

# A Call for Clarity on Consistency

Hussam Abu-Libdeh  
Cornell University  
Ithaca, NY 14853  
hussam@cs.cornell.edu

Robert Escriva  
Cornell University  
Ithaca, NY 14853  
escriva@cs.cornell.edu

## ABSTRACT

Designers of scalable high-performance cloud computing services make tradeoffs affecting the availability and consistency guarantees of their services. Yet, discussion of consistency models is often surrounded with vagueness and overloaded terms.

This position paper is a call to action for a clearer discussion of consistency. To start things off, we highlight common mistakes we’ve observed in both industry, and to a lesser extent, academia. We survey prominent consistency models from the past three and a half decades of research and promote clear specification of consistency guarantees in terms of explicitly stated invariants.

## Categories and Subject Descriptors

H.3.4 [Systems and Software]: Distributed Systems

## General Terms

Consistency, Specification, System Properties

## 1. INTRODUCTION

Consistency is a safety property describing an invariant that the system maintains. Presenting consistency from the perspective of systems builders is only half of the equation. One has to be careful not to fall into the all-too-common trap of defining the consistency model of a system in a nebulous fashion with respect to *which state* the system will find itself in, and not *which properties* a client may use when reasoning about the system.

Consider the presently popular “eventual consistency”; under this model, a storage system guarantees that “if no new updates are made to the object eventually all replicas will converge to the same value” [15, 16, 32]. This definition has become muddled by hand-wavy arguments in which a system is termed “eventually consistent” when people are unable or unwilling to better specify its behavior.

There is much confusion about how to properly utilize an eventually consistent storage backend. In discussing the various properties of an eventually consistent system, someone will eventually say something to the effect of, “the system, if left unperturbed will eventually become consistent and clients will see the latest values.” This statement may appear correct at first glance. After all, it attempts to paraphrase the earlier statement about eventual consistency. However, there is a deep flaw in this restatement: the original statement says nothing about what a client of the system<sup>1</sup> may expect of the system. A set of replicas in the system may converge on a common value in the absence of future updates without the clients necessarily seeing that.

This position paper calls for bringing clarity to discussions about systems’ consistency. This is necessary to understand the guarantees that different systems make, especially as consistency is traded for other properties. As a first step towards this goal, we advocate expressing consistency guarantees in terms of explicit invariants. We address common mistakes encountered in specifying a system’s consistency model and survey the most common consistency models in literature and describe them in terms of their upheld invariants. Finally, we discuss several issues related to system consistency and conclude.

## 2. COMMON MISTAKES

In this section, we address common mistakes we have encountered in the specification of various popular systems (examples include but are certainly not limited to [1, 5, 6, 19]), and to a lesser extent, in academic literature.

### 2.1 Consistency is a property

When reading the specification of well-known scalable data storage systems, one comes across discussions of consistency in terms of internal implementation-specific details, such as particular data replication, partitioning, request routing, or failure detection schemes.

However, consistency is a safety property of the system describing an invariant predicate over a client’s interaction with the system that holds over all possible execution histories of the system<sup>2</sup> [8, 9, chap.1]. This definition does not dictate the nature of the invariant or how it is implemented;

<sup>1</sup>Here we say “client”, but it may well be a developer who is building a system which will itself have clients.

<sup>2</sup>Invariants don’t necessarily have to define a client, but it is necessary for consistency properties.



of consistency model  $C$ . The  $\text{KVS}_C$  interface provides two simple operations:

$\text{PUT}(k, v)$  Associate the key  $k$  with the value  $v$ . Any previous association  $k \rightarrow v_{old}$  is overwritten.

$\text{GET}(k)$  Return the value  $v$  associated with  $k$ , or a special  $\perp$  when no value exists.

Note that the following consistency models are not listed in an “increasingly stronger” manner. There exists a partial order amongst them in terms of invariants strength, but they are not totally ordered.

### 3.1 No invariant

The simplest system maintains no read/write invariants and client operations may be responded to in any arbitrary way. Such a system is obviously not useful as clients cannot know what to expect.

### 3.2 Simplest invariant

$\text{KVS}_{\text{simple}}$  maintains the simplest invariant that would make it usable. Here a client’s  $\text{GET}(k)$  operation is responded to with either a value from *some* client’s  $\text{PUT}(k, v)$  operation or the special  $\perp$  value. Notice that under this invariant, a client may see arbitrarily stale values – even after reading the most recent value previously [4, 14, 16, 21].

### 3.3 Session consistency

$\text{KVS}_{\text{simple}}$  may be augmented to provide session guarantees [15] that specify the structure of a client’s session-oriented interaction with the system. The four common session guarantees are: *read your writes*, *monotonic reads*, *writes follow reads*, and *monotonic writes*. The former two guarantees influence only the users’ sessions. The latter two guarantees specify that one user’s behavior will be seen by all users in a predictable manner.

$\text{KVS}_{\text{RYW}}$  provides *read your writes consistency*. Specifically, it maintains the invariant that any execution of the system will only contain traces in which a client always receives values at least as new as its most recent  $\text{PUT}(k, v)$  operation when it does a  $\text{GET}(k)$  operation. Read your writes consistency does not specify anything about what the global execution will do for concurrent users’ sessions.

$\text{KVS}_{\text{MR}}$  provides *monotonic reads consistency*. For all executions, a system is said to support monotonic reads if it is the case that a client will never read more-stale data after observing a version which is less-stale. There is no statement made about the liveness of the system and a client may forever read the stale data, even if new versions are written.

$\text{KVS}_{\text{WFR}}$  provides *writes follow reads consistency* by preserving dependencies between writes and the data upon which they depend. The system maintains the invariant that an execution will only show a client observing a write if the client also is able to observe all data upon which the write depends.

$\text{KVS}_{\text{MW}}$  provides *monotonic writes consistency*. The system maintains the invariant that every client of the system sees the writes of every other client of the system according to the the partial orders imposed by every other client. The manner in which clients’ operations are interleaved is left unspecified by this session guarantee.

### 3.4 Causal consistency

*Causal consistency* [24] respects the dependencies between system events according to a “happens before” relation. Events internal to processes and the communication between pairs of processes define the partial ordering over all events (with inter-process causal dependency being when one process does a  $\text{GET}(k)$  that returns the value of the  $\text{PUT}(k, v)$  of another process). Valid executions of a causally consistent system will contain execution traces of each process which show events that are processed in accordance with the “happens before” relation. Clients of  $\text{KVS}_{\text{causal}}$  will all observe the same ordering of dependent events; however, the ordering of concurrent events is left unspecified.

It is worth noting that causal consistency is most useful when utilized by a service which exposes information about the “happens before” relation. For example,  $\text{KVS}_{\text{causal}}$  may expose a vector of logical clocks to the client on each  $\text{GET}(k)$ . Clients may use the returned logical clocks to provide the system with information about causal dependencies on  $\text{PUT}(k, v)$  and operations.

In the absence of an explicit means of handing control over the “happens before” relation to clients, the server must manage the “happens before” relation implicitly by tracking client interactions. A causally consistent scheme in which clients take no part in explicitly managing the “happens before” relation is equivalent to the properties provided by sequential consistency.

### 3.5 Sequential consistency

A *sequentially consistent system* [25] provides a guarantee that every execution of a sequentially consistent system obeys the partial order on events imposed by each client’s view. If several clients of  $\text{KVS}_{\text{sequential}}$  timestamp their interactions with the system, then the resulting realtime order of all clients’ interactions may not reflect the actual order in which the system executes the operations.

### 3.6 Linearizability

*Linearizability* [26] respects a real time ordering of events. A linearizable system presents the interface of an object with associated operations which may be performed. All executions of a linearizable object are equivalent to a valid sequence of operations on the object in which the initiation and termination of operations happen in lock-step, and the resulting order of operations does not contradict the original order of operations. Linearizability is a local invariant, and therefore allows for the composition of linearizable objects into systems which are also linearizable.  $\text{KVS}_{\text{linearizable}}$  provides clients with an ideal key-value store. A  $\text{GET}(k)$  operation always returns the latest value, and conflicting  $\text{PUT}(k, v)$  operations will be observed to be ordered the same by all clients of the system.

### 3.7 Serializability

*Serializability* [11, 23] is the guarantee that the execution of concurrent transactions is equivalent to some serial execution of the transactions on the data store. Serializability differs from linearizability in that the latter is modeled as a set of operations on linearizable objects, each of which completes independently of the others, while the former is a means of coordinating a set of writes across objects such that they appear to happen as one indivisible action. Intuitively, serializability specifies a means of altering a set of objects in arbitrary ways, while linearizability specifies a means of altering individual objects in ways which are tied to the definition of the objects themselves. As a consequence, the composition of two serializable schedules is not serializable.

Clients of  $KVS_{\text{serializable}}$  are able to begin a transaction, perform an arbitrary number of  $GET(k)$  and  $PUT(k, v)$  operations, and then commit the transaction. The transaction will commit if and only if it may do so without violating the serializability invariant.

### 3.8 Snapshot isolation

*Snapshot isolation* [7] provides transactions with a consistent view of the data on which all read operations may be performed. Specifically, all executions of a system which provides snapshot isolation contain transactions which are ordered such that all reads of a transaction occur at the same logical version of the database, and no transaction commits if another transaction which has written the same values as the pending transaction has committed between the start and end times of the pending transaction.

$KVS_{\text{snapshot}}$  provides clients with the ability to read large volumes of data from the same logical instant in time. Transactions which include  $PUT(k, v)$  operations may generate schedules which are not serializable, and would not be generated through any serial execution of the transactions.

## 4. DISCUSSION

We have briefly surveyed common consistency models and described them in terms of explicitly defined system invariants without any mention of implementation details. Clients are knowledgeable of the system’s interface and the invariants it maintains, and make no further assumptions of the it’s implementation or execution details.

The use of explicit invariants helps separates the policy (the consistency model) from the mechanism (the implementation details). This separation simplifies reasoning about system behavior agnostic to its implementation.

### 4.1 Strong vs. weak consistency

In our advocacy for clearly defining expectations, we have deliberately chosen not to use broad terms such as as strong or weak consistency. Both terms refer to full categories of consistency models and can result in unwanted confusion if not used carefully. For example, though both linearizability and sequential consistency are classified as strong consistency models, the first is composable and the other is not. If systems are labeled simply as *strongly consistent*, the client is left to wonder which properties it can rely upon. A similar argument can be made for the various flavors of weak

consistency. The main point here is that vague or subjective categorical descriptions leave room for misinterpretation and confusion, thus we favor abandoning them in favor of explicit precise definitions that help manage user expectation and reasoning about the system.

### 4.2 What about eventual consistency?

Many datacenter and cloud-computing services assert they provide eventual consistency, which is often defined operationally such that “if no new updates are made to the object, eventually all accesses will return the last updated value”. However this is a liveness property. Alternative, definitions state that “all replicas converge to the same value in the absence of further updates”. This also does not define a safety property invariant that the client can use to reason about the system. Finally, [8] provides a more formal definition of eventual consistency.

For these reasons, we suggest that the term “eventual consistency” be used only when coupled with stronger descriptions of the system which provide quantifiable properties about what a user may expect from the system.

### 4.3 Choosing the right invariants

We have advocated the use of explicitly defined invariants for describing a system’s consistency guarantees. This is not an independent decision and is largely motivated by prior work [9], and our experience building systems. Choosing the right invariants is ultimately a design decision.

#### 4.3.1 Per-client invariants

A system need not provide the same consistency guarantees for all of its clients; instead it can maintain invariants based on client “roles”. For example, consider a hospital system; a physician’s client might require linearizable consistency guarantees when accessing the list of patients and records whereas a front-desk client might require weaker guarantees. Thus the system does not need to be monolithic in its consistency guarantees, but instead can maintain multiple clearly defined role-based invariants without complicating reasoning about its operation.

#### 4.3.2 Balancing safety and liveness

The CAP theorem implies that developers must carefully specify the invariants that a system satisfies. For instance, there have been recent network redundancy trends [2, 12, 13, 20, 22, 28, 30] that might encourage designers to relax a system’s invariants as related to partition tolerance in order to maintain different consistency invariants [27]. This, of course, is a design decision that depends on the particular service deployment.

#### 4.3.3 Interface transparency

In single administrative domain settings, it might be beneficiary to have the system’s interface expose more of the underlying implementation such that clients might take a more active role in obtaining their desired properties.

For example, in a traditionally constructed RDBMS system which implements primary/backup [29], it is possible to operate in an asynchronous mode. This system will provide strong consistency while all components function properly;

however, it may lose recently committed data, or allow stale reads in the presence of failure of the primary. This system can be thought of as having two different consistency models.

We advocate for describing systems in terms of both the normal and the failure cases. Ideally, the consistency mode of the system should be exposed to the client via an observable interface. Ideally, this interface should be programmatic in nature so that a system may communicate to all clients that the invariants it is able to maintain are relaxed or different from the normal invariants maintained by the system.

For example, a Dynamo-like system which exposes causally consistent data could inform the client about the impact of the current replica placement strategy in use. In turn, clients may adapt their behavior based upon the observed consistency model of the system.

## 5. SUMMARY

Consistency is a safety property describing a system invariant. In this paper, we surveyed common consistency models, describing them in terms of the invariants they maintain. We also addressed commonly encountered mistakes in specifying system consistency and advocated the use of clearly defined invariants instead of vague, or operational, description of what should clients expect from the system. Using explicit invariants makes it easier for clients to reason about the behavior of systems.

## 6. ACKNOWLEDGMENTS

We thank Ken Birman, Mark Reitblatt, Emin Gün Sirer, and Robbert van Renesse for their insightful comments and valuable feedback during our discussions.

## References

- [1] 10gen, Inc. MongoDB. <http://www.mongodb.org/>.
- [2] Albert G. Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. VL2: A Scalable And Flexible Data Center Network. In *Proceedings of the SIGCOMM Conference*, pages 51-62, 2009.
- [3] Amazon Web Services LLC. Amazon Simple Storage Service. <http://aws.amazon.com/s3/>.
- [4] Andrew D. Birrell, Roy Levin, Michael D. Schroeder, and Roger M. Needham. Grapevine: An Exercise In Distributed Computing. In *Communications of the ACM*, 25:260–274, ACM, New York, New York, April 1982.
- [5] Avinash Lakshman and Prashant Malik. Cassandra - A Decentralized Structured Storage System. In *Proceedings of the International Workshop on Large Scale Distributed Systems and Middleware*, Big Sky, Montana, 2009.
- [6] Basho Technologies, Inc. Riak. [http://www.basho.com/products\\_riak\\_overview.php](http://www.basho.com/products_riak_overview.php).
- [7] Berenson, Hal, Bernstein, Phil, Gray, Jim, Melton, Jim, O’Neil, Elizabeth, and O’Neil, Patrick. A Critique Of Ansi Sql Isolation Levels. In *Proceedings of the SIGMOD Conference*, pages 1–10, San Jose, California, May 1995.
- [8] Bernadette Charron-Bost, Fernando Pedone, and Andre Schiper. *Replication: Theory And Practice*. Springer, New York, New York, 2010.
- [9] Bowen Alpern and Fred B. Schneider. Recognizing Safety And Liveness. In *Distributed Computing*, 2:117-126, Springer Berlin / Heidelberg, 1987.
- [10] Brad Fitzpatrick. Distributed Caching With Memcached. In *Linux Journal*, 2004:5– Specialized Systems Consultants, Inc., Seattle, Washington, August 2004.
- [11] Christos H. Papadimitriou. The Serializability Of Concurrent Database Updates. In *Journal of the ACM*, 26:631–653, ACM, New York, New York, October 1979.
- [12] Chuanxiong Guo, Guohan Lu, Dan Li, Haitao Wu, Xuan Zhang, Yunfeng Shi, Chen Tian, Yongguang Zhang, and Songwu Lu. Bcube: A High Performance, Server-centric Network Architecture For Modular Data Centers. In *Proceedings of the SIGCOMM Conference*, pages 63-74, 2009.
- [13] Chuanxiong Guo, Haitao Wu, Kun Tan, Lei Shi, Yongguang Zhang, and Songwu Lu. Dcell: A Scalable And Fault-tolerant Network Structure For Data Centers. In *Proceedings of the SIGCOMM Conference*, pages 75-86, 2008.
- [14] Derek C. Oppen and Yogen K. Dalal. The Clearinghouse: A Decentralized Agent For Locating Named Objects In A Distributed Environment. In *ACM Transactions on Information Systems*, 1:230–253, ACM, New York, New York, July 1983.
- [15] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike Spreitzer, Marvin Theimer, and Brent B. Welch. Session Guarantees For Weakly Consistent Replicated Data. In *Proceedings of the International Conference on Parallel and Distributed Information Systems*, pages 140-149, IEEE Computer Society, Austin, Texas, September 1994.
- [16] Douglas B. Terry, Marvin Theimer, Karin Petersen, Alan J. Demers, Mike Spreitzer, and Carl Hauser. Managing Update Conflicts In Bayou, A Weakly Connected Replicated Storage System. In *Proceedings of the Symposium on Operating Systems Principles*, pages 172-183, Copper Mountain, Colorado, December 1995.
- [17] Eric A. Brewer. Towards Robust Distributed Systems. (invited Talk). In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, Portland, Oregon, July 2000.
- [18] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. Bigtable: A Distributed Storage System For Structured Data. In *Proceedings of the Symposium on Operating System Design and Implementation*, pages 205-218, Seattle, Washington, November 2006.

- [19] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of the Symposium on Operating Systems Principles*, pages 205-220, Bretton Woods, New Hampshire, October 2007.
- [20] Hussam Abu-Libdeh, Paolo Costa, Antony I. T. Rowstron, Greg O'Shea, and Austin Donnelly. Symbiotic Routing In Future Data Centers. In *Proceedings of the SIGCOMM Conference*, pages 51-62, 2010.
- [21] James J. Kistler and Mahadev Satyanarayanan. Disconnected Operation In The Coda File System. In *Proceedings of the Symposium on Operating Systems Principles*, pages 213-225, Pacific Grove, California, October 1991.
- [22] Jayaram Mudigonda, Praveen Yalagandula, Mohammad Al-Fares, and Jeffrey C. Mogul. Spain: Cots Data-center Ethernet For Multipathing Over Arbitrary Topologies. In *Proceedings of the Symposium on Networked System Design and Implementation*, pages 18-18, 2010.
- [23] Kapali P. Eswaran, Jim Gray, Raymond A. Lorie, and Irving L. Traiger. The Notions Of Consistency And Predicate Locks In A Database System. In *Communications of the ACM*, 19:624-633, ACM, New York, New York, November 1976.
- [24] Leslie Lamport. Time, Clocks, And The Ordering Of Events In A Distributed System. In *Communications of the ACM*, 21:558-565, ACM, New York, New York, July 1978.
- [25] Leslie Lamport. How To Make A Multiprocessor Computer That Correctly Executes Multiprocess Programs. In *IEEE Transactions on Computers*, 28:690-691, 1979.
- [26] Maurice P. Herlihy and Jeanette M. Wing. Linearizability: A Correctness Condition For Concurrent Objects. In *ACM Transactions on Programming Languages and Systems*, 12:463-492, ACM, New York, New York, July 1990.
- [27] Michael Stonebreaker. Errors In Database Systems, Eventual Consistency, And The Cap Theorem. <http://cacm.acm.org/blogs/blog-cacm/83396-errors-in-database-systems-eventual-consistency-and-the-cap-theorem/fulltext>.
- [28] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A Scalable, Commodity Data Center Network Architecture. In *Proceedings of the SIGCOMM Conference*, pages 63-74, 2008.
- [29] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. *The Primary-backup Approach*. pages 199-216, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993.
- [30] Radhika Niranjana Mysore, Andreas Pamboris, Nathan Farrington, Nelson Huang, Pardis Miri, Sivasankar Radhakrishnan, Vikram Subramanya, and Amin Vahdat. Portland: A Scalable Fault-tolerant Layer 2 Data Center Network Fabric. In *Proceedings of the SIGCOMM Conference*, pages 39-50, 2009.
- [31] Seth Gilbert and Nancy Lynch. Brewer's Conjecture And The Feasibility Of Consistent, Available, Partition-tolerant Web Services. In *SIGACT News*, 33:51-59, ACM, New York, New York, June 2002.
- [32] Werner Vogels. Eventually Consistent - Revisted. [http://www.allthingsdistributed.com/2008/12/eventually\\_consistent.html](http://www.allthingsdistributed.com/2008/12/eventually_consistent.html).