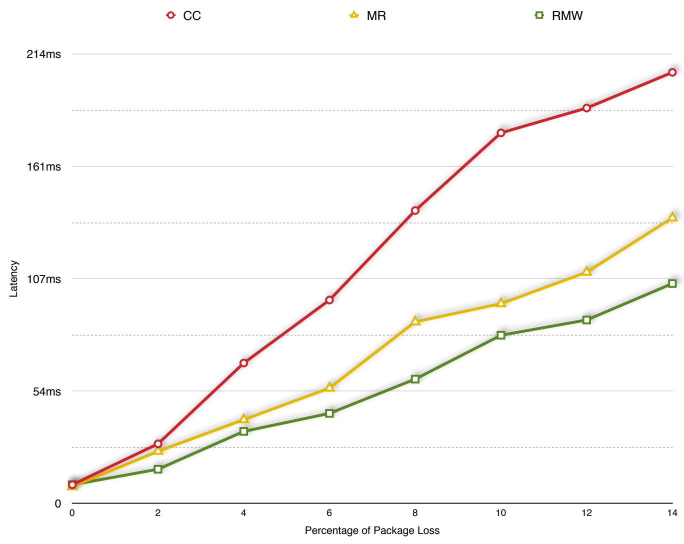
In this section, we present our evaluation study of SYNCOPE. The results are presented in three parts, where we first present the distribution of weak consistency requirements on benchmark programs. Second, we presents our studies on the performance of programs running on various consistency levels and finally, we present the complexity and perforamnce results from our study of implementing a well-understood ad-hoc prevention mechanism for lost-updates anomaly, compared to writing the same program in SYNCOPE.

7.1 Weak Consistency in Benchmark Programs

In this section, we present seven different benchmark applications we collected, in which various types of anomalous behavior under eventual consistency have been detected. We present these programs and their detected consistency requirements, in figure 9. For example, two following anomalies have been detected for operations of the microblog application:

1. When Alice unfollows Donald, but later sees more tweets from him. This is because the `getFolloweeList` operation did not witness the effect of the `unfollow` operation; a clear example of lost-updates anomaly which can be prevented by RMW guarantee.
2. When Donald posts a series of tweets, but after Alice refreshes her timline, only sees the fifth tweet. This can be prevented by requiring `getTweet` operation to return only tweets, whose prior tweets are also visible; which is exacly what is provided by enforcing MW contract.

In addition to having a large number of operations each of which might require a different level of consistency, above examples also show how in practice, some programs might inlcude operations that are involved in *multiple* types of anomalies. For example the `getBalance` operation of the bank account application above, shows two different types of anomalies, whose prevention requires 2VIS *and* RMW. The possiblity of showing *combinations* of anomalies, considering the large number of known anomalies, shows the inefficiency of any consistency enforcement technique specific to a certain type of anomaly.



(a) Example execution.

Fig. 10: A distributed application for comment section management

7.2 Latency and Staleness Comparison

7.3 Ad-hoc vs SYNCOPE

8 Related Works

9 Conclusion

References

1. Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.
2. KC Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. Declarative programming over eventually consistent data stores. *SIGPLAN Not.*, 50(6):413–424, June 2015.
3. Mohsen Lesani, Christian J. Bell, and Adam Chlipala. Chapar: Certified causally consistent distributed key-value stores. *SIGPLAN Not.*, 51(1):357–370, January 2016.
4. James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. Verdi: A framework for implementing and formally verifying distributed systems. *SIGPLAN Not.*, 50(6):357–368, June 2015.
5. Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Bolt-on causal consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 761–772, New York, NY, USA, 2013. ACM.
6. Daniel J. Abadi. Consistency tradeoffs in modern distributed database system design: Cap is only part of the story. IEEE Computer, 45(2), 2012.
7. M.P. Herlihy and J.M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.
8. Christos H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, 1979.
9. Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *Operating Systems Review*, 44(2):35–40, 2010.
10. Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike J. Spreitzer, Marvin M. Theimer, and Brent B. Welch. Session guarantees for weakly consistent replicated data. pages 140–149, 1994.
11. Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS'11, pages 386–400, Berlin, Heidelberg, 2011. Springer-Verlag.
12. Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, August 2008.
13. Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, October 2007.
14. P. Mahajan, L. Alvisi, and M. Dahlin. Consistency, availability, convergence. Technical Report TR-11-22, Computer Science Department, University of Texas at Austin, May 2011.

15. Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. Causal memory: definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, 1995.
16. Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. Providing high availability using lazy replication. *ACM Trans. Comput. Syst.*, 10(4):360–391, November 1992.
17. Kenneth Birman, André Schiper, and Pat Stephenson. Lightweight causal and atomic group multicast. *ACM Trans. Comput. Syst.*, 9(3):272–314, August 1991.
18. Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 401–416, New York, NY, USA, 2011. ACM.

A Proofs

Here, we present the detailed proofs of the theorems of the paper.

A.1 Proof of Theorem 1

We have the following two hypotheses and the goal:

$$\begin{array}{l} H_0 : (\mathsf{E}, op_{<s,i>}) \xrightarrow{V} (\mathsf{E}', \eta) \\ H_1 : V \texttt{ is } \psi\texttt{-consistent under } \mathsf{E} \\ G_0 : V \cup \{\eta\} \texttt{ is } \psi\texttt{-consistent under } \mathsf{E}' \end{array}$$

Based on the definition of consistency, we should prove the following:

$$G_1 : \forall (x \in V \cup \{\eta\}).\forall (a \in E'.A).R(a, x) \Rightarrow a \in V \cup \{\eta\}$$

By intro (and by the fact that $\mathsf{E}'.\mathsf{A} = \mathsf{E}.\mathsf{A} \cup \{\eta\}$) we will have the following new hypotheses and goal:

$$\begin{array}{l} H_2 : x \in V \cup \{\eta\} \\ H_3 : a \in E.A \cup \{\eta\} \\ H_4 : R(a, x) \\ G2 : a \in V \cup \{\eta\} \end{array}$$

By destructing H_2 we have two cases:
(i) $x \in V$: Since V is ψ-consistent, from H_4 we have $a \in V$ which proves the goal G_2
(ii) $x = \eta$: By replacing x with η in H_4 we will have the following:

$$H_5 : R(a, \eta)$$

Based on definitions, we will have the following:

$$H_6 : a \in R^{-1}_{E.A}(\eta)$$

We can now assume $R = r_1; r_2; ...; r_k$, and derive from H_6 that there exists c such that:

$$\begin{array}{l} H_7 : c \in r_k^{-1}(\eta) \\ H_8 : a \in (r_1; ...; r_{k-1})^{-1}_{E.A}(c) \\ H_9 : r_k^{-1}(\eta) \subseteq E.A \end{array}$$

Now by inversion on H_0 we will have the following:

$$H_{10} : R_V^{-1}(\eta) \subseteq V$$

which by definition will result:

$$H_{11} : r_k^{-1}(\eta) \subseteq V$$

Now from H_7, H_8 and H_{11} we can derive:

$$H_{12} : a \in R_V^{-1}(\eta)$$

and from H_{10} and H_{12} we have $a \in V$ which proves the final goal G_2.
QED.

A.2 Proof of Theorem 2

By contradiction:

$$\begin{aligned}
&H_0 : (\mathsf{E}, op_{<s,i>}) \xrightarrow{V} (\mathsf{E}', \eta)\\
&H_1 : (\mathsf{E}, op_{<s,i>}) \xrightarrow{V} (\mathsf{E}'', \eta)\\
&H_2 : a \in (E''.vis^{-1}(\eta))\\
&H_3 : a \notin (E'.vis^{-1}(\eta))\\
&G_0 : \bot
\end{aligned}$$

By inversion on H_1 we have:

$$\begin{aligned}
&H_4 : V'' \vdash (\mathsf{E}, op_{<s,i>}) \hookrightarrow (\mathsf{E}'', \eta)\\
&H_5 : V'' = \lfloor V \rfloor_V\\
\text{(by inversion on } H_4 \text{ and from } H_2\text{)}\ &H_6 : a \in V''
\end{aligned}$$

By inversion on H_0 and following a similar argument we can now derive:

$$\begin{aligned}
&H_7 : V'' \vdash (\mathsf{E}, op_{<s,i>}) \hookrightarrow (\mathsf{E}', \eta)\\
&H_8 : V' = \lfloor V \rfloor_V\\
&H_9 : a \notin V'
\end{aligned}$$

However from H_5 we know $V'' \subseteq V \wedge R_V^{-1}(V'') \subseteq V''$, which results in contradiction because of the maximality of V' defined in H_8.

A.3 Proof of Theorem 3

Before proving the theorem, let's first prove the following lemma:

Lemma 1. *Under an execution state E and for a given set $S \subseteq E.A$, if S is ψ-consistent under E, then $\forall(x \in S).R_S^{-1}(x) \subseteq S$ under E.*

Proof. From ψ-consistency of S we have the following:

$$H_0 : \forall(x \in S).\forall(y \in E.A).R.(y, x) \in S$$

Now based on the definition of R, we have the following:

$$H_1 : \forall(x \in S).R_{E.A}^{-1}(x) \subseteq S$$

However, since $S \subseteq E.A$ we have $R_S^{-1}(x) \subseteq R_{E.A}^{-1}(x)$ and the following will be derived:

$$H_2 : \forall(x \in S).R_S^{-1}(x) \subseteq S$$

which completes the proof of the lemma.

Now, in order to prove the theorem, we define $V = S$, so now we need to find E'' and η', such that the following hy potheses and goal hold.

$$\begin{aligned}
&H_0 : S \vdash (\mathsf{E}, op_{<s,i>}) \hookrightarrow (\mathsf{E}', \eta)\\
&H_1 : \mathsf{S} \cup \{\eta\}\ \texttt{is}\ \psi\texttt{-consistent under}\ E'\\
&G_0 : ((\mathsf{E}, op_{<s,i>}) \xrightarrow{S} (\mathsf{E}'', \eta'))
\end{aligned}$$

Case I: $vis \subseteq r_k$ We define S' to be the maximal closed subset of S, i.e. $S' = \lfloor S \rfloor_S$. Now we can $\eta' = F_{op}(S')$. Moreover, in order to define E'', we first define the following:

$$\begin{aligned} A'' &= E.A \cup \{\eta'\} \\ so'' &= E.so \cup \{(\eta'', \eta') | \eta'' \in E.A\} \\ vis'' &= E.vis \cup S' \times \{\eta'\} \end{aligned}$$

Finally, we define $E'' = (A'', vis'', so'')$, and now by applying [OPER] rule, we will have the following:

$$H_2 : S' \vdash ((\mathsf{A}, \mathsf{vis}, \mathsf{so}), op_{<s,i>})) \hookrightarrow ((\mathsf{A}'', \mathsf{vis}'', \mathsf{so}''), \eta')$$

Now, by applying [EXEC] rule on G_0, we have the following new goals:

$$\begin{aligned} &G_0 : S \subseteq E.A \\ &G_1 : R_S^{-1}(\eta) \subseteq S \\ &G_2 : vis \subseteq r_k \\ &G_3 : S' = S' = \lfloor S \rfloor_S \\ &G_4 : S' \vdash (\mathsf{E}, op_{<s,i>}) \hookrightarrow (\mathsf{E}', \eta) \end{aligned}$$

All the goals above are an already shown assumption but G_1, which is the direct result of applying lemma A.3 on H_1.

Case II: $vis \not\subseteq r_k$ We pick $E'' = E'$ and $\eta' = \eta$, so by applying the [EXEC] we have the following:

$$\begin{aligned} &G_0 : S \subseteq E.A \\ &G_1 : R_S^{-1}(\eta) \subseteq S \\ &G_2 : vis \not\subseteq r_k \\ &G_3 : S \vdash (\mathsf{E}, op_{<s,i>}) \hookrightarrow (\mathsf{E}', \eta) \end{aligned}$$

G_1 is the direct result of lemma A.3, and the rest of the goals are in fact in the assumptions, and thus the theorem is proved.

B Operational Semantics of the Augmented algorithm

Here, we explain our detailed operational semantics, to maintain multi-consistent replicated stores. We assume a given function from operation names, to consistency contracts: $\Psi : op \mapsto \psi$ and for simplicity reasons (again, it can be easily generalized) we consider contracts made by a single prop:

$$\Psi(op) = \forall(a,b).a \xrightarrow{R_{op}} \Rightarrow a \xrightarrow{vis} b.$$

For a given realtion R we also define $R[m]$ to refer to the m'th relation seed in R:

$$(r_1; r_2; ...; r_m; ...; r_k)[m] = r_m$$

Each replica in this semantics, maintains a pool of available effects, and a cache of filtered effects for each operation, each of which is a subset of pool that is closed under its associated contract, i.e. $\forall \eta \in \mathsf{cache}(op).(\mathsf{trunc}(R_{op}))^{-1}_{\mathsf{pool}}(\eta) \subseteq \mathsf{cache}(op)$ We also define DDP of effects which is maintained according to section 6. Following is the formal definitions and the operation semantics.

$$\delta \in \texttt{Replicated Data Type} \qquad v \in \texttt{Value} \qquad op \in \texttt{Operation Name}$$

$$s \in \texttt{Session Id} \qquad i \in \texttt{Effect Id} \qquad \rho \in \texttt{Replica Id}$$

$$\begin{array}{lll}
\eta & \in \texttt{Effect} & := (s, i, op, v) \\
\mathsf{pool} & \in \texttt{Pool} & := (v, \mathcal{P}(\eta)) \\
\mathsf{cache} & \in \texttt{Cache} & := op \mapsto (v, \mathcal{P}(\eta)) \\
\mathsf{DDP} & \in \texttt{Deps.Presence} & := op \mapsto (\eta \mapsto \{0, 1, ..., k-1\}) \\
F_{op} & \in \texttt{Op. Def.} & := v \rightarrow \eta \\
\mathsf{A} & \in \texttt{Eff Soup} & := \mathcal{P}(\eta) \\
\mathsf{vis}, \mathsf{so} & \in \texttt{Relations} & := \mathcal{P}((\eta, \eta)) \\
\mathsf{E} & \in \texttt{Exec State} & := (\mathsf{A}, \mathsf{vis}, \mathsf{so}) \\
\Theta & \in \texttt{Store} & := \rho \mapsto (\mathsf{pool}, \mathsf{cache}, \mathsf{DDP}) \\
\sigma & \in \texttt{Session} & := \cdot \;\mid\; op :: \sigma \\
\Sigma & \in \texttt{Session Soup} & := \langle s, i, \sigma \rangle \parallel \Sigma \;\mid\; \emptyset
\end{array}$$

$$\mathsf{ssn}(s, \_, \_, \_) = s \qquad \mathsf{id}(\_, j, \_, \_) = j \qquad \mathsf{oper}(\_, \_, op, \_) = op \qquad \mathsf{rval}(\_, \_, \_, n) = n$$

Auxiliary Reduction $\boxed{v \vdash (\mathsf{E}, \langle s, i, op \rangle) \hookrightarrow (\mathsf{E}', \eta)}$

[Oper]

$$\frac{\begin{array}{c} F_{op}(v) = \eta \qquad \mathsf{ssn}(\eta) = s \qquad \mathsf{id}(\eta) = i \qquad \mathsf{A}' = \mathsf{A} \cup \{\eta\} \\ \mathsf{vis}' = \mathsf{vis} \cup S \times \{\eta\} \qquad \mathsf{so}' = \mathsf{so} \cup \{(\eta', \eta) \,|\, \eta' \in \mathsf{A} \;\wedge\; \mathsf{ssn}(\eta') = s \;\wedge\; \mathsf{id}(\eta') < i\} \end{array}}{v \vdash ((\mathsf{A}, \mathsf{vis}, \mathsf{so}), \langle s, i, op \rangle \hookrightarrow ((\mathsf{A}', \mathsf{vis}', \mathsf{so}'), \eta)}$$

Operational Semantics $\boxed{(\mathsf{E}, \Theta, \Sigma) \xrightarrow{\eta} (\mathsf{E}', \Theta', \Sigma')}$

[Pool Refresh]

$$\frac{\begin{array}{c} \eta \in \mathsf{E.A} \quad \Theta(\rho) = (\mathsf{pool}, \mathsf{cache}, \mathsf{DDP}) \quad \eta \notin \mathsf{pool}_e \\ \mathsf{pool}' = (\mathit{apply}\ \eta\ \mathsf{pool}_v, \mathsf{pool}_e \cup \{\eta\}) \\ \Theta' = \Theta[\rho \mapsto (\mathsf{pool}', \mathsf{cache}, \mathsf{DDP})] \end{array}}{(\mathsf{E}, \Theta, \Sigma) \xrightarrow{\eta} (\mathsf{E}, \Theta', \Sigma)}$$

[DDP Refresh]

$$\frac{\begin{array}{c} \Theta(\rho) = (\mathsf{pool}, \mathsf{cache}, \mathsf{DDP}) \quad \eta \in \mathsf{pool}_e \quad \mathsf{oper}(\eta) = op \\ \mathsf{DDP}(op)(\eta) = i \quad i < k \quad \mathsf{DDP}'(op) = \mathsf{DDP}(op)[\eta \mapsto i+1] \\ \mathsf{DDP}(op)((R_{op}[i+1])^{-1}(\eta)) \subseteq \mathsf{DDP}^i \\ \Theta' = \Theta[\rho \mapsto (\mathsf{pool}, \mathsf{cache}, \mathsf{DDP}[op \mapsto \mathsf{DDP}'(op)])] \end{array}}{(\mathsf{E}, \Theta, \Sigma) \xrightarrow{\eta} (\mathsf{E}, \Theta', \Sigma)}$$

[Cache Refresh]

$$\frac{\begin{array}{c} \Theta(\rho) = (\mathsf{pool}, \mathsf{cache}, \mathsf{DDP}) \quad \eta \in \mathsf{pool}_e \quad \mathsf{oper}(\eta) = op \\ \eta \notin \mathsf{cache}(op)_e \quad \mathsf{cache}' = (\mathit{apply}\ \eta\ \mathsf{cache}(op)_v, \mathsf{cache}(op)_e \cup \{\eta\}) \\ \mathsf{DDP}(op)(\eta) = k - 1 \quad \Theta' = \Theta[\rho \mapsto (\mathsf{pool}, \mathsf{cache}', \mathsf{DDP})] \end{array}}{(\mathsf{E}, \Theta, \Sigma) \xrightarrow{\eta} (\mathsf{E}, \Theta', \Sigma)}$$

[LB Exec]

$$\frac{\begin{array}{c} \Theta(\rho) \vdash (\mathsf{E}, \langle s, i, op \rangle) \hookrightarrow (\mathsf{E}', \eta) \\ \Theta(\rho) = (\mathsf{pool}, \mathsf{cache}, \_) \quad \mathsf{so}^{-1}(\eta) \subseteq \mathsf{cache}(op)_e \end{array}}{(\mathsf{E}, \Theta, \langle s, i, op :: \sigma \rangle \parallel \Sigma) \xrightarrow{\eta} (\mathsf{E}', \Theta, \langle s, i+1, \sigma \rangle \parallel \Sigma)}$$

[UB Exec]

$$\frac{\begin{array}{c} \Theta(\rho) = (\mathsf{pool}, \mathsf{cache}, \_) \\ \mathsf{cache}(\mathsf{op}) \vdash (\mathsf{E}, \langle s, i, op \rangle) \hookrightarrow (\mathsf{E}', \eta) \end{array}}{(\mathsf{E}, \Theta, \langle s, i, op :: \sigma \rangle \parallel \Sigma) \xrightarrow{\eta} (\mathsf{E}', \Theta, \langle s, i+1, \sigma \rangle \parallel \Sigma)}$$

Fig. 11: Operational semantics of a replicated data store.