Towards Property-Based Consistency Verification

Paolo Viotti
EURECOM
viotti@eurecom.fr

Christopher Meiklejohn
Université catholique de Louvain
christopher.meiklejohn@gmail.com

Marko Vukolić
IBM Research - Zurich
mvu@zurich.ibm.com

CCS Concepts

•**Software and its engineering** → **Consistency;** *Empirical software validation;*

Keywords

Storage systems; Consistency; Verification.

ABSTRACT

We propose a novel approach to the verification of consistency models implemented in distributed storage systems. We base our work on a *declarative* semantic model defining consistency conditions as predicates expressing ordering and visibility of operations. This model allows for a testing methodology focused on correctness properties rather than operational semantics. Finally, we present and discuss the design and preliminary implementation of a practical *property-based* consistency verification framework.

1. INTRODUCTION

Consistency is the key correctness criterion of distributed storage systems. In spite of recent efforts proposing *consistency-by-construction* through formal methods [29, 20], most real world storage systems are still developed in an ad-hoc manner. Specifically, most of the time, practitioners start with an implementation and proceed with verification through limited testing afterwards (e.g., using unit and integration tests). In an attempt to bridge the gap between traditional testing and formal techniques proposed by academia, several approaches have been devised to provide a general way of verifying implementations of consistency models.

Strong consistency checkers.

Several works focused on devising efficient techniques to determine whether executions of storage systems are strongly consistent [23, 30, 13, 17]. This basic decision problem has then been extended to support other comparably strong semantics [7] and on-the-fly, incremental verification [14].

PaPoC'16, April 18-21, 2016, London, United Kingdom
© 2016 ACM. ISBN 978-1-4503-4296-4/16/04...$15.00
DOI: http://dx.doi.org/10.1145/2911151.2911162

Read-write staleness benchmarking.

A recent approach proposes client-side staleness measurements as a method to assess consistency. In this approach, data store clients perform write and read operations in a coordinated fashion in order to detect anomalies related to data staleness. This technique has been applied both to open source NoSQL databases [24] and, in a black-box testing manner, to commercial geo-replicated cloud stores [8, 28, 21], typically in the context of eventually consistent systems.

Precedence graph approach.

A number of works related to transactional systems adopted a graph-based approach in which inconsistencies are identified as cycles in a precedence (or serialization) graph [4, 32].

All these approaches, however, lack generality, as they target only a limited subset of consistency models. They also lack a comprehensive, structured view of the entire system, which makes them suboptimal in verifying the core semantics, or composition thereof, of different consistency models.

We believe that the first step towards building an effective and comprehensive consistency testing framework should be the adoption of a theoretically sound model of consistency. To this end, we advocate the use of a *declarative* approach to define a set of core semantics applicable to all consistency models. In particular, we aim at expressing both client-side visibility of read/write operations and global state configurations. By using logic predicates that encompass these two perspectives, we define consistency semantics that capture, in form of graph entities, the salient aspects of system executions, i.e. *ordering* and *visibility* of events. In this way, verifying an implementation of a given consistency semantics amounts to finding, for a given execution, the global state configurations that validate a logic predicate, taking into account client-side events.

In summary, we propose a declarative, *property-based* approach to consistency verification, in the vein of proposals made in the context of generic software testing [11]. We experiment this approach by implementing Conver, an early prototype of a practical property-based consistency verification framework developed in Erlang.

2. A DECLARATIVE FOUNDATION

Several works in literature (e.g., [22, 5]) illustrated the benefits of a declarative approach in the context of database, networking and distributed programming. Essentially, the declarative, axiomatic approach offers a better match to application-level semantics than the traditional imperative, operational approach. Therefore, the declarative approach allows for a more expressive, clean and compact way to describe the logic of distributed applications, fostering a structured programming model. Additionally, declarative approach is amenable to static checking of correctness conditions, allowing distributed systems problems to be naturally cast into SAT/SMT problems [6, 26, 15], which in turn allows to leverage the efficiency and maturity of the related state-of-the-art tools (e.g., [31, 1]).

We found a model supporting the declarative paradigm in the work by Burckhardt [9], later extended and refined in the context of both transactional [10] and non-transactional semantics [27]. This model allows the definition of declarative, composable consistency semantics, which can be expressed in terms of first-order logic predicates over graph entities which, in turn, describe the visibility and ordering of events. Table 1 overviews the most relevant entities of this model.

Entity	Description
Operation (op)	Single operation. Includes: process and object id, type, input and output values, start and end time.
History (H)	The set of operations of an execution. Described by: returns-before partial order, same-session and same-object equivalence relations.
Visibility (vis)	Acyclic partial order on operations. Accounts for propagation of write operations.
Arbitration (ar)	Total order on operations. Specifies how the system resolves conflicts.

Table 1: Summary of most relevant entities of the model described in [27].

As an example, a consistency semantics that requires respecting real time ordering would include the following predicate:

$$\textsc{RealTime} \triangleq rb \subseteq ar \tag{1}$$

In other words, the predicate requires arbitration (ar) to comply with the returns-before partial ordering (rb).[1]

3. PROPERTY-BASED CONSISTENCY VERIFICATION

Property-based testing (PBT) [11] is an approach to generic software testing that alleviates the burden of test case generation from the user, allowing the user to focus on specifying application-level properties that should hold for all executions. A PBT tool, when supplied with these properties along with information about the generic format of a valid input, generates random inputs, and then applies these inputs to the program while constantly verifying the validity of the supplied properties throughout the execution. In a sense, PBT combines the two old ideas of specification-based testing [19] and random testing [16]. Additionally, to make up for the possible "noise" induced by the random test case generation, modern PBT tools automatically reduce the complexity of failing tests to a minimal test case [33], thus serving as powerful diagnostic tools. As a basic example of PBT, given a function `lsort` that sorts a list of integers, writing a property that states that the function should not change the list length, would just require the following lines of Erlang code:

```
prop_same_length() ->
  ?FORALL(L,list(integer()),
          length(L)=:=length(lsort(L))).
```

In principle, the PBT approach of expressing and testing consistency as a set of predicates allows for a testing methodology focused on correctness properties rather than operational semantics. We embed this principle in the design of Conver, a prototype of a practical consistency verification framework that we developed in Erlang.[2] Conver generates test cases consisting in executions of concurrent operations invoked on the data store under test. After each execution, Conver collects the details of all the operations and builds the graph entities describing the client-side outcomes (e.g., returns-before relation rb, session-order relation so, etc.). Given client-side outcomes, Conver then builds graph and relations entities about global ordering and visibility of events. Essentially, Conver verifies the compliance to a given consistency model by building and validating the required entities for all logic terms composing the entire consistency predicate. As an example, verifying the RealTime term expressed in (1), requires building the total order ar as a *linear extension* of the partial order rb. As a result of the verification process, Conver outputs a visualization of each failing test case, i.e. all the executions that did not comply with a given consistency semantics. Additionally, the visualization highlights the anomaly that caused the test failure.

Using our preliminary implementation of Conver, we were able to run test executions against Riak[3] and ZooKeeper[4] to verify some of the most common consistency semantics (e.g., monotonic reads, monotonic writes, read-your-writes, writes-follow-reads, causal and strong consistency).

In the following, we briefly discuss the design of several potential extensions of Conver. Figure 1 provides an overview of the functional architecture of Conver, including these extensions.

Targeted test case generation.

We envision the implementation of heuristics to generate test case executions tailored to verify specific semantics. For instance, depending on the consistency model under verification, Conver could adjust the ratio between read and write operations, or establish a loose coordination among clients to better exercise their concurrency.

Fault injection.

Fault injection is a technique that challenges the implementation of data stores by exercising the intrinsic non-determinism of distributed systems [18]. Thanks to its black-box testing approach, Conver can easily be instrumented to inject external faults, such as network partitions and process failures.

[1]A complete description of each entity involved along with examples of other real world consistency semantics expressed using this model are available in [27].

[2]The source code of Conver is available at https://github.com/pviotti/conver.

[3]http://basho.com/products/riak-kv/

[4]https://zookeeper.apache.org/

Test case shrinking.

Similarly to what implemented in most PBT tools, once identified the anomaly that caused the failure of a test case, Conver could try to reproduce it by enacting a less complex execution (i.e. an execution involving less processes and operations). When using Conver as a tool for test-driven development, this feature would substantially ease the debugging tasks.

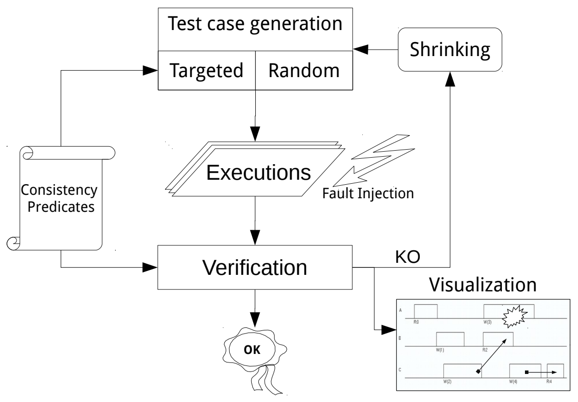

Figure 1: Functional architecture of the Conver verification framework.

4. DISCUSSION

Using Conver may require the additional effort of writing property-like specifications for the data store under test. However, we believe that this may yield better comprehension and more rigorous documentation of the software, which would be especially beneficial for commercial systems subject to SLAs. Moreover, to minimize this effort, Conver provides a set of pre-compiled predicates for the most common consistency models. Finally, to address the need of developing bindings for the data stores under test, we designed Conver as a modular framework, following the example of similar testing suites [12]. Thanks to this design, adding the support for a new data store entails implementing a simple interface, which usually amounts to writing less than 30 lines of Erlang code.

We further note that the property-based testing approach has already been applied by practitioners to verify the correctness of distributed applications. Specifically, modern PBT tools model the state of the system as a set of variables that are verified through postconditions [3]. This state model can only support the verification of consistency models that presume single-copy semantic, i.e. *strong* consistency models. Even though our work respects and builds upon the principles of property-based testing, it differs substantially from common PBT tools in the way the state of the system is represented and verified. In particular, the semantic model implemented by Conver describes the system state as a graph of prior operations [9]. Hence, Conver can verify a broader set of consistency models that apply to generic replicated storage systems.

5. FUTURE WORK

In the following, we briefly discuss several possible extensions of our work.

Client distribution and time tracking.

Because of the way it generates monotonic values and keeps track of time within the Erlang runtime environment [2], Conver requires hosting all data store clients on a single machine. While this design constraint does not represent a substantial limit in terms of scalability, it does limit the scope of Conver, as it prevents an exhaustive testing of geo-replicated cloud storage systems. Implementing the support for distributed clients entails dealing with the well-known problem of reliable time tracking in distributed systems [25]. Thus, this would require devising a solution that leverages the techniques that have been proposed in literature to address this problem.

Dynamic consistency verification.

Although Conver was conceived for static, offline verification of executions, its design does not prevent the implementation of a module for incremental, on-the-fly verification. We believe that the amount of computation required to enable on-the-fly detection of consistency anomalies would be sustainable so long as the number of processes involved remains moderate.

Transactional consistency models.

Conver can be extended to support the verification of transactional consistency models. In this regard, the state model supported by Conver has already been adapted to express transactional semantics [10]. Hence, this extension would entail the support of the additional semantic entities for expressing transactional features within the model, and the implementation of a mechanism to detect transactional anomalies [4].

References

[1] Alloy solver. http://alloy.mit.edu/alloy; visited February 2016.

[2] Erlang User's Guide. Time and time correction in Erlang. http://erlang.org/doc/apps/erts/time_correction.html; visited February 2016.

[3] PropEr testing of generic servers. http://proper.softlab.ntua.gr/Tutorials/PropEr_testing_of_generic_servers.html; visited February 2016.

[4] A. Adya. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions.* Ph.D., MIT, Cambridge, MA, USA, Mar. 1999. Also as Technical Report MIT/LCS/TR-786.

[5] P. Alvaro, T. Condie, N. Conway, K. Elmeleegy, J. M. Hellerstein, and R. Sears. Boom analytics: exploring data-centric, declarative programming for the cloud. *EuroSys*, 2010. http://doi.acm.org/10.1145/1755913.1755937.

[6] P. Alvaro, A. Hutchinson, N. Conway, W. R. Marczak, and J. M. Hellerstein. Bloomunit: declarative testing for distributed programs. *DBTest*, 2012. http://doi.acm.org/10.1145/2304510.2304512.

[7] E. Anderson, X. Li, M. A. Shah, J. Tucek, and J. J. Wylie. What consistency does your key-value store actually provide? *USENIX HotDep*, 2010.

[8] D. Bermbach and S. Tai. Eventual consistency: How soon is eventual? An evaluation of amazon s3's consistency behavior. *ACM Workshop on Middleware for Service Oriented Computing*, 2011. http://doi.acm.org/10.1145/2093185.2093186.

[9] S. Burckhardt. *Principles of Eventual Consistency*, volume 1 of *Foundations and Trends in Programming Languages.* now publishers, October 2014.

[10] A. Cerone, G. Bernardi, and A. Gotsman. A framework for transactional consistency models with atomic visibility. *CONCUR*, 2015. http://dx.doi.org/10.4230/LIPIcs.CONCUR.2015.58.

[11] K. Claessen and J. Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. *ACM SIGPLAN ICFP*, 2000. http://doi.acm.org/10.1145/351240.351266.

[12] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. *ACM SoCC*, 2010. http://doi.acm.org/10.1145/1807128.1807152.

[13] P. B. Gibbons and E. Korach. Testing shared memories. *SIAM J. Comput.*, 26(4), 1997. http://dx.doi.org/10.1137/S0097539794279614.

[14] W. M. Golab, X. Li, and M. A. Shah. Analyzing consistency properties for fun and profit. *ACM PODC*, 2011.

[15] A. Gotsman, H. Yang, C. Ferreira, M. Najafzadeh, and M. Shapiro. 'Cause i'm strong enough: reasoning about consistency choices in distributed systems. *ACM SIGPLAN-SIGACT POPL*, 2016. http://doi.acm.org/10.1145/2837614.2837625.

[16] R. Hamlet. Random testing. *Encyclopedia of Software Engineering*, pages 970–978, 1994.

[17] A. Horn and D. Kroening. Faster linearizability checking via p-compositionality. *IFIP International Conference on Formal Techniques for Distributed Objects, Components, and Systems, FORTE*, 2015.

[18] K. Kingsbury. Jepsen - distributed systems safety analysis. http://jepsen.io/; visited February 2016.

[19] G. T. Laycock. *The theory and practice of specification based software testing.* PhD thesis, University of Sheffield, 1993.

[20] M. Lesani, C. J. Bell, and A. Chlipala. Chapar: certified causally consistent distributed key-value stores. *ACM SIGPLAN-SIGACT POPL*, 2016. http://doi.acm.org/10.1145/2837614.2837622.

[21] A. Li, X. Yang, S. Kandula, and M. Zhang. Cloudcmp: comparing public cloud providers. *ACM SIGCOMM IMC*, 2010. http://doi.acm.org/10.1145/1879141.1879143.

[22] B. T. Loo, T. Condie, M. N. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative networking: language, execution and optimization. *ACM SIGMOD PODS*, 2006. http://doi.acm.org/10.1145/1142473.1142485.

[23] J. Misra. Axioms for memory access in asynchronous hardware systems. *ACM Trans. Program. Lang. Syst.*, 8(1), 1986. http://doi.acm.org/10.1145/5001.5007.

[24] S. Patil, M. Polte, K. Ren, W. Tantisiriroj, L. Xiao, J. López, G. Gibson, A. Fuchs, and B. Rinaldi. YCSB++: benchmarking and performance debugging advanced features in scalable table stores. *ACM SoCC*, 2011. http://doi.acm.org/10.1145/2038916.2038925.

[25] J. Sheehy. There is no now. *ACM Queue*, 13(3):20, 2015. http://doi.acm.org/10.1145/2742694.2745385.

[26] K. C. Sivaramakrishnan, G. Kaki, and S. Jagannathan. Declarative programming over eventually consistent data stores. *ACM SIGPLAN PLDI*, 2015. http://doi.acm.org/10.1145/2737924.2737981.

[27] P. Viotti and M. Vukolić. Consistency in non-transactional distributed storage systems. *ACM Comput. Surv. (to appear). Also available as arXiv pre-print at http://arxiv.org/abs/1512.00168.*

[28] H. Wada, A. Fekete, L. Zhao, K. Lee, and A. Liu. Data consistency properties and the trade-offs in commercial cloud storage: the consumers' perspective. *CIDR*, 2011.

[29] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. E. Anderson. Verdi: a framework for implementing and formally verifying distributed systems. *ACM SIGPLAN PLDI*, 2015. http://doi.acm.org/10.1145/2737924.2737958.

[30] J. M. Wing and C. Gong. Testing and verifying concurrent objects. *J. Parallel Distrib. Comput.*, 17(1-2), 1993. http://dx.doi.org/10.1006/jpdc.1993.1015.

[31] Z3. High-performance theorem prover. https://github.com/Z3Prover/z3; visited February 2016.

[32] K. Zellag and B. Kemme. Consistency anomalies in multi-tier architectures: automatic detection and prevention. *VLDB Journal*, 23(1), 2014. http://dx.doi.org/10.1007/s00778-013-0318-x.

[33] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Software Eng.*, 28 (2), 2002. http://dx.doi.org/10.1109/32.988498.