Fine-grained Distributed Consistency Guarantees with Effect Orchestration

Kia Rahmani
Purdue University
USA
rahmank@purdue.edu

Gowtham Kaki
Purdue University
USA
gowtham@purdue.edu

Suresh Jagannathan
Purdue University
USA
suresh@cs.purdue.edu

ABSTRACT

Highly-available distributed applications typically require data to be replicated over geo-distributed stores that offer weak consistency guarantees by default. Unfortunately, undesirable behaviors may arise under weak consistency that can violate application correctness, forcing designers to either implement complex ad-hoc mechanisms to avoid these anomalies, or choose to run applications using stronger levels of consistency, sacrificing performance. In this paper, we describe a lightweight runtime 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. Experimental results show that the performance of our (provably optimal and safe) automatically derived fine-grained consistency enforcement mechanisms is better than common store-offered consistency guarantees.

Keywords

Runtime Safety; Weak Consistency; Key-value stores; Haskell

1. INTRODUCTION

Historically, the *de facto* system abstraction for developing distributed programs has typically included ACID[1] properties [17, 11]. However, for web-scale applications that need to be "always-on" even in the presence of network partitions, extensive synchronization overhead is often unacceptable [10]. Such applications are therefore usually designed to tolerate certain inconsistencies and thus to be commonly deployed on eventually consistent data stores.

In order to tolerate the level of inconsistency imposed by eventual consistency (EC), researchers have introduced numerous add-on consistency control mechanisms to equip applications with [22, 16, 2, 4, 5]. Unfortunately, such enforcement mechanisms are usually closely tied to the application logic, confounding standardization and reusability, while complicating application development and maintainability.

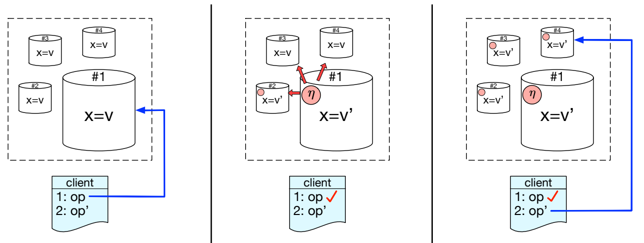
In this paper, we propose an alternative approach to weak consistency enforcement that circumvents the aforementioned issues. SYNCOPE is a lightweight runtime verification system for Haskell that allows application developers to take advantage of weak consistency without having to re-engineer their code to accommodate anomaly preemption mechanisms, by *declaratively* admitting consistency requirements via a specification language.

[1]Atomicity, Consistency, Isolation and Durability

The remainder of this paper is organized as follows: Section 2 introduces our system model and in Section 3 we discuss different approaches on distributed consistency enforcement which we collectively refer to as *orchestration* techniques. In sections 4 and 5 we present evidence for real-world applications requiring fine-grained consistency guarantees and experimental results suggesting the possibility of performance gain in off-the-shelf datastores deploying fine-grained consistency guarantees by using our tool. We present our conclusions and a review of the related works in section 8.

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. Fig. 1(a) and Fig. 1(b) illustrate this process.



(a) A client operation is routed to replica #1.

(b) An effect is created and propagated

(c) Second operation is routed to the replica #4

Figure 1: System Model

In order to admit all inconsistencies and anomalies associated with EC we assume no direct synchronization between replicas when an operation is executed and as a result, concurrent and possibly conflicting updates can be generated 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.

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 their session id and their sequence number in that particular session, which is used by replicas to record the set of all updates that are locally applied. Due to traffic balancing requirements, operations (even from the same session) might be routed to different replicas (Fig. 1(a) and Fig. 1(c)).

Lastly, we define two relations over effects. 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 asymetric 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.1(c) `vis` (η,η') holds, since η (the effect of op) has already been delivered and applied to the replica #4, when op' is executed and thus has influenced the generation of η'.

3. EFFECT ORCHESTRATION

In this section, we introduce SYNCOPE, a shim-layer for off-the-shelf EC key-value stores, which extends the basic consistency guarantees offered by them into a vast set of fine-grained guarantees (Fig. 2). SYNCOPE's design realizes two fundamental classes of run-time procedures on effects and operations, a combination of which is necessary to enforce fine-grained consistency requirements in a message-passing distributed system (such as our system model). In the following subsections, we will explain these techniques in detail.

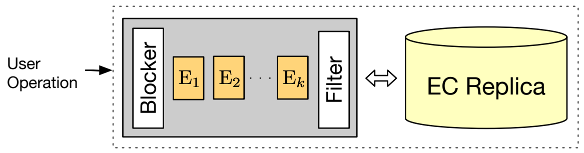


Figure 2: Shim Layer Layout

3.1 Blocking

SYNCOPE's first consistency enforcement technique is called *blocking*, which is a generalized mechanism to stall a user operation until a certain set of *necessary* dependencies of that operation become available at the underlying replica. SYNCOPE relies on the underlying store for effect propagation which guarantees the eventual delivery of all effects to all replicas. Consequently, all user operations will eventually proceed in an unpartitioned network[2].

A use case of this technique is when two operations from the same session are routed to different replicas. This can *sometimes* be problematic, e.g. when a user deposits some money into her bank account and then queries the balance. If the effect of the deposit is not available at the replica executing the query, she might think that the bank has stolen her money. In this scenario, the second operation should be blocked, until the effect of the deposit operation is delivered to the replica.

[2]Detailed and formal explaination of all mechanisms with formal proofs of liveness, correctness and optimality can be found the longer version of this paper [19]

The effects that an operation must wait for, are known as the dependency set of that operation. In the following parts we will introduce a formal language that allows users to define a rich set of fine-grained dependencies for each operation, in order to prevent undesired behaviors in their application.

3.2 Filtration

Similar to other systems offering add-on consistency guarantees for EC key-value stores, SYNCOPE's shim layer also maintains a *safe* environment in the memory by periodically (or on-demand) reading from the underlying EC database and adding the effects to the environment, only if the dependencies of them have already been added to the environment. For example, [6, 21] enforce causal consistency using similar shim layers, which maintain consistent caches that are guaranteed to contain a *causal cut* of the global execution history at any given point in time.

SYNCOPE generalizes this idea into a fine-grained filtration mechanism which maintains *multiple* safe environments (E_1, E_2,...), each for the use of a specific operation. Users then can specify arbitrary consistency guarantees in a language that is seeded with `so` and `vis` relations and define constraints on read operations that can be used to synthesize appropriate filtration mechanisms.

For example, in a microblogging application where Bob tweets two consecutive related messages B_1 and B_2, Alice who is connected to a different replica, should not be able to read Bob's second message without witnessing the first. Fig. 3 (left) shows this scenario on an unprotected EC system which permits this undesired anomaly. The middle and right parts of the figure, however, depict how a simple filtration mechanism can prevents Alice from seeing Bob's second message by allowing B_2 into the memory only after B_1 is also available. Even though filtration imposes extra staleness on the underlying database (e.g. in the middle part of Fig.3), it is very effective in eliminating many anomalous behaviors of applications running under EC.

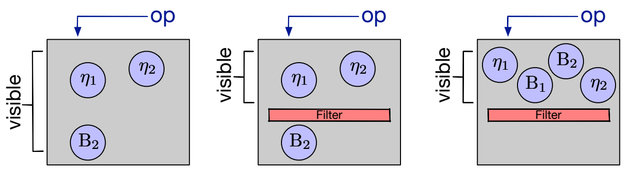


Figure 3: Filteration

3.3 Specification Language

The formal syntax of our specification (or contract) language, presented in Fig. 4, allows definitions 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.

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

Figure 4: Specification Language

The language is seeded with so and vis, respectively representing session order and visibility relations over effects, and defines dependency `relation` as a sequence[3] 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. [22] (see e.g., Fig. 5).

Guarantee	Contract
Read My Writes (RMW)	$\forall a.a \xrightarrow{\mathsf{so}} \hat{\eta} \Rightarrow a \xrightarrow{\mathsf{vis}} \hat{\eta}$
Monotonic Writes (MW)	$\forall a.a \xrightarrow{\mathsf{so;vis}} \hat{\eta} \Rightarrow a \xrightarrow{\mathsf{vis}} \hat{\eta}$
Monotonic Reads (MR)	$\forall a.a \xrightarrow{\mathsf{vis;so}} \hat{\eta} \Rightarrow a \xrightarrow{\mathsf{vis}} \hat{\eta}$
Transitive Visibility (2VIS)	$\forall a.a \xrightarrow{\mathsf{vis;vis}} \hat{\eta} \Rightarrow a \xrightarrow{\mathsf{vis}} \hat{\eta}$

Figure 5: Contract Examples

4. PRACTICALITY

In this section, we report on benchmark applications that utilize fine-grained weak consistency requirements, expressable in SYNCOPE's specification language. Fig. 6 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 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 Visibility (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 to their sheer number; even if we ignore bespoke consistency requirements, there are 15 combinations of just the 4 aforementioned consistency guarantees.

Causal consistency (CC), the strongest of the weak consistency guarantees, is often used as a metaphorical one-size to fit all weak consistency requirements (including all the benchmarks above). Notably, none of the operations we analyzed intrinsically requires CC and the undesired anomalous behaviors could be prevented by combinations of weaker consistency levels, which SYNCOPE can enforce exactly as they are specified.

5. EVALUATION

SYNCOPE is implemented as an extension to a GHC Haskell add-on called Quelea [21]. Quelea maintains a causally consistent [6] cache on top of Cassandra, and *all* operations whose contract is satisfied under causal consistency, are performed witnessing that cache (even if they require weaker guarantees.).

In SYNCOPE, we maintain a generic cache in which operations maintained by the cache are associated with tags, and are allowed to witnesses only the subset of effects in the cache that also have 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.

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

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
Microblog	MW, RMW	A Twitter-like application modeled after Twissandra [1]
Rubis	RMW, RMW∧2VIS	eBay-like application with browsing, supporting user wallet

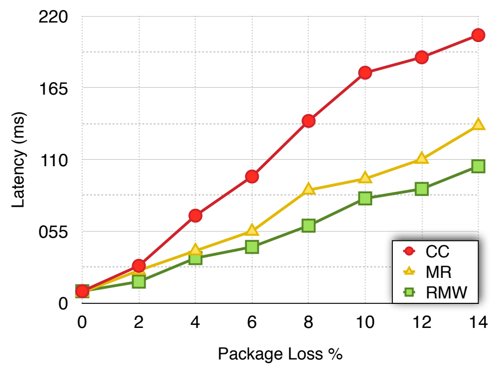
Figure 6: Fine-grained consistency requirement in benchmark programs

Considering the arbitrary length of the dependency relations that may be generated and the fact that verifying the presence of dependencies for an effect might fail for an unbounded number of trials until all dependencies arrive, special care must be taken to ensure performance does not grade at scale. We implemented a number of techniques to improve cache efficiency such as memoization 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 has already been verified. Consequently, when verification fails, we can avoid checking previously computed and known dependencies when subsequent effects arrive. SYNCOPE's runtime, by performing periodic DDP refreshes, tries to assign larger DDP values to each effect when more dependencies arrive at the replica. The details of this technique, captured as an operational semantics in available in the longer version of this paper [19].

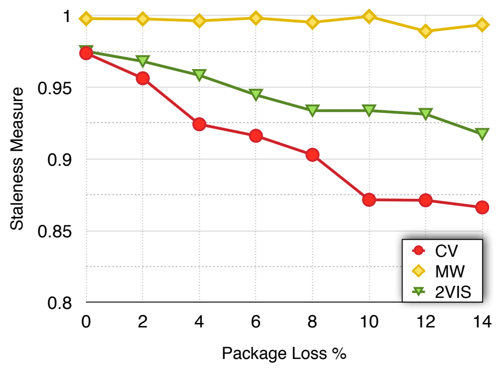
We have deployed SYNCOPE on a cloud cluster, consisting of three fully replicated Cassandra replicas, running on separate machines within the same datacenter. Each machine is instantiated with a SYNCOPE shim layer, that responds to clients, which are instantiated on a VM co-located with one of the replicas on a machine. We deploy the cluster on three `m4.4xlarge` Amazon EC2 instances 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 and pure 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. 7(a) and 7(b) represent our experimental results, with



(a) Latency



(b) Staleness

Figure 7: Performance Comparison

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 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. 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 other contracts to measure the staleness imposed by them. 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.

Our results are strong evidence for practicality of the implementation of arbitrary weak consistency guarantees in off-the-shelf EC key-value stores, which can greatly benefit from losing the unnecessary reliable inter-replica connections for certain applications.

6. RELATED WORKS AND CONCLUSION

Distributed data structures composed of operation-based replicated data types (RDTs) [9, 20] have been utilized in a number of real-world systems [7, 12]. However, these systems are developed without assuming any principled notions of consistency, and thus have goals different from SYNCOPE. Like [6], 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 [14, 8, 3], where multiple sufficient conditions and analysis techniques are proposed to detect potential coordination points in programs to enforce different notions of consistency. SYNCOPEshares similar goals, manifested within a lightweight runtime enforcement mechanism that dynamically validates fine-grained consistency specifications.

Numerous systems [15, 18, 23, 13, 6, 21] define and implement various levels of consistency guarantees in order to protect applications from anomalies admitted under EC. [13] 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 [6] and is also utilized in [21], which offers three coarse-grained levels of consistency. SYNCOPEextends the shim layer in [21] by maintaining *multiple* fine-grained weak consistency levels.

This paper presents SYNCOPE, a lightweight runtime mechanism and specification framework for enforcing fine-grained consistency contracts in eventually consistent distributed systems. Our design is provably optimal and safe, and experimental results indicate that automatic consistency validation using the techniques described here outperforms *ad hoc* manual solutions. We believe these results pave the way for strengthening any off-the-shelf distributed data store with consistency validation support for free.

7. REFERENCES

[1] twissandra. https://github.com/twissandra/twissandra. Accessed: 2017-05-11.

[2] AHAMAD, M., NEIGER, G., BURNS, J. E., KOHLI, P., AND HUTTO, P. W. Causal memory: definitions, implementation, and programming. *Distributed Computing 9*, 1 (1995), 37–49.

[3] ALVARO, P., CONWAY, N., HELLERSTEIN, J. M., AND MARCZAK, W. R. Consistency analysis in bloom: A calm and collected approach. In *In Proceedings 5th Biennial Conference on Innovative Data Systems Research* (2011), pp. 249–260.

[4] BAILIS, P., DAVIDSON, A., FEKETE, A., GHODSI, A., HELLERSTEIN, J. M., AND STOICA, I. Highly available transactions: Virtues and limitations. *Proc. VLDB*

Endow. 7, 3 (Nov. 2013), 181–192.

[5] Bailis, P., Fekete, A., Hellerstein, J. M., Ghodsi, A., and Stoica, I. Scalable atomic visibility with ramp transactions. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2014), SIGMOD '14, ACM, pp. 27–38.

[6] Bailis, P., Ghodsi, A., Hellerstein, J. M., and Stoica, I. Bolt-on causal consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2013), SIGMOD '13, ACM, pp. 761–772.

[7] Balakrishnan, M., Malkhi, D., Wobber, T., Wu, M., Prabhakaran, V., Wei, M., Davis, J. D., Rao, S., Zou, T., and Zuck, A. Tango: Distributed data structures over a shared log. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 325–340.

[8] Balegas, V., Duarte, S., Ferreira, C., Rodrigues, R., Preguiça, N., Najafzadeh, M., and Shapiro, M. Putting consistency back into eventual consistency. In *Proceedings of the Tenth European Conference on Computer Systems* (New York, NY, USA, 2015), EuroSys '15, ACM, pp. 6:1–6:16.

[9] Burckhardt, S., Gotsman, A., Yang, H., and Zawirski, M. Replicated data types: Specification, verification, optimality. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2014), POPL '14, ACM, pp. 271–284.

[10] Gilbert, S., and Lynch, N. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News 33*, 2 (June 2002), 51–59.

[11] Herlihy, M., and Wing, J. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems 12*, 3 (July 1990), 463–492.

[12] Lakshman, A., and Malik, P. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev. 44*, 2 (Apr. 2010), 35–40.

[13] Lesani, M., Bell, C. J., and Chlipala, A. Chapar: Certified causally consistent distributed key-value stores. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2016), POPL '16, ACM, pp. 357–370.

[14] Li, C., Leitão, J. a., Clement, A., Preguiça, N., Rodrigues, R., and Vafeiadis, V. Automating the choice of consistency levels in replicated systems. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2014), USENIX ATC'14, USENIX Association, pp. 281–292.

[15] Li, C., Porto, D., Clement, A., Gehrke, J., Preguiça, N., and Rodrigues, R. Making geo-replicated systems fast as possible, consistent when necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2012), OSDI'12, USENIX Association, pp. 265–278.

[16] Lloyd, W., Freedman, M. J., Kaminsky, M., and Andersen, D. G. Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 401–416.

[17] Papadimitriou, C. H. The serializability of concurrent database updates. *J. ACM 26*, 4 (1979), 631–653.

[18] Petersen, K., Spreitzer, M. J., Terry, D. B., Theimer, M. M., and Demers, A. J. Flexible update propagation for weakly consistent replication. *SIGOPS Oper. Syst. Rev. 31*, 5 (Oct. 1997), 288–301.

[19] Rahmani, K., Kaki, G., and Jagannathan, S. syncope: Automatic Enforcement of Distributed Consistency Guarantees. Tech. rep., Purdue University, DEPARTMENT OF COMPUTER SCIENCE, 05 2017.

[20] Shapiro, M., Preguiça, N., Baquero, C., and Zawirski, M. Conflict-free replicated data types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems* (Berlin, Heidelberg, 2011), SSS'11, Springer-Verlag, pp. 386–400.

[21] Sivaramakrishnan, K., Kaki, G., and Jagannathan, S. Declarative programming over eventually consistent data stores. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2015), PLDI '15, ACM, pp. 413–424.

[22] Terry, D. B., Demers, A. J., Petersen, K., Spreitzer, M., Theimer, M., and Welch, B. W. Session guarantees for weakly consistent replicated data. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems* (Washington, DC, USA, 1994), PDIS '94, IEEE Computer Society, pp. 140–149.

[23] Terry, D. B., Prabhakaran, V., Kotla, R., Balakrishnan, M., Aguilera, M. K., and Abu-Libdeh, H. Consistency-based service level agreements for cloud storage. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 309–324.