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. Modern-day distributed systems often use replication to improve communication latency and fault tolerance among geographically distributed servers and clients. But, replication compromises simplicity because it forces applications to weigh the correctness benefits of guaranteeing strong consistency against a potentially significant cost in performance. Because not all operations on a replicated data type require strongly consistent behavior, however, library designers ought be able to direct implementations to provide precisely the level of consistency needed, eschewing the costs of strong consistency except when absolutely necessary. Unfortunately, implementations typically support only a small set of consistency levels. Consequently, developers must either be content with mapping application requirements to a supported consistency level that may well enforce stronger guarantees than required, or hand weave a specialized protocol that more accurately matches application needs. The former approach sacrifices performance, while the latter introduces unwanted complexity.

In this paper, we describe a lightweight runtime system for weakly consistent distributed systems that dynamically customizes a consistency protocol based on a declarative axiomatic specification that reflects the necessary constraints any correct implementation must satisfy. Our technique generates a provably optimal runtime enforcement mechanism that imposes no additional communication or blocking overhead beyond what is required to satisfy the specification, thus freeing developers from writing bespoke and complex consistency protocols without having to sacrifice availability and performance by not doing so.

We demonstrate the applicability of our approach by automatically deriving enforcement mechanisms for various well-known weak consistency instantiations. Experimental results show that the performance of our automically derived mechanisms is comparable to specialized hand-written protocols, 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. **??**.

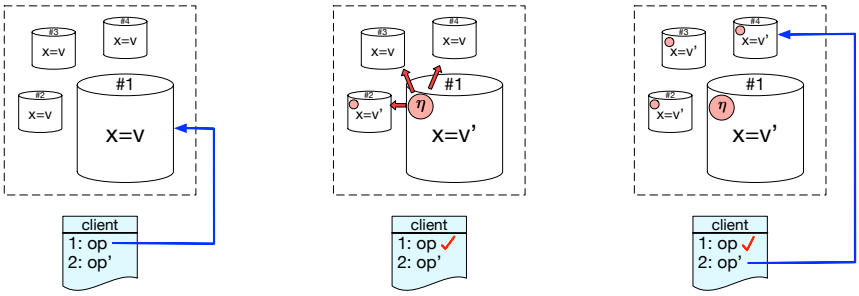
2 System Model

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

References

1. Clarke, F., Ekeland, I.: Nonlinear oscillations and boundary-value problems for Hamiltonian systems. Arch. Rat. Mech. Anal. 78, 315–333 (1982)
2. Clarke, F., Ekeland, I.: Solutions périodiques, du période donnée, des équations hamiltoniennes. Note CRAS Paris 287, 1013–1015 (1978)
3. Michalek, R., Tarantello, G.: Subharmonic solutions with prescribed minimal period for nonautonomous Hamiltonian systems. J. Diff. Eq. 72, 28–55 (1988)
4. Tarantello, G.: Subharmonic solutions for Hamiltonian systems via a $\mathbb{Z}_p$ pseudoindex theory. Annali di Matematica Pura (to appear)
5. Rabinowitz, P.: On subharmonic solutions of a Hamiltonian system. Comm. Pure Appl. Math. 33, 609–633 (1980)