Consistency Oracles: Towards an Interactive and Flexible Consistency Model Specification

Beom Heyn Kim
University of Toronto

Sukwon Oh
University of Toronto

David Lie
University of Toronto

ABSTRACT

Many modern distributed storage systems emphasize availability and partition tolerance over consistency, leading to many systems that provide weak data consistency. However, weak data consistency is difficult for both system designers and users to reason about. Formal specifications offer precise descriptions of consistency behavior, but they require expertise and specialized tools to apply to real software systems. In this paper, we propose and describe *consistency oracles*, an alternative way of specifying the consistency model of a system that provides interactive answers, making them easier and more flexible to use in a variety of ways. A consistency oracle mimics the interface of a distributed storage system, but returns all possible values that may be returned under a given consistency model. This allows consistency oracles to be directly applied in the testing and verification of both distributed storage systems and the client software that uses those systems.

CCS CONCEPTS

• **Software and its engineering → Consistency**; • **General and reference →** *Verification*;

KEYWORDS

Consistency Oracles, Consistency Models, Formal Specification, Verification, Distributed Systems

ACM Reference format:
Beom Heyn Kim, Sukwon Oh, and David Lie. 2017. Consistency Oracles: Towards an Interactive and Flexible Consistency Model Specification. In *Proceedings of HotOS '17, Whistler, BC, Canada, May 08-10, 2017,* 6 pages.
https://doi.org/10.1145/3102980.3102994

1 INTRODUCTION

As data storage systems become more and more distributed, they have increasingly moved towards various weak data consistency models. As demonstrated by Amazon's well-known shopping cart example [26], many of these systems, such as Cassandra, Amazon S3 and DynamoDB emphasize availability and partition tolerance over consistency, and thus motivated a departure from strong to weak consistency. However, weak consistency models, such as eventual consistency, are not well specified, leading developers of both storage systems and applications to disagree over what the correct consistency behavior should be. For example, developers of Cassandra had a long, complicated discussion on a correct behavior for Casandra Bug-2494 [6] as described in Section 2.2.

To properly take advantage of weak consistency, developers using such systems should be able to easily get the answer to two important questions. First, given the current state of the system and a read of a value, what possible values could be returned? Second, if my application uses a weakly consistent system, how should I handle the stale or inconsistent values it might return? To answer these questions, we need to know the precise and correct behavior of various consistency models.

Formal specifications are one well-known way of having a precise description of what the behavior of a system should be. However, formal specifications are often only useful when used in conjunction with formal verification tools, which require specialized knowledge that is beyond the domain of most software developers. Even with such tools, the application of a formal specification to check the correctness of a system often involves manual tuning and a considerable amount of human effort [17]. As a result, even if employed, their use is often limited to a specialized subset of the development team in a large software project. Furthermore, as formal specifications need to be used with formal verification tools and typically used to verify the system implementation, they usually offer little benefit to developers of client applications who use the verified service.

We propose a more practical alternative to formal specifications, called a *consistency oracle*, which addresses these two issues. Currently, formal specifications exist as descriptions of a system, usually defining an abstract state machine, invariants and safety properties of a system. Rather than describing system's behavior, consistency oracles provide answers to queries posed to them about how the system should behave. Specifically, they answer the question "given all data and meta-data of a system, what are all the valid values a system could return to a given query". We argue that this format is both easier and more flexible (in that they can be used in more ways) than current formal specifications. It can be used to directly answer the first question posed earlier, and combined with testing, can answer the second by allowing a developer to see all the implications of the values a weakly consistent system could return.

Our prototype consistency oracle takes the history of the current and all previous operations capturing all data and meta-data of the system and the description of the consistency model that a service should provide. The consistency oracle then produces an answer that is the set of all values that could be read as a result of the current operation. Since the oracle takes essentially the same interface as the service, users may integrate the consistency oracle with their testing framework to verify their system's behaviors. Moreover,

HotOS '17, May 08-10, 2017, Whistler, BC, Canada

ACM ISBN 978-1-4503-5068-6/17/05…$15.00
https://doi.org/10.1145/3102980.3102994

users do not need specialized knowledge to use the consistency oracle but just need to properly invoke API calls similar to those of any data storage system.

We envision two usage scenarios of consistency oracles.

(1) To check storage system servers: Driving servers using advanced testing tools such as concrete model checker, fuzzer or concolic execution engine [7], we check if servers would violate the consistency model by comparing values from real storage system servers with answers from consistency oracles.
(2) To check client applications: While driving applications using the aforementioned tools, we can redirect each I/O request to consistency oracles instead of storage system servers. Then, for each write, consistency oracles will produce an answer that is a set of all valid values to read. From the answer, a value can be uniformly picked and returned to applications. Finally, we can check whether values returned to applications end up breaking any application-specific invariant or not.

Furthermore, consistency oracles can contribute towards the standardization of consistency models. Standardization of generic consistency models like "eventual consistency" will increase the portability of the client applications between services, whereas clients today are often highly tied to the service they interact with, not only because of the service's API, but also because of the subtle differences in consistency behaviors across services. Finally, having consistency oracles also can help to establish unambiguous service level agreement (SLA) between service providers and customers. By exactly specifying the set of values a service may return in any circumstance via ⟨CM⟩ described in Section 3.1, the SLA can simply and precisely specify what consistency guarantees the service will provide.

Section 2 will elaborate on the motivations for consistency oracles. Then, we discuss the design of our prototype consistency oracle in more detail in Section 3. Section 4 will describe what we are currently working on and planning to do in the future. Finally, we talk about related work in Section 5 and conclude in Section 6.

2 MOTIVATION

Current consistency model specifications have shortcomings that consistency oracles can address. We outline them below.

2.1 Lack of specifications

Current distributed storage systems do not precisely specify the consistency behavior in a way that is easy for users to understand all the possible values a system may return. This underspecification can arise in many ways. We give examples of several that we have observed in our experience. First, many distributed systems are highly configurable in the way they handle reads, writes and replication. For example, the MongoDB documentation [14] states:

> *You can configure each write to return after success on the primary, on multiple set members, a majority of set members, or all members. Reads can be applied to the primary member, to secondary members if the primary is unavailable, to specific members exclusively (for workload isolation), or to the nearest secondary based on ping distance.*

From this we can see there are 4 ways to configure writes and 4 ways to configure reads resulting in 16 combinations! However, the specification just gives the options without specifying how the combinations of read and write configurations may interact with each other. Developers must fill in the missing information themselves. Unfortunately, it is often the case that different developers may come to different conclusions when filling in such information.

Second, descriptions like the MongoDB one above describe how the system is implemented, but not what the behavior that results from that implementation will be. For example, the MongoDB configuration options do not describe any behavior such as how stale the values returned can be or whether values can be observed in an order different from how they are written. This information is underspecified and could likely only be gleaned by the user if they had detailed knowledge of both the code implementation and the system deployment.

The tendency to have many configuration options and describe the implementation of the system in the specification means that the actual behavior of the system is up to the reader of the specification to interpret. This is because the specifications are neither precise nor complete. Consistency oracles are by their nature both precise and complete. For any query, they will return the set of possible values the system could return.

2.2 Confusion about behavior

Imprecise specification and the lack of a reference model mean that users and even system developers often cannot agree on what the expected system behavior should be. A good example is given by Casandra Bug-2494 [6]. This bug concerns the possibility that a client may see non-monotonically increasing writes when Cassandra reads are configured to use quorums. The bug report has a total of 17 messages plus a reference to an e-mail thread with another 10 messages. The bug messages start off with a statement from one developer stating that the requested level of consistency part of the specified behavior of Cassandra:

> *As far as I can tell the consistency being asked for was never promised by Cassandra is in fact not expected.*

Only to be refuted by another developer who believes it is in the behavioral specification:

> *I think the guarantee of quorum reads not seeing old writes once a quorum read sees a new write is very useful. I suspect most people already think that this guarantee occurs, including, it seems, Jonathan Ellis*[1] *...*

Finally, the last comment, which appears 4 years after the patch to fix the bug was applied, asks the question:

> *The relevant code in the patch has changed significantly. Is the monotonic read consistency guarantee still provided?*

This simple example illustrates the problems that arise when people have to mentally translate complex specification into a mental model of system behavior – it is rare for all people to do it in the same way once the specification gets suitably complex. Instead

[1] Jonathan Ellis was the chairman of the Cassandra Project at the time

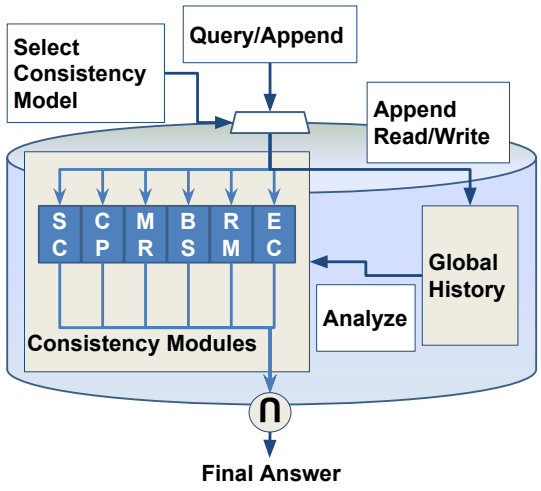

Figure 1: Consistency Oracle Architecture

of specifying behavior, consistency oracles demonstrate behavior and thus cannot lead to differences in interpretation.

2.3 Lack of standards

Consistency specifications for deployed systems have subtle and complex differences between services. For example, Amazon's S3 service generally provides eventual consistency, but in some cases, offers slightly more esoteric behavior [1]:

> *Amazon S3 provides read-after-write consistency for PUTS of new objects in your S3 bucket in all regions with one caveat. The caveat is that if you make a HEAD or GET request to the key name (to find if the object exists) before creating the object, Amazon S3 provides eventual consistency for read-after-write.*

Such differences make migration from one service to another service non-trivial. Even if application developers replace all API calls from the old service with those of the new one, differences in consistency behavior may lead to subtle and unexpected misbehavior. As a result, changing underlying storage systems is always a risky and difficult engineering task.

As the number of cloud services and systems increases, the need to be able to easily migrate from one system to another will become increasingly important as users seek to avoid vendor lock-in and remove bottlenecks as their application usage increases. Consistency oracles can act as a standard to which both services and client applications can be engineered to meet. Clients and services that have been successfully tested against the same consistency oracle will be more likely to be compatible, even if they have never been used together before.

3 CONSISTENCY ORACLE

3.1 Overview

A consistency oracle is a software artifact with an interface similar to a generic distributed storage system. Clients interact with the oracle by submitting read and write operations. However, rather than trying to store data quickly, efficiently and reliably, the consistency oracle's only goal is to compute all possible responses to any operation performed by the client. As a result, the architecture of a consistency oracle is very different from that of a distributed storage system.

One major difference is that while a consistency oracle typically describes consistency behavior for a distributed system, it itself does not have to be distributed and exists as a monolithic, single-threaded application. Figure 1 shows architecture overview. Our current consistency oracle takes a basic read and write operation[2]. To interact with the oracle, clients make requests to the oracle like they would do to a distributed storage system. More formally, our consistency oracle takes each operation as an ⟨INPUT⟩ tuple:

$$\langle \texttt{CID} \rangle \ \langle \texttt{KEY} \rangle \ \langle \texttt{OP} \rangle \ \langle \texttt{VAL} \rangle \ \langle \texttt{TS} \rangle \ \langle \texttt{USR} \rangle$$

where ⟨CID⟩ means client ID, ⟨KEY⟩ is data object ID, ⟨OP⟩ is the operation type (either read or write), ⟨VAL⟩ means hash of the data value, ⟨TS⟩ means timestamp, ⟨USR⟩ is user provided tag for extensibility. Specifically, a consistency oracle provides two types of interface calls. One is Append(⟨INPUT⟩, ⟨CM⟩) and the other is Query(⟨INPUT⟩, ⟨CM⟩). The parameter ⟨CM⟩ defines the consistency model the operation is made under. For Append, ⟨CM⟩ is also passed in to update session information appropriately. Then, the user can call Query with the ⟨CM⟩ to select the consistency model. The Query also takes ⟨INPUT⟩ as we need to provide a read operation for which the consistency oracle returns the answer given the global history.

We assume the existence of a global history ⟨GHIST⟩, which is a totally ordered sequence of ⟨INPUT⟩s. ⟨GHIST⟩ is initially an empty string and on each request, an ⟨INPUT⟩ is appended to the global history: ⟨GHIST⟩·⟨INPUT⟩. Because the global history contains every operation of every client, any consistency model can be supported as long as the consistency model can be expressed as restrictions on what values are allowed to be read within this global history. We currently cannot express concurrent overlapping operations in ⟨GHIST⟩, and discuss how we may extend consistency oracles to support these in Section 4.2.

Our prototype supports various consistency guarantees: Strong Consistency, Consistent Prefix, Monotonic Reads, Bounded Staleness, Read-My-Write, and Eventual Consistency, which we will refer to as SC, CP, MR, BS, RM, or EC, respectively [25]. For instance, we can specify a consistency model supporting consistent prefix, monotonic reads and bounded staleness by the set ⟨CM⟩ = [CP, MR, BS].

Upon receiving a Query, the oracle returns the list of all valid values from the global history that a system with the specified consistency model could return. The oracle models each consistency guarantee using the module specific to that guarantee. When receiving a Query request, the consistency oracle will have each of the consistency modules read the global history and return the set of values that can be returned given the restrictions in each respective module. Then, the final answer is produced by computing the intersection of the selected consistency modules. Hence, the values contained in all of sets produced by the selected modules are the ones satisfying all selected consistency guarantees and therefore satisfying the consistency model composed of those guarantees.

[2]We can emulate most other common storage operations such as CRUD (create, read, update, delete), list, batch operation, lock, etc. using read and write operations.

As an example, suppose we perform the sequence of operations below at the specified times, and model a system with a consistency model that includes constants prefix, monotonic reads and bounded staleness with 5 as the staleness bound (i.e. ⟨CM⟩ = [CP, MR, BS]):

```
At t=0: write(X, 1)
At t=2: write(Y, 1)
At t=6: write(X, 2)
At t=7: read(): {X=1,Y=1}
At t=8: write(X, 3)
At t=9: write(Y, 2)
At t=12: read(): {X=?,Y=?}
```

where X and Y are keys for integer values initialized to 0. We query the consistency oracle for the set of valid return values for the read at t=12. The consistency oracle computes the partial set of values that can be returned for each consistency guarantee individually: [{0,0}, {1,0}, {1,1}, {2,1}, {3,1}, {3,2}] for CP, [{1,1}, {2,1}, {3,1}, {3,2}] for MR, [{2,1}, {2,2}, {3,1}, {3,2}] for BS with a 5-unit time bound, where each tuple represents the possible values for {X,Y}. Computing the intersection gives [{2,1}, {3,1}, {3,2}]. Consistency oracles can be easily extended to support new consistency models by implementing and adding a module that models the new consistency model.

3.2 Computing the ordering of operations

The consistency oracle is expected to be used in the testing environment where every client and server can be running on a single machine. Thus, the single clock enables a total order of all operations to be defined in the global history. This total order is the same total order that would exist in a distributed system if all servers had perfectly synchronized clocks and timestamped every operation they performed in the same way.

Given this total order, the consistency oracle must compute all possible orders a distributed system with a certain consistency model may interpret the operations to have occurred. A simple case is a system that enforces strong consistency, where the only order of operations that the system can interpret the operations as having occurred in is the same as the total order specified in the global history. A more complex case is that of eventual consistency, where the operations in the total order can be interpreted by the consistency oracle to have happened in any arbitrary order. In that case, for a certain query in the global history, the consistency oracle must return results for all permutations of operations previous to the current operation in the global history. Finally, for other consistency models that define limited partial orders, such as monotonic reads or read-my-writes, the consistency oracle must determine which operations may be interpreted as being unordered by the consistency model (i.e. there is no happens-before relationship defined for example), and apply conflict resolution as defined by the consistency model.

3.3 Discussion

Because consistency oracles are assumed to be ground truth in the same way as formal specifications. As a result, it is important that they are correct. Since they are very simple to implement, rigorous unit testing may be enough to have high confidence about the correctness of consistency oracles themselves. Yet, more cautious users can construct consistency oracles using formal methods and the same techniques used to produce provably correct systems [12, 13, 17]. Although this may require as much effort as any other formal specification and verification, consistency oracles can be constructed once by formal verification experts and used many times by a number of other average developers who do not have specialized knowledge and skills on formal verification. Moreover, because they are simpler than real systems, they are more amenable to formal methods.

Table 1: Size of Prototype Implementation (in LOC)

Component	Count
Core (⟨GHIST⟩, Query/Append handling)	109
Data Structure (⟨CM⟩, ⟨INPUT⟩)	43
Bounded Staleness Module	22
Eventual Consistency Module	19
Monotonic Reads Module	26
Read-My-Writes Module	41
Strong Consistency Module	21
Total	281

Distributed systems are required to have a mechanism to deal with various types of failures such as crash or network partition. Failures cause nodes to have different views on the history of previous operations, because some nodes may not see the operations performed on other nodes, and vice versa. Hence, a total order may not exist initially after a failure. However, distributed systems usually provide conflict resolution mechanisms to re-establish a consistent view of the system across components. While consistency oracles currently do not model failures, and assume that systems seek to make the values they return agnostic to such failures, this is not always the case and we may wish to have consistency oracles include failures as yet another type of ⟨INPUT⟩ to the Append operation. In this case, conflict resolution logic would need to be added to each module so that consistency oracles can also simulate what values systems may return after a failure has occurred.

4 CURRENT AND FUTURE WORK

4.1 Current Work

Our current consistency oracle prototype has been constructed manually. We believe consistency oracles can be generated automatically from formal specifications, and leave this for future work. Because consistency oracles simulate a distributed system but are not distributed themselves, and don't need to be resource efficient or fast, they can be fairly simple. Table 1 gives the line counts for our consistency oracle, which is implemented in Java.

We are currently applying our consistency oracle as an invariant generator for a concrete model checker. We intend to use this model checker to detect consistency bugs in distributed storage systems such as ZooKeeper, HBase and Cassandra. To combine the oracle with the concrete model checker, we interpose on API calls to the system and record the history of those operations. We then use the model checker to explore various states the data storage system can enter, by calling different system APIs and interleaving asynchronous events. After each sequence of events, a series of API

calls is made that attempts to read each valid value in the storage system. At the same time, the history of API operations is passed to the consistency oracle. Then, the value returned by the data storage system is checked for membership in the set of values returned by the consistency oracle.

We hope that consistency oracles can prevent the confusion demonstrated in Cassandra Bug-2494 by providing a reference example of how a system that provides Monotonic Reads should behave. By specifying `MR` as a consistency guarantee and running the sequence of operations that demonstrates the issue, they can compare the behavior of the system with that of the oracle. Using a model checker would explore all possible interleavings of internal events and detect if it is possible for the system to violate the consistency guarantee.

4.2 Future Work

Currently, our consistency oracles only handle read and write operations. In the future, we plan to expand its interface and the consistency modules to also handle more complex semantics such as locks and transactions that many real data storage systems support.

We also plan to explore verifying whether client applications can correctly handle inconsistent data that can be returned by weakly consistent systems. To do this, we assume the existence of a test harness that would be used to test the client application against the data storage service. To use the consistency oracle, we replace the data storage system with the consistency oracle and include an adaptation layer that translates API calls to the data storage system into the `Append` and `Query` operations of the consistency oracle. For read and write operations, we can append the operation in the same order that they are invoked. For read operations, the consistency oracle will pick one of the possible values and return it to the client operation. Repeated testing will enable the client to be tested against all possible return values, guaranteeing that the client will have been tested against obscure or uncommon values, as well as common ones.

Using consistency oracles to test clients is agnostic to the way inputs are generated for the client application. As long as the inputs cover a variety of cases, it does not matter whether the client inputs are generated using a fuzzer, concolic execution engine or just manually specified test scripts.

The consistency oracle prototype currently requires the total ordering to be determined by the oracle users using the synchronized clock. In cases where the storage system does not itself determine a total order, it may not be possible to define such a total order for the consistency oracle's global history. We plan to address this by extending the global history to be a database of operations, recorded by the start and end time of each operation, thus permitting partially ordered concurrent operations. Each consistency module would then need to be altered to take this into account.

5 RELATED WORK

Several previous works devised consistency checking algorithms[2, 4, 5]. Also, there were a few works looking at the consistency from the security perspective where untrusted storage provider may get compromised and violate consistency guarantees[16, 20, 22]. Yet, these systems do not deal with various composable consistency models.

There have been several consistency benchmarking techniques developed by researchers[3, 8, 9, 23, 27]. However, these are just trying to measure how fast the update on data gets converged across replicas.

There were automation techniques for finding bugs in distributed systems implementation. Concrete model checkers were studied for detecting concurrency bugs in distributed systems[10, 11, 15, 18, 21, 24, 28, 29]. None of them checks for the violation of relaxed consistency model.

Also, there was previous work on providing automation to select the appropriate consistency level[19]. However, the work does not support as many consistency models.

6 CONCLUSION

Weak consistency models have gained popularity over the last decade, as more and more services have been moving to the cloud. However, many systems and services do not precisely specify the consistency model that is supported, leaving the consistency behavior open to interpretation. To overcome the limitations of current state-of-the-art, we propose *consistency oracle* as a pragmatic instantiation of a formal specification. We plan to develop techniques using the oracle to help using various consistency models safer and, therefore, can encourage systems and services to support a larger range of consistency guarantees from which client application may benefit.

ACKNOWLEDGEMENTS

We would like to thank Eyal de Lara, Michael Stumm, Michelle Wong, Zhen Huang and the participants of the HotOS workshop for their helpful comments on this work. The research in this work is supported by a Tier 2 Canada Research Chair and an NSERC Discovery Grant. Beom Heyn Kim and Sukwon Oh are both supported by Bell Graduate Scholarships.

REFERENCES

[1] Amazon. 2017. Amazon S3 Data Consistency Model. (2017). http://docs.aws.amazon.com/AmazonS3/latest/dev/Introduction.html#ConsistencyModel.

[2] Eric Anderson, Xiaozhou Li, Mehul A. Shah, Joseph Tucek, and Jay J. Wylie. 2010. What Consistency Does Your Key-value Store Actually Provide?. In *Proceedings of the Sixth International Conference on Hot Topics in System Dependability (HotDep'10)*. USENIX Association, Berkeley, CA, USA, 1–16. http://dl.acm.org/citation.cfm?id=1924908.1924919

[3] Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica. 2012. Probabilistically Bounded Staleness for Practical Partial Quorums. *The VLDB Endowment* 5, 8 (April 2012), 776–787.

[4] David Bermbach, Sherif Sakr, and Liang Zhao. 2013. Towards Comprehensive Measurement of Consistency Guarantees for Cloud-Hosted Data Storage Services. In *The 5th TPC Technology Conference on Performance Evaluation & Benchmarking (TPCTC 2013)*. Springer, Cham, Switzerland, 32–47.

[5] David Bermbach and Stefan Tai. 2011. Eventual Consistency: How Soon is Eventual? An Evaluation of Amazon S3's Consistency Behavior. In *Proceedings of the 6th Workshop on Middleware for Service Oriented Computing (MW4SOC '11)*. ACM, New York, NY, USA, Article 1, 6 pages. https://doi.org/10.1145/2093185.2093186

[6] Sean Bridges. 2011. Quorum reads are not monotonically consistent. (2011). https://issues.apache.org/jira/browse/CASSANDRA-2494.

[7] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, Berkeley, CA, USA, 209–224. http://dl.acm.org/citation.cfm?id=1855741.1855756

[8] Wojciech Golab, Xiaozhou Li, and Mehul A. Shah. 2011. Analyzing Consistency Properties for Fun and Profit. In *Proceedings of the 30th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC '11)*. ACM, New York, NY, USA, 197–206. https://doi.org/10.1145/1993806.1993834

[9] Wojciech Golab, Muntasir Raihan Rahman, Alvin Au Young, Kimberly Keeton, Jay J. Wylie, and Indranil Gupta. 2013. Client-centric Benchmarking of Eventual Consistency for Cloud Storage Systems. In *Proceedings of the 4th Annual Symposium on Cloud Computing (SOCC '13)*. ACM, New York, NY, USA, Article 28, 2 pages. https://doi.org/10.1145/2523616.2525935

[10] Rachid Guerraoui and Maysam Yabandeh. 2011. Model Checking a Networked System Without the Network. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI'11)*. USENIX Association, Berkeley, CA, USA, 225–238. http://dl.acm.org/citation.cfm?id=1972457.1972481

[11] Huayang Guo, Ming Wu, Lidong Zhou, Gang Hu, Junfeng Yang, and Lintao Zhang. 2011. Practical Software Model Checking via Dynamic Interface Reduction. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*. ACM, New York, NY, USA, 265–278. https://doi.org/10.1145/2043556.2043582

[12] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. 2015. IronFleet: Proving Practical Distributed Systems Correct. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. ACM, New York, NY, USA, 1–17. https://doi.org/10.1145/2815400.2815428

[13] Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. 2014. Ironclad Apps: End-to-end Security via Automated Full-system Verification. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, Berkeley, CA, USA, 165–181. http://dl.acm.org/citation.cfm?id=2685048.2685062

[14] MongoDB Inc. 2017. MongoDB Documentation - FAQ. (2017). https://www.mongodb.com/faq.

[15] Charles Killian, James W. Anderson, Ranjit Jhala, and Amin Vahdat. 2007. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. In *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation (NSDI'07)*. USENIX Association, Berkeley, CA, USA, 18–18. http://dl.acm.org/citation.cfm?id=1973430.1973448

[16] Beom Heyn Kim and David Lie. 2015. Caelus: Verifying the Consistency of Cloud Services with Battery-Powered Devices. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP '15)*. IEEE Computer Society, Washington, DC, USA, 880–896. https://doi.org/10.1109/SP.2015.59

[17] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP '09)*. ACM, New York, NY, USA, 207–220. https://doi.org/10.1145/1629575.1629596

[18] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F. Lukman, and Haryadi S. Gunawi. 2014. SAMC: Semantic-aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, Berkeley, CA, USA, 399–414. http://dl.acm.org/citation.cfm?id=2685048.2685080

[19] Cheng Li, João Leitão, Allen Clement, Nuno Preguiça, Rodrigo Rodrigues, and Viktor Vafeiadis. 2014. Automating the Choice of Consistency Levels in Replicated Systems. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC'14)*. USENIX Association, Berkeley, CA, USA, 281–292. http://dl.acm.org/citation.cfm?id=2643634.2643664

[20] Jinyuan Li, Maxwell Krohn, David Mazières, and Dennis Shasha. 2004. Secure Untrusted Data Repository (SUNDR). In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6 (OSDI'04)*. USENIX Association, Berkeley, CA, USA, 9–9. http://dl.acm.org/citation.cfm?id=1251254.1251263

[21] Madanlal Musuvathi, David YW Park, Andy Chou, Dawson R Engler, and David L Dill. 2002. CMC: A pragmatic approach to model checking real code. *ACM SIGOPS Operating Systems Review* 36, SI (2002), 75–88.

[22] Raluca Ada Popa, Jacob R. Lorch, David Molnar, Helen J. Wang, and Li Zhuang. 2011. Enabling Security in Cloud Storage SLAs with CloudProof. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference (USENIXATC'11)*. USENIX Association, Berkeley, CA, USA, 31–31. http://dl.acm.org/citation.cfm?id=2002181.2002212

[23] Muntasir Raihan Rahman, Wojciech Golab, Alvin AuYoung, Kimberly Keeton, and Jay J. Wylie. 2012. Toward a Principled Framework for Benchmarking Consistency. In *Proceedings of the Eighth USENIX Conference on Hot Topics in System Dependability (HotDep'12)*. USENIX Association, Berkeley, CA, USA, 8–8. http://dl.acm.org/citation.cfm?id=2387858.2387866

[24] Jiri Simsa, Randy Bryant, and Garth Gibson. 2010. dBug: Systematic Evaluation of Distributed Systems. In *Proceedings of the 5th International Conference on Systems Software Verification (SSV'10)*. USENIX Association, Berkeley, CA, USA, 3–3. http://dl.acm.org/citation.cfm?id=1929004.1929007

[25] Doug Terry. 2013. Replicated Data Consistency Explained Through Baseball. *Commun. ACM* 56, 12 (Dec. 2013), 82–89. https://doi.org/10.1145/2500500

[26] Werner Vogels. 2008. Eventually Consistent - Revisited. (2008). http://www.allthingsdistributed.com/2008/12/eventually_consistent.html.

[27] Hiroshi Wada, Alan Fekete, Liang Zhao, Kevin Lee, and Anna Liu. 2011. Data Consistency Properties and the Tradeoffs in Commercial Cloud Storages: the Consumers' Perspective. In *The 5th Biennial Conference on Innovative Data Systems Research (CIDR)*. CIDR Conference, Asilomar, California, USA, 134–143.

[28] Maysam Yabandeh, Nikola Knezevic, Dejan Kostic, and Viktor Kuncak. 2009. CrystalBall: Predicting and Preventing Inconsistencies in Deployed Distributed Systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI'09)*. USENIX Association, Berkeley, CA, USA, 229–244. http://dl.acm.org/citation.cfm?id=1558977.1558993

[29] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. 2009. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI'09)*. USENIX Association, Berkeley, CA, USA, 213–228. http://dl.acm.org/citation.cfm?id=1558977.1558992