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 distributed applications typically requires data to be replicated over geo-distributed stores. But, such architectures force application developers to make an undesirable tradeoff between ease of reasoning, possible when replicated data is required to be strongly consistent, and performance, possible when such guarantees are weakened. Unfortunately, undesirable behaviors may arise under weak consistency that can violate application correctness, forcing designers to either implement ad-hoc mechanisms to avoid these anomalies, or choose to run applications using stronger levels of consistency than necessary. The former approach introduces unwanted complexity, while the latter sacrifices performance. In this paper, we describe a lightweight runtime verification system that relieves developers from having to make such tradeoffs. Instead, our approach leverages declarative axiomatic specifications that reflect the necessary constraints any correct implementation must satisfy to guide a runtime consistency enforcement and monitoring mechanism. This mechanism guarantees a *provably optimal* strategy that imposes no additional communication or blocking overhead beyond what is required to satisfy the specification, allowing distributed operations to run in a *provably safe* environment. Experimental results show that the performance of our automatically derived mechanisms is better than both specialized hand-written protocols and common store-offered consistency guarantees, providing strong evidence of its practical utility.

Keywords: Runtime Safety Enforcement and Montoring, Weak Consistency, Distributed Systems, Haskell

1 Introduction

Historically, the *de facto* system abstraction for developing distributed programs has typically included ACID[1] properties. These properties guarantee replication transparency (i.e. requiring distributed systems to *appear* as a single compute and storage server to users), and make it straightforward to develop standardized implementation and reasoning techniques around *strongly consistent* (SC) distributed stores. Although such strong notions of consistency simplify reasoning, they also introduce extensive synchronization overhead which is often

[1] Atomicity, Consistency, Isolation and Durability

unacceptable for web-scale applications that need to be "always-on" even in the presence of 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 local state of each application server only represents an *unspecified order* of an *unspecified subset* of the aggregate collection of all updates submitted to the system globally. Applications that may not tolerate the level of inconsistency imposed by EC are often equipped with *ad hoc* mechanisms to enforce the required level of consistency. Unfortunately, such enforcement mechanisms are closely tied to the application logic, confounding standardization, while complicating application reasoning, maintainability, and reusability.

In this paper, we propose an alternative approach to weak consistency enforcement that circumvents the aforementioned issues. SYNCOPE is a lightweight runtime verification system for Haskell that allows application developers to take advantage of weak consistency without having to re-engineer their code to accommodate anomaly preemption mechanisms. 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 consistency enforcement logic, requiring reasoning about both *operationally*. By separating application semantics from consistency enforcement semantics, however, admitting operational reasoning for the former, and declarative reasoning for the latter, programmers are liberated from having to worry about implementation details of anomaly preemption mechanisms, allowing them to be focused instead on application semantics, under the assumption that specified consistency requirements are automatically enforced by the data store at runtime. Our approach admits declarative reasoning for consistency enforcement via a specification language that allows programmers to formally specify consistency requirements. The design of our specification language is based on the observation that all anomalous behaviors allowed under EC occur as the result of nodes executing operations before certain *dependencies* are satisfied. Using SYNCOPE, one can specify arbitary dependency relations between updates, and a runtime monitoring and verification system working on top of each EC data store replica guarantees that an operation will only proceed if it can witness all of its dependencies. For example, *lost-updates* is a well-known anomaly possible under EC that occurs when an operation o from a client session is routed to a replica different than the replica that served earlier operations from the same session, because of transient system properties, such as load balancing or network partitions. When this happens, o may successfully execute without witnessing updates from those earlier operations. In this case, because o is dependent on updates from *all previous operations from the same sessions*, SYNCOPE guarantees to temporarily block operations until these dependencies become available at o's replica. Of course, if this anamolous behavior can be tolerated by the application, this level of dependency tracking will be elided.

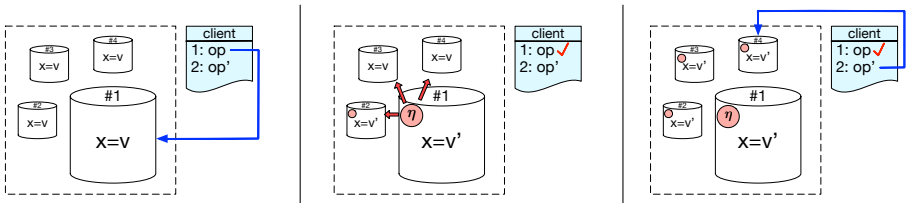
We make the following contributions: (i) We propose an expressive specification language to express fine-grained consistency requirements of applications in terms of the dependencies between operations. (ii) We describe a generic runtime consistency enforcement mechanism that analyzes each operation's consistency specification, and ensures that its dependencies are available at the replicas on which it executes. We formalize the system's operational semantics, and prove its correctness and optimality. (iii) We describe an implementation of these ideas in a tool called SYNCOPE, which works on top of an off-the-shelf EC data store. Evaluation over realistic applications and microbenchmarks, 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. A system model that describes the key notions of consistency and replication is presented in Sec. 2. In Sec. 3, we provide a detailed example to further motivate the problem. In Sec. 4 and Sec. 5, we formally present our specification language and the high level operational semantics of the runtime system, with correctness and optimality theorems. Sec. 6 elaborates on the algorithmic aspects of our runtime that is key to its efficient realization. Sec. 7 describes implementation of SYNCOPE, and evaluates its applicability and practical utility. Related work and conclusion are presented in Sec. 8 and Sec. **??**.

2 System Model

A data store in our system model is a collection of *replicas* (#1,#2,...), each of which maintains a copy of a set of replicated *data object* (x,y,...). Each data object includes and maintains a *state value* (v,v',...) and is equipped with a set of *operations* (op,op',...). Operations may read the state of an object residing in a replica, and modify it by generating *update effects* (η,η',...). These effects are then asynchronously sent to all other replicas where they are applied to the state of the object instance at the recipient replica using a user-supplied function. Figs. 1a and 1b illustrate this process.

Because there is no direct synchronization between replicas when an operation is executed, concurrent and possibly conflicting updates can be generated



(a) A client submits an operation op to the store, which is routed to the replica #1.

(b) The state of the replica #1 is updated, an effect is created and is being propagated

(c) Second operation op′ is submitted to the store, which is routed to the replica #4

Fig. 1: system model of SYNCOPE

at different replicas. Conflict resolution is handled at the point when an effect is applied to the current state of the object, and must be designed to ensure that all replicas eventually converge to the same value, assuming generated effects quiesce. This model admits all inconsistencies and anomalies associated with eventual consistency [12, 13].

Clients in our model interact with the store by invoking operations on objects. A *session* is a sequence of operations invoked by a particular client. Consequently, operations (and effects) can be uniquely identified by the *session id* that invoked them, and their *sequence number* in that particular session, which is used by replicas to record the set of all updates that are locally applied. Since, the data store may be concurrently accessed by a large number of clients, operations (even from the same session) might be routed to different replicas to improve latency (see Figs. 1a and 1c).

Lastly, we define two relations over effects created in the store. *Session order* (so) is an irreflexive, transitive relation that relates an effect to all subsequent effects from the same session. *Visibility* (vis) is an irreflexive and assymetric relation that relates an effect to all others that are influenced by it (i.e., witnesses its update) at the time of their generation. For example, in Fig. 1c $\texttt{vis}(\eta, \eta')$ holds, since η (the effect of `op`) has already been delivered and applied to the replica #4, when $\texttt{op}'$ is executed and thus has influenced generation of η'.

3 Motivation

3.1 Replicated Data Types in ECDS

To motivate our approach, consider a highly available (low latency) application for managing comments on posts in a photo sharing web site. Fig. 2a presents a simple Haskell implementation of such an application cognizant of our system model.

In this implementation, `Effect` and `State` strings are respectively defined as the text of a single comment, and 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, returns the updated state of the replica, which is the concatenation of the old state and the given effect. For perspicuity, we omit any conflict resolution strategy in the code; however, developers (using timestamps, roll-backs, etc.) can design the `apply` function to resolve conflicting concurrent updates as they desire, consistent with application invariants.

An example of how users interact with this application is presented in Fig.2b, where Alice and Bob invoke operations on an object (here, a photo of Alice in Chicago); the chronological order of events is given in black circles. At time ❶, Bob writes a comment, which is routed to replica #1, whose effect is then propagated and delivered to replica #2 at ❷; while Alice's first read operation is routed to ❸. Alice and Bob then keep talking, generating more read and write events, while updates are propagated concurrently between the two replica.

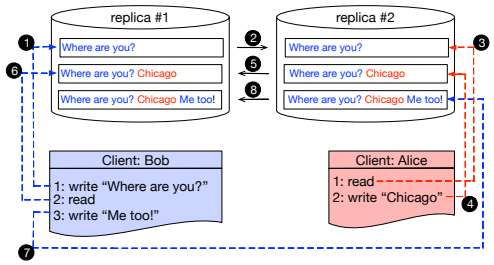
```
type Effect = String
type State =  String

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

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

apply :: State -> Effect -> State
apply s comment = s ++ comment
```

(a) A simple implementation



(b) Example execution

Fig. 2: A distributed application for comment section management

As mentioned before, lost updates, is a well-known, often undesirable, behavior admitted by eventually consistent data stores (ECDS). An example of such anomaly can occur here if at time ❻, Bob is temporarily disconnected from both replicas in the figure, and his read operation is routed to another replica #3, that has not yet received any updates from #1 or #2. Consequently, Bob cannot see his first comment and would retry submitting it, assuming it failed the first time it was sent. This would result in multiple copies of the message eventually being displayed on each replica.

3.2 Ad-hoc Anomaly Prevention

One way to prevent the above anomaly is to tag each effect using a unique identifier as mentioned in Sec. 2. Using these tags, replicas will be able to track all locally available effects, and temporarily *block* operations, until all the preceding effects from the same session arrive at the replica. For example, the replica #3 that receives Bob's read in the above undesired scenario, can simply postpone its execution until all prior effects upon which this operation depends reach this replica..

In order to reduce the overhead of tracking dependencies per operation, the above idea can be refined using another technique called *filtration*, which is based on separating the locally available effects at each replica that have not yet been applied to the state from those who have. In the above example each replica can maintain a *safe environment* for operations (e.g. using a soft-state cache), that contains an effect only if it also contains all the previous effects from the same session. This way, an operation can proceed, when the effect of its previous operation from the same session is already applied to the state (which transitively yields the presence of all dependencies).

We present a modified version of the running example in Appendix A, updated to tolerate the lost-update anomaly by implementing the blocking mechanism in the `read` function and incorporating filtration in the `apply` function as explained above. Unfortunately, these modifications require fundamental and pervasive changes to the original code. Additionally, the changes are heavily

tangled with application logic, complicating reasoning, hampering correctness arguments, and sacrificing composability and scalability.

3.3 Our Solution

SYNCOPE is a runtime enforcement mechanism that allows developers to define a consistency level for each operation *a priori*, delegating the responsibility for ensuring that constraints defined by this level are respected. Our technique generalizes blocking and filtration mechanisms, admits arbitrary user-defined dependency relations for each operation, and maintains a consistent *shim layer* on top of each ECDS replica.

The SYNCOPE shim layer maintains multiple safe environments $(E_1, E_2, ...)$ by periodically (or on-demand) reading from the underlying ECDS database, and adding effects to each environment, only if its dependencies have already been added (Fig.3). SYNCOPE realizes this idea efficiently, using a simple tagging mechanism that represents effects in an environment via a tag associated with that environment. Each operation only witnesses effects from its associated environment, and is blocked by the runtime system if the necessary dependencies are not present.

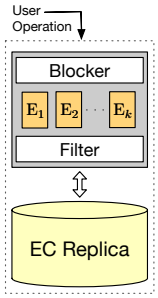


Fig. 3: SYNCOPE

Users can specify arbitrary consistency guarantees in a language that is seeded with `so` and `vis` relations. Constraints on read operations can be used to synthesize appropriate filtration and blocking mechanisms. For example, the following *contract*, eliminates the possibilty of lost-update anomalies by establishing the appropriate conditions under which an effect may be witnessed by the current operation:

$$\psi : \forall a.\ \xrightarrow{\textsf{so}} \hat{\eta} \Rightarrow a \xrightarrow{\textsf{vis}} \hat{\eta}$$

4 Specification Language

The formal syntax of our specification (or contract) language, presented in Fig.4a, allows definitiosn of `prop`, a first-order formula that establishes dependency relations between effects, necessary to determine the effects an operation may witness, under a given consistency level. The language is seeded with so and vis, respectively representing session order and visibility relations over effects, and defines dependency `relation` as a sequence[2] of seeds, where $(a \xrightarrow{\mathtt{r}_1;\ldots;\mathtt{r}_k} b)$ is interpreted as $\exists c.(a \xrightarrow{\mathtt{r}_1;\ldots;\mathtt{r}_{k-1}} c \wedge c \xrightarrow{\mathtt{r}_k} b)$. `null` is the empty relation. Additionally, the language allows conjunctions of propositions, `spec`, used to define a safe environment free from *multiple* inconsistencies. Our language is crafted to capture all fine-grained weak consistency levels, including well-known ones such as those explicated by Terry et al. [13] (see e.g., Fig.4b).

We provide two important classes of contracts, and explain how they can be satisfied with different enforcement techniques.

[2] SYNCOPE also allows using closures of seeds, which is omitted here for simplicity.

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

(a) syntax of contracts

Guarantee	Contract
Read My Writes	$\forall a.a \xrightarrow{\mathsf{so}} \hat{\eta} \Rightarrow a \xrightarrow{\mathsf{vis}} \hat{\eta}$
Monotonic Writes	$\forall a.a \xrightarrow{\mathsf{so};\mathsf{vis}} \hat{\eta} \Rightarrow a \xrightarrow{\mathsf{vis}} \hat{\eta}$
Monotonic Reads	$\forall a.a \xrightarrow{\mathsf{vis};\mathsf{so}} \hat{\eta} \Rightarrow a \xrightarrow{\mathsf{vis}} \hat{\eta}$
Transitive Visibility	$\forall a.a \xrightarrow{\mathsf{vis};\mathsf{vis}} \hat{\eta} \Rightarrow a \xrightarrow{\mathsf{vis}} \hat{\eta}$

(b) examples

Fig. 4: SYNCOPE Specification Language

LB: A *lower bound* (LB) contract is one in which all defined dependency relations end with an so, i.e. are of the following form: $(\forall a.a \xrightarrow{r_1;r_2;\ldots;\mathsf{so}} \hat{\eta} \Rightarrow a \xrightarrow{\mathsf{vis}} \hat{\eta})$. It specifies the smallest set of effects that any operation should witness to maintain consistency, e.g. RMW and MR in Fig.4b.

UB: Similarly, we define *upper bound* (UB) contracts as those whose dependency relations end with a vis. These contracts define constraints on the set of effects made visible to each operation; if an effect is in the set, certain dependencies of that effect must also be included, e.g. 2VIS and MW in Fig.4b.

Our consistency enforcement approach is based on blocking operations with LB contracts to make sure that they witness *all effects that they are supposed to*, and filtering for UB contracts to make sure that they do not witness *effects that they are not supposed to* (e.g. see Fig. 5).

1. Consider a replica containing $\{\eta_1, \eta_2, \eta_3\}$ when an operation *op* arrives with a UB contract according to which, any operation witnessing η_1, must also witness η_4.
2. A violation of the contract occurs if *op* witnesses η_1 (left), whereas a filteration mechanism enforces the contract (right).

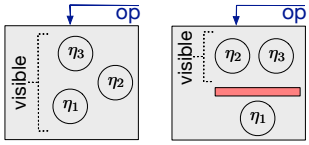


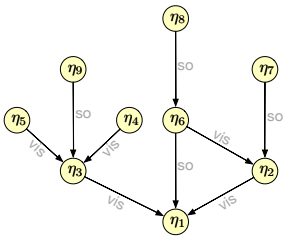
Fig. 5: Filteration and UB contracts

5 Semantics

In this section, we present the consistency enforcement mechanism of SYNCOPE, abstracted as a formal operational semantics. Our approach is complete for the specification language defined in Sec.4. However for better comprehensibility, we present the semantics and the theorems paramterized over a contract consisting of a single proposition. Therefore, in the rest of this section, we will assume a given contracat ψ of the following form:

$$\psi = \forall a.a \xrightarrow{\mathtt{r}_1;\mathtt{r}_2;\ldots;\mathtt{r}_k} \hat{\eta} \Rightarrow a \xrightarrow{\mathsf{vis}} \hat{\eta} \qquad \mathtt{r}_i \in \{\mathsf{vis}; \mathsf{so}\}$$

The operational semantics defines 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 stands for the set of all effects produced in the system, and *primitive relations* $\mathsf{vis}, \mathsf{so} \subseteq \mathsf{A} \times \mathsf{A}$, respectively represent the visibility and session order among such effects. Figs. 6a



(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) Primitive relations

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

(c) Inverse of relations

Fig. 6: A simple execution state

and 6b present a simple execution state consisting of 9 effects with associated primitive relations[3]. We denote the subset of A consisting of effects that satisfy a certain condition as $\mathsf{A}_{(\texttt{condition})}$.

Note that SYNCOPE's contracts 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 expressible in the specification language. If an execution state E is a valid model for a contract ψ, we say that E satisfies ψ ($\mathsf{E} \models \psi$).

The reduction relation in our semantics is of the form $(\mathsf{E}, \mathsf{op}_{<s,i>}) \xrightarrow{V} (\mathsf{E}', \eta)$, which can be interpreted as a transformation of the initial execution state E, caused by a replica with a local set of effects V, when it executes op, the i^{th} operation from the session s. During this reduction step, a new effect η is produced and added to the system, resulting in a new execution state E' composed of an updated effect soup and new primitive relations.

5.1 Preliminaries

Before presenting the operational semantics, we first introduce supporting definitions and notations. We start by defining the interpretation of an *inversed* dependency relation R^{-1} under an execution state E, which is utilized in the basis of our consistency enforcement mechanism. We previously mentioned our interpretation for so and vis between effects under E; this can now be straightforwardly extended to their inverse as follows[4]:

$$\mathtt{r}^{-1}(S) = \bigcup_{b \in S} \{a | (a, b) \in \mathsf{E}.\mathtt{r}\} \qquad \mathtt{r} \in \{\mathsf{so}, \mathsf{vis}\} \tag{1}$$

Additionally, based on our interpretation of the sequences of seed relations given in Sec.4, we can extend the above definition:

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

[3] We omit drawing transitive so edges (e.g. between η_8 and η_1) for better readability.

[4] Note that when the input of an inversed relation is a singleton $\{\eta\}$, we drop the brackets and simply write it as $\mathtt{r}^{-1}(\eta)$

It might seem that we are ready to define any R^{-1} based on the two definitions above; however, note that definition (2) fails to capture the reality of our system model, where all computations are performed by replicas independently; at any given moment, a replica might have access to only a *subset of all produced effects* in the system. For example, consider $(\mathsf{so};\mathsf{vis})^{-1}(\eta_1)$ under the execution state presented in Fig.6. In order to compute this set, based on (2) we have:

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

Now, since there exists *mid-level* effects $c = \eta_2$ and $c = \eta_3$, that satisfy the above definition respectively for η_7 and η_9, we can conclude: $(\mathsf{so};\mathsf{vis})^{-1}(\eta_1) = \{\eta_7, \eta_9\}$. Consider a replica that contains $\{\eta_1, \eta_6, \eta_7, \eta_9\}$ at the moment, and wants to check if the dependencies of η_1 are locally present or not. Even though based on the above definition, the answer is affirmative (since the replica does contain $\{\eta_7, \eta_9\}$), the replica has no way to verify it, since the mid-level effects η_2 and η_3 are not present at the replica yet.

To capture the above property, we redefine the inverse of R, *according to a set of available effects* V, that considers whether all required mid-level effects are present in V. The following definition is based on (1) and a more strict version of (2):

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

For example, in Fig.6, $(\eta_9 \in (\mathsf{so};\mathsf{vis})^{-1}_{\{\eta_1,\eta_3\}}(\eta_1))$ holds, but $(\eta_9 \notin (\mathsf{so};\mathsf{vis})^{-1}_{\{\eta_1\}}(\eta_1))$.

We define a set V to be *self-contained* for a given effect η, written as $\mathbb{SC}^R_\eta(V)$, if V contains all the required mid-level effects to compute R inverse of η in totality, i.e.

$$\mathbb{SC}^R_\eta(V) \iff R_V^{-1}(\eta) = R^{-1}_{E.A}(\eta) \tag{4}$$

For example in Fig.6, $\mathbb{SC}^R_{\eta_1}(V)$ holds for an arbitrary R and for any V that is a superset of $\{\eta_1, \eta_2, \eta_3, \eta_4, \eta_5\}$.

We define $\mathsf{trunc}()$ as a function that given $R \in \texttt{relation}$, returns a new relation by removing the last element from the sequence in R:

$$\mathsf{trunc}(R) = \begin{cases} \texttt{null} & \texttt{if} \quad R = \mathtt{r} \quad \texttt{or} \quad R = \texttt{null} \\ R' & \texttt{if} \quad R = R';\mathtt{r} \end{cases} \tag{5}$$

Finally, we define *closed subsets* of a given set V, as the subsets that are closed under $(\mathsf{trunc}(R)_V^{-1})$, that also contain all the required mid-level effects to compute $\mathsf{trunc}(R)^{-1}$. Moreover, we define the largest element among such subsets, as the *maximally closed subset* of V as follows[5]:

$$\begin{aligned} \texttt{closed subsets} : V' \in \lfloor V \rfloor \quad &\iff V' \subseteq V \quad \wedge (\mathsf{trunc}(R))_V^{-1}(V') \subseteq V' \ \wedge \ \mathbb{SC}_\eta^{\mathsf{trunc}(R)}(V') \\ \texttt{maximally closed subset} : V' = \lfloor V \rfloor_{\texttt{max}} \quad &\iff V' \in \lfloor V \rfloor \wedge \nexists V'' \in \lfloor V \rfloor . |V''| > |V'| \end{aligned}$$

Auxiliary Definitions

$op \in \texttt{Oper. Name}$
$v \in \texttt{Ret. Val.}$
$s \in \texttt{Sess. ID}$
$\eta \in \texttt{Effect} := (s, op, v)$

$F_{op} \in \texttt{Op. Def.} := \mathcal{P}(\eta) \mapsto v$
$\mathsf{A} \in \texttt{Eff Soup} := \mathcal{P}(\eta)$
$\mathsf{vis}, \mathsf{so} \in \texttt{Relations} := \mathcal{P}((\eta, \eta))$
$\mathsf{E} \in \texttt{Exec State} := (\mathsf{A}, \mathsf{vis}, \mathsf{so})$

Auxiliary Reduction

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

[Oper]

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

Operational Semantics

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

[UB Exec]

$$\frac{\mathtt{r}_k = \mathsf{vis} \quad V \subseteq E.A \quad V' = \lfloor V \rfloor_{\mathtt{max}} \quad V' \vdash (E, op_{<s,i>})) \hookrightarrow (E', \eta)}{(\mathsf{E}, op_{<s,i>}) \xrightarrow{V} (\mathsf{E}', \eta)}$$

[LB Exec]

$$\frac{\mathtt{r}_k = \mathsf{so} \quad V \subseteq E.A \quad \mathbb{SC}^R_\eta(V) \quad R_V^{-1}(\eta) \subseteq V \quad V \vdash (E, op_{<s,i>})) \hookrightarrow (E', \eta)}{(\mathsf{E}, op_{<s,i>}) \xrightarrow{V} (\mathsf{E}', \eta)}$$

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

5.2 Core Operational Semantics

Our operational semantics is defined as a set of reduction rules representing our consistency enforcement approach (see Fig.7). The small-step reduction relation ($\rightarrow$) is parametrized over a set V, which stands for the locally available set of effects at the replica taking the reduction step. Trivially, V must be a subset of all effects in the system at the initial execution state, however, there is no other restrictions on V, since we only assume eventual consistency in the underlying store.

The rule [oper] defines the abstract procedure of generating a new effect η, by witnessing a set of effects S, using a user-defined function F_{op}. We formally define an effect as a tuple $\eta = (s, op, v)$, representing the session s, operation name op whose execution created η, and the value v that the replica returns, responding to that operation. Moreover, the rule explains how the execution state changes after a new effect is produced. Specifically, in the new execution state, the effect soup A' contains the newly created effect η, the relation vis' captures the fact that all effects in the set S were made visible to η, and so' states that all effects from the same session as the current operation that are already presenet in the system, should be in session order with η in the final execution state.

Rule[ub exec], defines the execution of an operation in a replica under a UB contract. The rule requires operations to only witness V', the maximally closed

[5] We slightly abuse the previously defined notation in (3) and use a *set* of effects as the input of R^{-1}, which is defined as: $x \in R_V^{-1}(S) \iff \exists (y \in S).x \in R_V^{-1}(y)$.

subset of V. Thus,, the rule governs how replicas create safe environments for operations, by filtering out undesirable or uwanted effects.

Rule [LB EXEC] defines the step taken by a replica when an operation is executed 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, assuming that V contains all the mid-level effects to determine dependencies of the newly created effect η (i.e. is a self contained set). Thus, the rule governs replicas to block execution of an operation under an LB contract, if the replica is unable to verify the presence of all necessary dependent effects.

5.3 Soundness and Optimality

In this section we present our meta-theoretic results on the desired properties for our consistency enforcement mechanism. Three theorems are presented, regarding the correctness of our approach, maximality of witnessed effects by each operation (i.e. minimum staleness) and the liveness guarantee of the system assuming the eventual delivery of all effects at all replicas. Detailed proofs of all theorems can be found in appendix B.

Before presenting the theorems, we define a ψ-consistent set of effects S under an execution state E as a set that is closed under $(R = \mathtt{r}_1; ...; \mathtt{r}_k)$, i.e.

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

Theorem 1. *For all reduction steps* $(\mathsf{E}, op_{<s,i>}) \xrightarrow{V} (\mathsf{E}', \eta)$*, the followins hold:*

(i) $\texttt{If } V \texttt{ is } \psi\texttt{−consistent under } \mathsf{E}, \texttt{ then } V \cup \{\eta\} \texttt{ is } \psi\texttt{−consistent under } E'$
(ii) $E' \models \psi[\eta/\hat{\eta}]$

The above theorem states the preservation of ψ-consistency at replicas, under reduction stpes. Moreover, it states the correctness of the enforced consistency guarantee at the final execution state.

Theorem 2. *For all reduction steps* $(\mathsf{E}, op_{<s,i>}) \xrightarrow{V} (\mathsf{E}', \eta)$*, the set of effects made visible to* η *is maximal. i.e. for all* $a \in V$*, if* $\mathbb{SC}_a^{\mathsf{trunc}(R)}(V)$*, then*

$$(a,\eta) \notin \mathsf{E}'.\mathsf{vis} \Rightarrow (\mathsf{E}'.A, \mathsf{E}'.\mathsf{vis} \cup \{(a,\eta)\}, \mathsf{E}'.so) \not\models \psi[\eta/\hat{\eta}]$$

Theorem 3. *For all execution states* E*, if there exists a set of effects* $S \subseteq \mathsf{E.A}$*, such that:*

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

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

A trivial corollary of the above theorem is the liveness of our operational semantics, since at least one set S with the requested properties always exists[6] at any execution state.

[6] $S = \mathsf{E.A}$. This requires the preservation of ψ-consistency under the resuction step, that is already shown in theorem 1.

6 Implementation

We implemented our tool as an extension to a GHC Haskell add-on, called Quelea [12], previously developed by Sivaramakrishnan and two of the authors of this paper. Quelea maintains a causally consistent cache on top of Cassandra, and *all* operations whose contract is satisfied under causal consistency, are performed witnessing that cache (even if they require considerably weaker guarantees than causal).

In SYNCOPE, we maintain a generic cache equipped with a tagging mechanism, where each operation is associated with a tag, and is allowed to witnesses only the subset of effects in the cache, holding that tag (i.e. effects that are in the *logical cache* associated with that operation). We implemented a dependency finder mechanism in SYNCOPE, that is used to verify the presence of arbitrarily defined dependencies of an effect in each logical cache. Consequently, SYNCOPE's filtration and blocking mechanisms are added to the runtime system, which rely on this dependency finder to keep each logical cache consistent according to its associated contract. Specifically, considering a dependency relation R and a replica containing a localset V of effects, an effect η is allowed to enter a logical cache, only if $\mathsf{trunc}(R)^{-1}_V(\eta)$ is already in that cache.

Considering the arbitrary length of the dependency relations expressible in our contract language, and the fact that verifying the presence of dependancies for an effect might fail for an unbounded number of trials until all the dependencies arrive, we noticed considerable redundencies in our dependency finder mechanism, which adversely affected the performance. To overcome this problem, we implemented a simple memoization technique in SYNCOPE that extends the binary notion of dependency presence to the *degree of dependency presence* (DDP) representing the maximum *depth* (or size) of the dependencies of an effect, whose presence have been verified so far. Consequently, when verification of the presence of dependencies for an effect fails, the runtime system can avoid redundant computations the time it tries to verify the same property for the same effect. SYNCOPE's runtime, by performing periodic DDP refreshes, tries to assign larger DDP values to each effect when more dependencies arrive at the replica. We leave the details of this technique, captured as an operational semantics in appendix C for an interested reader.

7 Evaluation

In this section, we present an evaluation study of our implementation, including a report on benchmark applications that utilize fine-grained weak consistency requirements, expressable in SYNCOPE's specification language. Fig. 8 presents seven such programs, that include library definitions of individual replicated data types as well as larger applications consisting of multiple replicated types.

Each program supports various operations, some of which have non-trivial consistency requirements. Out of the 38 non-SC operations defined in these programs, there are 11 such operations, whose consistency requirements can be

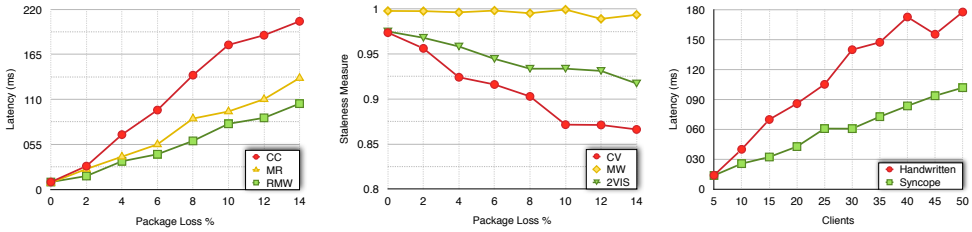
expressed as a combination of four previously described consistency guarantees: Monotonic Reads (MR), Monotonic Writes (MW), Read-My-Writes (RMW), and Transitive Vis (2VIS). The significant diversity among the consistency requirements of these operations emphasizes the need for a multi-abled environment that can understand and enforce fine-grained consistency requirements efficiently. It is clearly not practical to hard code them all, due their sheer number; even if we ignore bespoke consistency requirements, there are 15 combinations of just the 4 aforementioned consistency guarantees, even as there are other known guarantees, such as writes-follow-reads (WFR), which we ignored previously. Causal Consistency (CC), the strongest of the weak consistency guarantees, is often used as a metaphorical one-size to fit all weak consistency requirements. Notably, none of the operations we analyzed intrinsically require CC. Furthermore, we found that enforcing CC is significantly more expensive than enforcing weaker guarantees (see below), thus making CC a bad substitute for weaker guarantees. On the other hand, we found SYNCOPE's generic consistency enforcement mechanism to be very useful in this case.

While the consistency requirements of the applications in Fig. 8 are expressible as the combination of known guarantees as described above, this is not necessary in general. The bespoke consistency specifications of an application are usually crafted by analyzing the anomalies that need to be preempted. For example, consider a bank account application, which offers `deposit`, `withdraw` and `get_balance` operations, where `withdraw` is a strongly consistent operation that succeeds only if there are sufficient funds in the account. There are two anomalous scenarios associated with `get_balance` in this program: (i) when a user performs a `deposit` that is not reflected in subsequent `get_balance` operations; (ii) when a `get_balance` witnesses a `withdraw` effect without witnessing all `deposit` effects visible to it, resulting in `get_balance` returning a potentially (incorrect) negative balance. The consistency specifications of the bank account application are crafted to preempt these anomalies. As presented in Fig. 8, `get_balance` requires both RMW and 2VIS guarantees to be simultaneously satisfied.

We have deployed SYNCOPE on a cloud cluster, consisting of three fully replicated Cassandra replicas, running on separate machines within the same datacenter. Each machine is instantiated with a SYNCOPE shim layer, that responds to clients, which are instantiated on a VM co-located with one of the replicas on a machine. We deploy the cluster on three `m4.4xlarge` Amazon EC2 instances

Benchmark	Consistency	Description
Counter	MR	Monotonicly increasing counter, e.g. YouTubes' watch count
DynamoDB	RMW	Integer register allowing various conditional puts and gets
Online Store	RMW	Online store with shopping carts and modifiable item prices
Bankaccount	2VIS ∧ RMW	Offering deposit, withdraw and get balance operations
Shopping List	MW ∧ RMW	A shopping list with concurrent adds and deletes functionality
Microblog	MW, RMW	A Twitter-like application modeled after Twissandra
Rubis	RMW, RMW∧2VIS	eBay-like application with browsing, supporting user wallet

Fig. 8: Fine-grained consistency requirement in benchmark programs



(a) Latency (b) Staleness (c) Manual RMW

Fig. 9: A distributed application for comment section management

in the US-West (Oregon) region, with an inter-machine communication time of 5ms.

Inter-replica communication in Cassandra uses TCP connections, causing all messages to get delivered with no loss and reordering, which is in practice, far more consistent than EC, and masks out the performance gain from our fine-grained consistency guarantees. Consequently, to simulate a realistic EC environment, we injected artificial message losses in SYNCOPE's shim layer, forcing random messages to be delayed for 1s, simulating messages losses in a network with 600ms RTT.

Fig. 9(a) and 9(b) represent our experimental results, with a workload generated by 50 concurrent clients repeatedly running sessions, each composed of three operations, where operations uniformly choose from 5 objects, performed under a specified consistency level. We increase the percentage of delayed messages from 0 to 14. Each experiment ran for 100 repeated sessions per client. In addition to client perceived latency, we also measure the staleness of operations, which we define as the average ratio of the number of visible effects, to the number of all available effects, when executing an operation.

In the first set of experiments, we measure latency under three different LB contracts, all implemented in SYNCOPE. As expected, causal consistency and RMW experience respectively the highest and the lowest performance loss as the percentage of lost messages is increased[7]. With only a 4% percent message loss rate, we see 17% higher latency under an MR contract compared to RMW, and similarly 67% higher latency in CC compared to MR; with 10 percent message loss, the numbers are increased to 18% and 87%.

Similarly, we repeated the experiment with 3 UB contracts. Here, a *causal visibility* (CV) contract (i.e. $\forall a.a \xrightarrow{(\mathsf{so}\cup\mathsf{vis})^*;\mathsf{vis}} \hat{\eta} \Rightarrow a \xrightarrow{\mathsf{vis}} \hat{\eta}$), yields the most stale data when the percentage of lost messages is increased, whereas staleness in MW is the lowest and is barely affected. We report 3% (6%) difference between staleness of data under MW and 2VIS, and 4% (7%) difference between 2VIS and CV, at four (ten) percent message loss rate.

[7] In fact, they define the strongest and the weakest LB dependency relations expressable in our language: ($\xrightarrow{\mathsf{so}}$) and ($\xrightarrow{(\mathsf{so}\cup\mathsf{vis})^*}$)

Finally, in order to provide evidence on the practicality of SYNCOPE, we implemented an ad-hoc mechanism to prevent the lost-updates anomaly, for a simple counter application. Fig. 9(c) shows the latency results of this application compared to the same in SYNCOPE, under the same setting as before (albeit with no message loss). We see 78% higher latency for the handwritten code compared to SYNCOPE with 50 concurrent clients. The reason hand-written code performs worse than SYNCOPE is because it relies on certain Cassandra-specific features that are not optimized for the current purpose (each time a new session is created, it performs a strongly-consistent schema alteration to create a new column that records the sequence number of the last update from that session to every object). Short of replicating the SYNCOPE's shim layer-based approach, we found no other way to enforce the required consistency guarantee beyond the one we implemented. Beyond the performance numbers, the ad-hoc approach to consistency enforcement is also qualitatively inferior since it required significant re-engineering of the counter application (nearly 40% of the original code needed refractoring to perform book-keeping needed for the ad-hoc approach to work). With SYNCOPE, however, no refractoring was needed to enforce the required consistency.

8 Related Works

Distributed data structures composed of operation-based replicated data types (RDTs) [5, 11] have been utilized in a number of real-world systems [3, 6]. However, these systems are developed without assuming any principled notions of consistency, and thus have goals different from SYNCOPE. Like [2], SYNCOPE's focus is entirely on consistency management, and leaves issues of liveness and durability management to the underlying data store.

The specification of consistency requirements of replicated data-objects have been studied in several works [1, 4, 8], where multiple sufficient conditions and analysis techniques are proposed to detect potential coordination points in programs to enforce different notions of consistency. SYNCOPE shares similar goals, manifested within a lightweight runtime enforcement mechanism that dynamically validates fine-grained consistency specifications.

Numerous systems [2, 7, 9, 10, 12, 14] define and implement various levels of consistency guarantees in order to protect applications from anomalies admitted under EC. [7] presents a verified implementation for a causally consistent store, assuming a system model with *session stickiness*, where unlike SYNCOPE, operations from a session are always routed to the same replica. The idea of a causally consistent shim layer on top of an off-the-shelf ECDS, is proposed in [2] and is also utilized in [12], which offers three coarse-grained levels of consistency. SYNCOPE extends the shim layer in [12] by maintaining *multiple* fine-grained weak consistency levels.

References

[1] P. Alvaro, N. Conway, J. M. Hellerstein, and W. R. Marczak. "Consistency analysis in Bloom: A CALM and collected approach". In: *In Proceedings 5th Biennial Conference on Innovative Data Systems Research*. 2011, pp. 249–260.

[2] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica. "Bolt-on Causal Consistency". In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. SIGMOD '13. New York, New York, USA: ACM, 2013, pp. 761–772. ISBN: 978-1-4503-2037-5. DOI: 10.1145/2463676.2465279. URL: http://doi.acm.org/10.1145/2463676.2465279.

[3] M. Balakrishnan, D. Malkhi, T. Wobber, M. Wu, V. Prabhakaran, M. Wei, J. D. Davis, S. Rao, T. Zou, and A. Zuck. "Tango: Distributed Data Structures over a Shared Log". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. Farminton, Pennsylvania: ACM, 2013, pp. 325–340. ISBN: 978-1-4503-2388-8. DOI: 10.1145/2517349.2522732. URL: http://doi.acm.org/10.1145/2517349.2522732.

[4] V. Balegas, S. Duarte, C. Ferreira, R. Rodrigues, N. Preguiça, M. Najafzadeh, and M. Shapiro. "Putting Consistency Back into Eventual Consistency". In: *Proceedings of the Tenth European Conference on Computer Systems*. EuroSys '15. Bordeaux, France: ACM, 2015, 6:1–6:16. ISBN: 978-1-4503-3238-5. DOI: 10.1145/2741948.2741972. URL: http://doi.acm.org/10.1145/2741948.2741972.

[5] S. Burckhardt, A. Gotsman, H. Yang, and M. Zawirski. "Replicated Data Types: Specification, Verification, Optimality". In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '14. San Diego, California, USA: ACM, 2014, pp. 271–284. ISBN: 978-1-4503-2544-8. DOI: 10.1145/2535838.2535848. URL: http://doi.acm.org/10.1145/2535838.2535848.

[6] A. Lakshman and P. Malik. "Cassandra: A Decentralized Structured Storage System". In: *SIGOPS Oper. Syst. Rev.* 44.2 (Apr. 2010), pp. 35–40. ISSN: 0163-5980. DOI: 10.1145/1773912.1773922. URL: http://doi.acm.org/10.1145/1773912.1773922.

[7] M. Lesani, C. J. Bell, and A. Chlipala. "Chapar: Certified Causally Consistent Distributed Key-value Stores". In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '16. St. Petersburg, FL, USA: ACM, 2016, pp. 357–370. ISBN: 978-1-4503-3549-2. DOI: 10.1145/2837614.2837622. URL: http://doi.acm.org/10.1145/2837614.2837622.

[8] C. Li, J. Leitão, A. Clement, N. Preguiça, R. Rodrigues, and V. Vafeiadis. "Automating the Choice of Consistency Levels in Replicated Systems". In: *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*. USENIX ATC'14. Philadelphia, PA: USENIX Association,

2014, pp. 281–292. ISBN: 978-1-931971-10-2. URL: http://dl.acm.org/citation.cfm?id=2643634.2643664.

[9] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. "Making Geo-replicated Systems Fast As Possible, Consistent when Necessary". In: *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*. OSDI'12. Hollywood, CA, USA: USENIX Association, 2012, pp. 265–278. ISBN: 978-1-931971-96-6. URL: http://dl.acm.org/citation.cfm?id=2387880.2387906.

[10] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. "Flexible Update Propagation for Weakly Consistent Replication". In: *SIGOPS Oper. Syst. Rev.* 31.5 (Oct. 1997), pp. 288–301. ISSN: 0163-5980. DOI: 10.1145/269005.266711. URL: http://doi.acm.org/10.1145/269005.266711.

[11] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. "Conflict-free Replicated Data Types". In: *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*. SSS'11. Grenoble, France: Springer-Verlag, 2011, pp. 386–400. ISBN: 978-3-642-24549-7. URL: http://dl.acm.org/citation.cfm?id=2050613.2050642.

[12] K. Sivaramakrishnan, G. Kaki, and S. Jagannathan. "Declarative Programming over Eventually Consistent Data Stores". In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '15. Portland, OR, USA: ACM, 2015, pp. 413–424. ISBN: 978-1-4503-3468-6. DOI: 10.1145/2737924.2737981. URL: http://doi.acm.org/10.1145/2737924.2737981.

[13] D. B. Terry, A. J. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. W. Welch. "Session Guarantees for Weakly Consistent Replicated Data". In: *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*. PDIS '94. Washington, DC, USA: IEEE Computer Society, 1994, pp. 140–149. ISBN: 0-8186-6400-2. URL: http://dl.acm.org/citation.cfm?id=645792.668302.

[14] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh. "Consistency-based Service Level Agreements for Cloud Storage". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. Farminton, Pennsylvania: ACM, 2013, pp. 309–324. ISBN: 978-1-4503-2388-8. DOI: 10.1145/2517349.2522731. URL: http://doi.acm.org/10.1145/2517349.2522731.

A Modified Haskell Program

```
data Sess = Bob | Alice
type ID = (Sess,Int)
type Effect= (ID,String)
type State = (String,Int,Int)

read :: ID -> State -> String
read (sess,seq) (st,sq1,sq2) =
 case sess of
  Bob ->   if (seq==sq1+1) then st
           else read (sess,seq)(st,sq1,sq2)
  Alice -> if (seq==sq2+1) then st
           else read (sess,seq)(st,sq1,sq2)
```

```
apply :: State -> Effect -> State
apply (st,sq1,sq2) ((sess,seq),cm) =
  case sess of
    Bob ->   if (sq1==seq-1)
             then (st++cm,sq1+1,sq2)
             else (st,sq1,sq2)
    Alice -> if (sq2==seq-1)
             then (st++cm,sq1,sq2+1)
             else (st,sq1,sq2)
```

The guarded application to prevent the lost-updates anomaly, assuming 2 known clients Bob and Alice. Even with 2 known clients, the application has become much more complex with changed logic. This is much worse in reality where clients constantly join and leave the system

B Proofs

Here, we present detailed proofs of the theorems of the paper by first presenting a useful lemma.

Lemma 1. *For all relations* $R \in$ `relation` *and execution steps:* $(\mathsf{E}, op_{<s,i>}) \xrightarrow{V} (\mathsf{E}', \eta)$ *interpretatin of* R *under* E *and* E' *(Simply denoted by* R *and* R'*) differ only on the single effect* η*, i.e.* $\forall a, b \neq \eta \Rightarrow (R'(a,b) \Leftrightarrow R(a,b))$.

Proof. We prove $\Rightarrow$, the other direction can be shown similarly. We have the following goal and hypotheses:

$$\begin{array}{l} H_0 : (\mathsf{E}, op_{<s,i>}) \xrightarrow{V} (\mathsf{E}', \eta) \\ H_1 : a, b \neq \eta \\ H_2 : R'(a,b) \\ G_0 : R(a,b) \end{array}$$

Now by destructing R, we get the followings, in the only non-trivial case:

$$\begin{array}{l} H_3 : (\mathsf{trunc}(R); r)'(a,b) \\ G_1 : (\mathsf{trunc}(R); r)(a,b) \end{array}$$

which by rewriting the definition in H_4 and G_1, we get that y exists s.t.

$$\begin{array}{l} H_4 : (\mathsf{trunc}(R))'(a,y) \\ H_5 : r'(y,b) \\ G_2 : \exists x.(\mathsf{trunc}(R))(a,x) \wedge (r)(x,b) \end{array}$$

Now by instantiating G_2 with y and by inversion:

$$G_3 : (\mathsf{trunc}(R))(a, y)$$
$$G_4 : r(y, b)$$

Now by induction on the length of the relation R, G_3 is trivially proved and we are left with the following:

$$H_5 : r'(y, b)$$
$$G_4 : r(y, b)$$

Now by inversion on H_0 we get two cases. We show the [LB EXEC] case, and the other case can be shown similarly (only difference is V being replaced by V', which has no effect on the proof):

$$H_7 : \mathsf{vis}' = \mathsf{vis} \cup V \times \{\eta\}$$
$$H_8 : \mathsf{so}' = \mathsf{so} \cup (\mathsf{A}_{(\mathsf{SessID}=s)} \times \{\eta\})$$

Now, because of H_1 (and the fact that $y \neq \eta$) it is easy to get the following from H_7 and H_8:

$$H_9 : \mathsf{vis}'(y, b) \Rightarrow \mathsf{vis}(y, b)$$
$$H_{10} : \mathsf{so}'(y, b) \Rightarrow \mathsf{so}(y, b)$$

which directly prove G_4, in both cases derived by destructing r.

B.1 Proof of Theorem 1

(Part i)

We have the following two hypotheses and the goal:

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

Rewriting the definition in G_0 results in the following. We denote the interpretation of R under E' as R':

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

By intros we have:

$$H_2 : b \in V \cup \{\eta\}$$
$$H_3 : a \in E'.A$$
$$H_4 : R'(a, b)$$
$$G_2 : a \in V \cup \{\eta\}$$

by inversion on H_0, there is two cases, in case [UB EXEC] we have the following:

$$T_1 : V' \vdash (\mathsf{E}, op_{<s,i>}) \hookrightarrow (\mathsf{E}', \eta)$$

by inversion on T_1 we will have the following:

$$T_2 : E'.A = E.A \cup \{\eta\}$$

Since the other case ([LB EXEC]) also includes similar premises which yields T_2, we can add it to the hypotheses:

$$H_5 : E'.A = E.A \cup \{\eta\}$$

by rewriting H_5 in H_3 and by inversion, we get two cases: $(a = \eta)$ and $(a \in E.A)$. The first case immediatly proves G_2, so we only consider the second case where we have:

$$H_6 : a \in E.A$$

Now, by inversion on H_2, we have two cases:

– **Case 1:**

$$b \in V$$

by inversion in H_1 we have:

$$H_7 : \forall(x \in V).\forall(y \in E.A).R(y,x) \Rightarrow y \in V$$

and by instantiating it with a and b:

$$H_8 : R(a,b) \Rightarrow a \in V$$

Now by applying the lemma 1 on H_4 we get that $R(a,b)$ holds (since $a, b \neq \eta$), which can be applied on H_8 to get $a \in V$ which proves the goal G_2.

– **Case 2:**

$$\begin{array}{ll} & H_9 : \ b = \eta \\ (\texttt{by rewriting } H_9 \texttt{ in } H_4) & H_{10} : R'(a,\eta) \end{array}$$

Now we use inversion on H_0 and get two cases: [LB EXEC] and [UB EXEC]:
– **SCase** [LB EXEC]: we have H_{11} and H_{12} from the reduction rule premises:

$$\begin{array}{l} H_{11} : R_V^{-1}(\eta) = R_{E'.A}^{-1}(\eta) \\ H_{12} : R_V^{-1}(\eta) \subseteq V \end{array}$$

now from H_{10} we have H_{13} which can be rewritten by H_{11} to get H_{H14}:

$$\begin{array}{l} H_{13} : a \in R_{E'.A}^{-1}(\eta) \\ H_{14} : a \in R_V^{-1}(\eta) \end{array}$$

The goal G_2 is now proved from H_{12} and H_{14}.
– **SCase** [UB EXEC]: We have the following from the premises:

$$\begin{array}{l} H_{15} : V' = \lfloor V \rfloor_{\texttt{max}} \\ H_{16} : V' \subseteq V \end{array}$$

By destructing R, the only non-trivial cases are $(R = \mathsf{trunc}(R); \mathsf{vis})$ and $(R = \mathsf{vis})$:
SSCase $(R = \mathsf{trunc}(R); \mathsf{vis})$:
From H_{10} we get H_{17} which based on the definition, yields that there exists c such that H_{18}, H_{19} and H_{20} hold:

$$\begin{array}{l} H_{17} : a \in (\mathsf{trunc}(R)'; \mathsf{vis}')_{E'.A}^{-1}(\eta) \\ H_{18} : c \in \mathsf{vis}'^{-1}(\eta) \\ H_{19} : a \in \mathsf{trunc}(R)'^{-1}(c) \\ H_{20} : \mathsf{vis}'^{-1}(\eta) \subseteq E'.A \end{array}$$

from H_{15} we have:

$$H_{21} : (\mathsf{trunc}(R))_V^{-1}(V') \subseteq V'$$

Now from H_{18} (and T_1) it is straightforward to get:

$$H_{22} : c \in V'$$

which after appying the lemma 1 on H_{19}, and by H_{21} yields the following, which proves the goal G_2:

$$H_{23} : a \in V'$$

SSCase $(R = \mathsf{vis})$: From H_{10} we get that $\mathsf{vis}'(a,\eta)$, which -with a similar argument to the previous subcase- yields the following and the goal is proved:

$$H_{24} : a \in V'$$

QED.

(Part ii)

For this part we have the following hypothesis and the goal:

$$\begin{array}{l} H_0 : (\mathsf{E}, op_{<s,i>}) \xrightarrow{V} (\mathsf{E}', \eta) \\ G_0 : E' \models [\eta/\hat{\eta}] \end{array}$$

By inversion on H_0, we have two cases:
Case1 [UB EXEC]:

$$\begin{array}{l} H_1 : r_k = \mathsf{vis} \\ H_2 : V \subseteq E.A \\ H_3 : V' = \lfloor V \rfloor_{\mathsf{max}} \\ H_4 : V' \vdash (\mathsf{E}, op_{<s,i>}) \hookrightarrow (\mathsf{E}', \eta) \end{array}$$

The goal G_0 can also be rewritten as:

$$G_1 : E' \models \forall a.a \xrightarrow{R} \eta \Rightarrow a \xrightarrow{vis} \eta$$

Since the $E'.A$ gives the interpretation for the universe of quantification:

$$G_2 : \forall (a \in E'.A).E' \models a \xrightarrow{R} \eta \Rightarrow a \xrightarrow{vis} \eta$$

by intros:

$$\begin{array}{l} H_5 : a \in E'.A \\ G_3 : E' \models a \xrightarrow{R} \eta \Rightarrow a \xrightarrow{vis} \eta \end{array}$$

Now since $((\mathcal{M} \models A \Rightarrow B) \Leftrightarrow (\mathcal{M} \models A \Rightarrow \mathcal{M} \models B))$ we can rewrite G_3 as:

$$G_4 : (E' \models a \xrightarrow{R} \eta) \Rightarrow (E' \models a \xrightarrow{vis} \eta)$$

by intros:

$$\begin{array}{l} H_6 : E' \models a \xrightarrow{R} \eta \\ G_5 : E' \models a \xrightarrow{vis} \eta \end{array}$$

Now we use the interpretation given by E', to rewrite the relations as follows. Note that we denote the interpretation of R under E' as R' and $E.\mathsf{vis}$ as vis'.

$$\begin{array}{l} H_7 : R'(a, \eta) \\ G_6 : \mathsf{vis}'(a, \eta) \end{array}$$

by inversion on H_4:

$$H_8 : \mathsf{vis}' = \mathsf{vis} \cup V' \times \{\eta\}$$

Now since $\eta \notin E.A$, we get that $a \in V' \Rightarrow \mathsf{vis}'(a, \eta)$, which can be applied to G_6 to get the following:

$$G_7 : a \in V'$$

Now, destructing R yileds multiple cases, only one of which is non-trivial: $R = \mathsf{trunc}(R); \mathsf{vis}$, which can be rewritten in H_7 to get:

$$H_9 : (\mathsf{trunc}(R); \mathsf{vis})'(a, \eta)$$

Now we can rewrite the definition in H_9, and derive that there exists b such that:

$$H_{10} : \mathsf{trunc}(R)'(a, b)$$
$$H_{11} : \mathsf{vis}'(b, \eta)$$

Now using a similar argument, from H_8 and H_{11} we get:

$$H_{12} : b \in V'$$

Now by applying the lemma 1 on H_{10} we get:

$$H_{13} : \mathsf{trunc}(R)(a, b)$$

since we have $V' \in \lfloor V \rfloor$, we get the following:

$$H_{14} : \forall (x \in V').(\mathsf{trunc}(R))^{-1}_{E.A}(V') \Rightarrow x \in V'$$

which yields the following from H_{12} and H_{13}:

$$H_{15} : a \in V'$$

which proves the goal G_7.

Case2 [LB EXEC]:
We prove this case by induction on the length of the given relation R. We have the followings, from the premises of the reduction rule:

$$H_1 : r_k = \mathsf{so}$$
$$H_2 : V \subseteq E.A$$
$$H_3 : R^{-1}_V(\eta) = R^{-1}_{E.A}(\eta)$$
$$H_4 : R^{-1}_V(\eta) \subseteq V$$
$$H_5 : V \vdash (\mathsf{E}, op_{<s,i>}) \hookrightarrow (\mathsf{E}', \eta)$$

Using the same argument as the previous section, we get the following new goal and hypotheses:

$$H_6 : a \in E'.A$$
$$H_7 : R'(a, \eta)$$
$$G_1 : \mathsf{vis}'(a, \eta)$$

We now destruct R to get H_8 from H_7, and rewrite the definition in it to get the next two hypotheses. Note that by destructing R, there are only two non-trivial cases ($R = \mathsf{trunc}(R); \mathsf{so}$) and ($R = \mathsf{so}$), which we only consider the former, since the latter can be proved similarly:

$$H_8 : \ (\mathsf{trunc}(R); \mathsf{so})'(a, \eta)$$
$$H_9 : \ \mathsf{trunc}(R)'(a, b)$$
$$H_{10} : \mathsf{so}'(b, \eta)$$

Now, from the previous section we know that $(so')^{-1}(\eta) \subseteq V$ which yields the following from H_{10}:

$$H_{11} : b \in V$$

The goal is proved by the induction hypothesis, H_9 and H_{11}.
QED.

B.2 Proof of Theorem 2

We prove the theorem by contradiction:

$$\begin{array}{l} H_0 : (\mathsf{E}, op_{<s,i>}) \xrightarrow{V} (\mathsf{E}', \eta) \\ H_1 : a \in V \\ H_2 : (a, \eta) \notin E'.\mathsf{vis} \\ H_3 : (E'.A, E'.\mathsf{vis} \cup \{(a, \eta)\}, E'.\mathsf{so}) \models \psi[\eta/\hat{\eta}] \\ H_4 : (\mathsf{trunc}(R)_V^{-1}(a) = \mathsf{trunc}(R)_{E.A}^{-1}(a)) \\ G_0 : \bot \end{array}$$

Now we call $(E'.A, E'.\mathsf{vis} \cup \{(a, \eta)\}, E'.\mathsf{so})$ as E'' and derive the following from H_3:

$$H_5 : E'' \models \forall x.x \xrightarrow{R} \eta \Rightarrow x \xrightarrow{\mathbf{vis}} \eta$$

because E'' defines the universe of quantification (and since $E''.A = E'.A$), we get the following:

$$H_6 : \forall (x \in E'.A).E'' \models x \xrightarrow{R} \eta \Rightarrow x \xrightarrow{\mathbf{vis}} \eta$$

and is rewritten as the following:

$$H_7 : \forall (x \in E'.A).(E'' \models x \xrightarrow{R} \eta) \Rightarrow (E'' \models x \xrightarrow{\mathbf{vis}} \eta)$$

Now by inversion on H_0 we get two cases, one of which is trivial. We skip the formal proof for it but it is easy to see that in [LB exec] case, ALL effects in V are made visible to η, so the set is trivially maximal, i.e. H_1 and H_2 yield $\bot$. For the other case (UB exec), we get the following:

$$\begin{array}{l} H_8 : V' = \lfloor V \rfloor_{\mathsf{max}} \\ H_9 : V' \vdash (\mathsf{E}, op_{<s,i>}) \hookrightarrow (\mathsf{E}', \eta) \end{array}$$

by inversion on H_9 we get H_{10} and from that and from H_2, following a similar argument from the proof of theorem 1, we get H_{11}:

$$\begin{array}{l} H_{10} : \mathsf{vis}' = \mathsf{vis} \cup V' \times \{\eta\} \\ H_{11} : a \notin V' \end{array}$$

Now by denoting the interpretation of R under E'' as R'', H_7 can be rewritten as follows:

$$H_{12} : \forall (x \in E'.A).R''(x, \eta) \Rightarrow \mathsf{vis}''(x, \eta)$$

Now by inversion on H_8, we get the following:

$$\begin{array}{rl} & H_{13} : V' \in \lfloor V \rfloor \\ & H_{14} : \nexists V'' \in \lfloor V \rfloor .|V''| > |V'| \\ (\texttt{from H}_{13}) & H_{15} : V' \subseteq V \ \wedge \ (\mathsf{trunc}(R))_V^{-1}(V') \subseteq V' \wedge \\ & \qquad (\mathsf{trunc}(R))_V^{-1}(V') = (\mathsf{trunc}(R))_{E.A}^{-1}(V') \end{array}$$

Now we can destruct R, where we get multiple cases, only two of which are non-trivial, $(R = \mathsf{vis})$ and $(R = \mathsf{trunc}(R); \mathsf{vis})$

- **Case1**$(R = \mathsf{vis})$:
  $\mathsf{trunc}(R) = \texttt{null}$, thus V itself satisfies the requirements in H_{15} and we get that $(V = \lfloor V \rfloor_{\mathsf{max}})$ and the following holds:

  $$H_{16} : V = V'$$

  which results in contradiction from H_1 and H_{11}.

– **Case2**$(R = \mathsf{trunc}(R); \mathsf{vis})$:
Since $|V' \cup \{a\}| > |V'|$ we have the following:

$$H_{17} : (V' \cup \{a\}) \notin \lfloor V \rfloor$$

which based on the definition yields that the conditiones for holding the above relation are not true, i.e.

$$H_{18} : \neg((V' \cup \{a\}) \subseteq V \ \wedge \ (\mathsf{trunc}(R))^{-1}_{V}(V' \cup \{a\}) \subseteq (V' \cup \{a\}) \wedge (\mathsf{trunc}(R))^{-1}_{V}(V' \cup \{a\}) = (\mathsf{trunc}(R))^{-1}_{E.A}(V' \cup \{a\}))$$

or equally:

$$H_{19} : (V' \cup \{a\}) \not\subseteq V \vee (\mathsf{trunc}(R))^{-1}_{V}(V' \cup \{a\}) \not\subseteq (V' \cup \{a\}) \vee (\mathsf{trunc}(R))^{-1}_{V}(V' \cup \{a\}) \neq (\mathsf{trunc}(R))^{-1}_{E.A}(V' \cup \{a\})$$

By inversion on the above, we get three cases, two of which are trivial. The last conjunct can't hold because of H_4 and the first one also contradics with H_1 and H_{15}. Thus, we are left with only one case:

$$H_{20} : (\mathsf{trunc}(R))^{-1}_{V}(V' \cup \{a\}) \not\subseteq (V' \cup \{a\})$$

Now, from the second conjunct in H_{15} we know that it should be the case that:

$$(\mathtt{from}\ \mathtt{H_{15}} : (\mathtt{trunc}(\mathtt{R}))^{-1}_{\mathtt{V}}(\mathtt{V'}) \subseteq \mathtt{V'}) \qquad H_{21} : ((\mathsf{trunc}(R))^{-1}_{V}(a) \not\subseteq (V' \cup \{a\}))$$

The above hypothesis yields the existance of $c \neq a$ such that:

$$H_{22} : c \in (\mathsf{trunc}(R))^{-1}_{V}(a)$$
$$H_{23} : c \notin V'$$

Now, by rewriting $(R = \mathsf{trunc}(R); \mathsf{vis})$ in H_{12} we get H_{24}, which can be rewritten again into H_{25} from the definition:

$$H_{24} : \forall(x \in E'.A).((\mathsf{trunc}(R); \mathsf{vis})''(x, \eta) \Rightarrow \mathsf{vis}''(x, \eta))$$
$$H_{25} : \forall(x \in E'.A).(\exists b.\mathsf{trunc}(R)''(x, b) \wedge \mathsf{vis}''(b, \eta) \Rightarrow \mathsf{vis}''(x, \eta))$$

Now, we instantiate H_{25} with $x = c$:

$$H_{26} : \exists b.\mathsf{trunc}(R)''(c, b) \ \wedge \ \mathsf{vis}''(b, \eta) \Rightarrow \mathsf{vis}''(c, \eta)$$

we can replace $\mathsf{trunc}(R)''$ with $\mathsf{trunc}(R)'$ in above definition, since from H_3, the only difference in interpretation under E' and E'' is the extra element (a, η) in $E''.\mathsf{vis}$ which does not effect $\mathsf{trunc}(R)''(c, b)$:

$$H_{27} : \exists b.\mathsf{trunc}(R)'(c, b) \ \wedge \ \mathsf{vis}''(b, \eta) \Rightarrow \mathsf{vis}''(c, \eta)$$

Moreover, since $c \neq a$, we can replace $\mathsf{vis}''(c, \eta)$ with $\mathsf{vis}'(c, \eta)$:

$$H_{28} : \exists b.\mathsf{trunc}(R)'(c, b) \ \wedge \ \mathsf{vis}''(b, \eta) \Rightarrow \mathsf{vis}'(c, \eta)$$

From H_{15} and H_{22} we get H_{29}, and H_{30} also holds trivially from H_3:

$$H_{29} : \mathsf{trunc}(R)'(c, a)$$
$$H_{30} : \mathsf{vis}''(a, \eta)$$

which can be used in instantiation of H_{28} with $b = a$ and derive the following:

$$H_{31} : \mathsf{vis}'(c, \eta)$$

However, we know -from the previously explained argument- that H_{31} results in H_{32}, which results in contradiction with H_{23}.

$$H_{32} : c \in V'$$

QED.

B.3 Proof of Theorem 3

Before proving the theorem, we first present and prove a useful lemma and then we will present a new definition, regarding sets of effects.

Lemma 2. *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.

$$H_0 : Sis\psi\texttt{-consistent}$$
$$G_0 : \forall(x \in S).R_S^{-1}(x) \subseteq S$$

after intros:

$$H_1 : x \in S$$
$$G_1 : R_S^{-1}(x) \subseteq S$$

inversion on H_0 gives the following:

$$H_2 : \forall(\eta \in S).\forall(a \in E.A).R(a, \eta) \Rightarrow a \in S$$

which can be rewritten to:

$$H_3 : \forall(\eta \in S).R^{-1}(\eta) \subseteq S$$

however, since $S \subseteq E.A$ then[8]:

$$H_4 : \forall(a \in E.A).R_S^{-1}(a) \subseteq R^{-1}(a)$$

Now we can instantiate H_3 and H_4 into:

$$H_5 : R^{-1}(x) \subseteq S$$
$$H_6 : R_S^{-1}(x) \subseteq R^{-1}(x)$$

which trivially yields G_1 and the proof is completed.
QED.

Definition 1. *We define the completment of a given set of effects S (under an execution state E) as the super set of S, containing ALL the mid-level effects required to determine ALL the dependencies of the effects in S, i.e.*

$$S' \in \lceil S \rceil \iff R_{S'}^{-1}(S) = R_{E.A}^{-1}(S)$$

Now, using the above theorem and lemma, we present the proof of the theorem 3, which starts by listing the following hypotheses and the goal:

$$H_0 : S \vdash (\mathsf{E}, op_{<s,i>}) \hookrightarrow (\mathsf{E}', \eta)$$
$$H_1 : S \cup \{\eta\} \textit{ is } \psi\texttt{-consistent}$$
$$G_0 : \exists E''.\exists \eta'.\exists V.((\mathsf{E}, op_{<s,i>}) \xrightarrow{V} (\mathsf{E}'', \eta'))$$

Now, by destructing R we get two non-trivial cases:

[8] we skip the formal proof of this claim, however, since the only difference in the definitions of R^{-1} and R_S^{-1} is the extra requirement about mid-level effects in the latter, it should be a subset of the former.

– **Case1**($R = \mathsf{trunc}(R); \mathsf{vis}$):
In this case[9], we generate the premises of the [UB EXEC] to achieve the goal as follows. Firstly, we define S' and present η':

$$\begin{array}{l} H_3 : S' = \lfloor S \rfloor_{\mathtt{max}} \\ H_4 : \eta' = (s, op, F_{op}(S')) \end{array}$$

Moreover, we will define the followings, which will be used when presenting E'':

$$\begin{array}{l} H_5 : \mathsf{so}'' = \mathsf{so} \cup A_{(\mathtt{SessID}=s)} \times \{\eta'\} \\ H_6 : \mathsf{vis}'' = \mathsf{vis} \cup S' \times \{\eta'\} \\ H_7 : A'' = E.A \cup \{\eta'\} \end{array}$$

Now we present V and E'' as follows and rewrite the goal:

$$\begin{array}{l} H_8 : V = S \\ H_9 : E'' = (A'', \mathsf{so}'', \mathsf{vis}'') \\ G_1 : (\mathsf{E}, op_{<s,i>}) \xrightarrow{V} (\mathsf{E}'', \eta') \end{array}$$

by applying [UB EXEC] on G_1 we get the following new goals (after rewriting H_9 and H_3):

$$\begin{array}{l} G_2 : r_k = \mathsf{vis} \\ G_3 : S \subseteq E.A \\ G_4 : S' = \lfloor S \rfloor \\ G_5 : S' \vdash (\mathsf{E}, op_{<s,i>}) \hookrightarrow (\mathsf{E}'', \eta') \end{array}$$

first three goals are proved via the assumptions, and the last one can be easily shown to hold by applying [OPER] and deriving the following new goals:

$$\begin{array}{ll} G_6 : & S' \subseteq E.A \\ G_7 : & F_{op}(S') = v \\ G_8 : & \eta' = (s, op, v) \\ G_9 : & \eta \notin S' \\ G_{10} : & E''.A = E.A \cup \{\eta'\} \\ G_{11} : & E''.\mathsf{vis} = E.\mathsf{vis} \cup S' \times \{\eta\} \\ G_{12} : & E''.\mathsf{so} = E.\mathsf{so} \cup (A_{(\mathtt{SessID}=s)}) \times \{\eta\} \end{array}$$

all the above goals have already been shown in the assumptions and the case is proved.

– **Case2**($R = \mathsf{trunc}(R); \mathsf{so}$):
Similarly in this case we define the following:

$$H_{13} : V = \lceil S \cup \{\eta\} \rceil$$

which yields:

$$H_{14} : \forall (x \in S \cup \{\eta\}).R_V^{-1}(x) = R_{E'.A}^{-1}(x)$$

and also:

$$H_{15} : R_V^{-1}(\eta) = R_{E'.A}^{-1}(\eta)$$

Similar to the previous case, we now define the followings:

$$\begin{array}{l} H_{16} : \eta' = (s, op, F_{op}(V)) \\ H_{17} : \mathsf{so}'' = \mathsf{so} \cup A_{(\mathtt{SessID}=s)} \times \{\eta'\} \\ H_{18} : \mathsf{vis}'' = \mathsf{vis} \cup V \times \{\eta'\} \end{array}$$

[9] Note that in this case the goal G_0, trivially holds. That is because the contract in this case is [UB], which represents executions whitout blocking or waiting, that can always make progress by showing *some* set of effects to the operations

Now we present E'' as follows and rewrite the goal:

$$H_{19} : E'' = (A'', \mathsf{so}'', \mathsf{vis}'')$$
$$G_1 : \ (\mathsf{E}, op_{<s,i>}) \xrightarrow{V} (\mathsf{E}'', \eta')$$

by applying [LB EXEC] on G_1 we get the following new goals

$$G_2 : r_k = \mathsf{so}$$
$$G_3 : V \subseteq E.A$$
$$G_4 : R_V^{-1}(\eta') = R^{-1}_{E''.A}(\eta')$$
$$G_5 : R_V^{-1}(\eta') \subseteq V$$
$$G_6 : V \vdash (\mathsf{E}, op_{<s,i>}) \hookrightarrow (\mathsf{E}'', \eta')$$

Now, G_2 and G_3 are trivially proved from the assumptions, and G_6 also can be easily proved following the argument from the previous case. We prove G_4 and G_5, by a new claim that $R^{-1}_{E'.A}(\eta) = R^{-1}_{E''.A}(\eta')$ which will be proved separately. Thus, we can rewrite the goals and add the new claim:

$$G_7 : R^{-1}(\eta) = R^{-1}_{E'.A}(\eta)$$
$$G_8 : R^{-1}(\eta) \subseteq V$$
$$G_9 : R^{-1}_{E'.A}(\eta) = R^{-1}_{E''.A}(\eta')$$

Now G_7 is equal to the assumption H_{15}, and G_8 is the direct result of applying the lemma 2 on H_1. Now by rewriting $R = \mathsf{trunc}(R); \mathsf{so}$ in G_9 we have the following:

$$G_{10} : (\mathsf{trunc}(R); \mathsf{so})^{-1}_{E'.A}(\eta) = (\mathsf{trunc}(R); \mathsf{so})^{-1}_{E''.A}(\eta')$$

Now, note that the only difference in E' and E'' is in how the update the vis relation from E, the former makes the set S visible to the operation and the latter the set $\lceil S \cup \{\eta\} \rceil$. Now since the given relation R ends with an so relation, it is straigtforward to show that G_10 holds and thus the case (and the theorem) is proved.

QED.

C 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:

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

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. The formal definitions and the operation semantics are presented in the next page.

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

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

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

[Oper]

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

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

[Pool Refresh]

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

[DDP Refresh]

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

[Cache Refresh]

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

[LB Exec]

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

[UB Exec]

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

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