Consistability: Describing usually consistent systems

Amitanand S. Aiyer, Eric Anderson, Xiaozhou Li, Mehul A. Shah, Jay J. Wylie
Hewlett-Packard Laboratories
1501 Page Mill Road, Palo Alto, CA 94304, USA

Abstract

Current weak consistency semantics provide *worst-case* guarantees to clients. These guarantees fail to adequately describe systems that provide varying levels of consistency in the face of distinct failure modes, or that achieve better than worst-case guarantees during normal execution. The inability to make precise statements about consistency throughout a system's execution represents a lost opportunity to clearly understand client application requirements and to optimize systems and services appropriately. In this position paper, we motivate the need for and introduce the concept of *consistability*—a unified metric of consistency and availability. Consistability offers a means of describing, specifying, and discussing how much consistency a *usually* consistent system provides, and how often it does so. We describe our initial results of applying consistability reasoning to a key-value store we are developing and to other recent distributed systems. We also discuss the limitations of our consistability definition.

1 Introduction

Internet applications tend to stress availability rather than consistency, and so we believe it is important to be able to describe the trade-offs present for such applications. Towards that end, we introduce the notion of *consistability* which provides a way of describing how the consistency that a system provides may change during its execution. Our inspiration for consistability comes from the performability metric [20] that defined a unified metric for examining the fraction of time a system provides a given level of performance. *Performability* describes how often a system operates above some performance level. An example performability statement is that 90% of the time the system processes at least 1000 *put* requests per second and 99% of the time it processes at least 500. Performability captures the notion that a fault-tolerant system which degrades gracefully will offer different levels of performance, at different times, depending on the system's health. Performability also captures the notion that the system may be considered unavailable, or to have failed, if a certain minimum level of performance is not achieved. Our hope is that consistability will allow us to make similar statements about systems such as the following: The system offers linearizability 90% of the time, and eventual consistency (within two hours) 99.9% of the time.

The tension between strong consistency, availability, and partition-tolerance is well-known (e.g., [9]). However, it is possible to offer weak consistency with good availability and partition-tolerance (e.g., [6, 7, 5]). Unfortunately, there is a dearth of language with which to precisely describe such weakly consistent systems. Whereas we can describe how performance and availability of a system degrade in the face of failures, we cannot describe the consistency of a system that is *usually* consistent.

Consistency definitions provide *worst-case* guarantees. Such guarantees fail to capture the possibility that during failure-free periods the system can *provide* stronger guarantees. Beyond this, the normal language of consistency does not allow us to describe the fact that a system may *achieve* better consistency than the worst-case guarantee. We want to be able to describe and understand a system that provides distinct consistency classes throughout its execution, and that offers different applications distinct, appropriate consistency guarantees.

Part of our motivation for defining consistability comes from our experience in specifying and designing a global-scale *key-value storage* (KVS) system. By *key* we mean some client-specified unique string, by *value* we mean a client-specified binary object of arbitrary size, and by *store* we mean it offers a put-get interface. The design goals for the KVS, in order, are as follows: low-cost, availability, reliability (including disaster-tolerance), performance, and as strong consis-

tency as possible. To achieve disaster-tolerance, the KVS must necessarily span multiple data centers. The design goal of availability is in tension with the goals of disaster-tolerance and strong consistency. Effectively, we want the KVS to provide *best effort* consistency: i.e., strong consistency in the face of most failures, and weaker consistency during a network partition or other large correlated failure.

The definition of consistability and approach to evaluating consistability that we describe in this position paper is an initial step towards developing a language capable of precisely evaluating usually consistent systems. As a metric, consistability offers an avenue by which to compare and contrast different techniques for implementing similar weak consistency guarantees. Consistability has helped us understand the affect of failures on the distinct consistency guarantees that the KVS can offer. This is especially useful to us since we intend the KVS to support many distinct client applications, each with slightly different consistency requirements; consistability will allow us to understand which applications can be supported with what availability.

2 Related Work

We focus our discussion of related work on consistency guarantees that we considered during the design of the KVS. We discuss other additional related systems in the case studies section §4 and in the discussion section §5.

Lamport's definition of *atomic* registers provides the basis of most definitions of *strong* consistency [15]. Atomic registers guarantee that all reads and writes to a single object can be ordered, in a manner consistent with their relative orderings, such that reads always return the value of the latest completed write. Linearizability [11] and transactions [10] define strong consistency of operations that affect multiple objects. In this paper, we limit our discussion of consistability to an individual object. Lamport also defines *regular* and *safe* registers that have weaker consistency [15]. Regular registers allow a read operation concurrent to write operations to return the value from any concurrent write, or the previous completed write. Safe registers allow a read operation concurrent to write operations to return any value.

There are many forms of *weak* and *eventual* consistency; we discuss some examples of each to provide context for consistability. In the TACT project, Yu and Vahdat propose three axis along which consistency can be weakened to improve availability: numerical-error, order-error, and staleness [25]. PRACTI [3] also offers tunable consistency levels. The *bounded ignorance* techniques of Krishnakumar and Bernstein [14] permits concurrent transactions to complete while being ignorant of some bounded number of other concurrent transactions. *K-atomic* registers, proposed by Aiyer et al. [1], bound the number of distinct recently completed writes that may be returned by a read operation. Malkhi et al. proposed probabilistic quorums [19] with which Lee and Welch have shown how to build a P*-randomized* register [17] whose staleness follows probability distribution function P.

In eventually consistent systems, a write operation may return before its value propagates throughout the system; in such systems, once all transient failures have been resolved, and no further writes are issued, replicas converge to a consistent state [23]. Examples of research on weak consistency in the presence of network partitions include the following: Davidson et al. [6] survey partition-aware database techniques, Babaoğlu et al. developed partition-aware systems [2], and Pleisch et al. have clarified the partitionable group membership problem [22]. The definitions of P-registers, K-registers, and *conits* in TACT, do not explicitly guarantee best-case consistency in well-conditioned executions. These definitions therefore lack the sense that as concurrency, or latency, or the number of failures increase, the consistency provided gracefully approaches the worst-case guarantee.

3 Consistability

We define consistability for a single object in three steps. First, we define consistency classes and failure scenarios. Then we describe how to map failure scenarios to consistency classes. Finally, we describe how to calculate consistability given this mapping.

3.1 Consistency classes & failure scenarios

To define consistability, all of the possible consistency classes achievable must be enumerated. Let C be the set of all consistency classes the system can provide. A *consistency class* is a definition of some form of consistency, for example, atomic/regular/safe [15], linearizable [11], or some weaker variants (e.g., [25]).

The consistency classes in C can be partially ordered. For example, atomicity is stronger than regularity [15], K-atomicity is weaker than atomicity [1], but K-atomicity and regularity are not comparable. Figure 1 provides an illustrative example of a lattice of consistency classes that includes the traditional register consistency classes, the recent K-variants of those classes, and some eventual consistency guarantees based on time in seconds. In this example lattice, none of the consistency classes for traditional and K registers are comparable with any of the consistency classes for eventual guarantees.

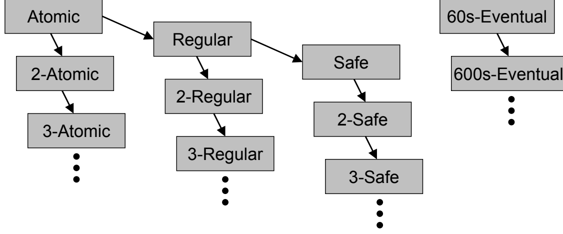

Figure 1: Lattice of consistency classes.

Let F be the set of all possible failure scenarios. A *failure scenario* is a description of a specific failure pattern; failure scenarios are, by definition, disjoint. Examples of failure scenarios include "no failures," "all failures of a single server and no other failures," "a network partition and no other failures," and "all failures of a single server and a network partition."

3.2 The mapping

To calculate consistability, every failure scenario must be mapped to some subset of all of the consistency classes:

$$\mathscr{C} : F \to 2^C$$

Given a failure scenario, f, the mapping, $\mathscr{C}(f)$, indicates the set of consistency classes achievable by the system in f. If the system provides no service in a failure scenario f, then $\mathscr{C}(f) = \emptyset$. The $\emptyset$ corresponds to either unavailability or inconsistency. The mapping $\mathscr{C}(f)$ may map f to consistency classes that are not comparable. For example, a system may provide both 2-Regular and 60s-Eventual for some failure scenario. This mapping is an example of combinatorial fault tolerance modeling [24]: it maps all possible failures of interest to system behaviors.

We say that a consistency class c is *maximal* for a failure scenario f, if there is no consistency class in $\mathscr{C}(f)$ that is stronger than c. Based on the partial order among the consistency classes, one could define the mapping from failure scenarios to the set of maximal consistency classes which are achievable. For such an alternative mapping, if there is a total order among all consistency classes in C then each failure scenario would map to exactly one (maximal) consistency class.

3.3 Consistability defined

Consistability is the expected portion of time that a system provides each consistency class. To provide such an expectation, like reliability and performability metrics do, we need probability distributions over the failure scenarios. Let $\mathrm{FProb}(f)$ denote the probability that the system is in a given failure scenario. We compute the probability of providing a consistency class, $\mathrm{CProb}(c)$, by summing the probabilities of each failure class which maps to the consistency class:

$$\forall c \in C, \mathrm{CProb}(c) = \sum_{f \in F} \begin{cases} \mathrm{FProb}(f) & \text{if } c \in \mathscr{C}(f), \\ 0 & \text{otherwise.} \end{cases}$$

Because failure scenarios are disjoint, $\sum_{f \in F} \mathrm{FProb}(f) = 1$. However, $\sum_{c \in C} \mathrm{CProb}(c) \geq 1$. This is because the mapping from a failure scenario to a consistency class can be one-to-many.

There is no need to include every possible consistency class in C: only the ones that a client may care about, that an application can effectively use, or that the system may actually provide are of interest. Once C is decided, the system designer may be able to define a single appropriate failure scenario for each consistency class. I.e., figuring out the appropriate members of C and F may be an iterative process. To account for correlated failures, failure scenarios must be defined that include many different failures and the probability distribution over failure scenarios must capture this correlation.

We expect that the practice of mapping failure scenarios to consistency classes will help designers better understand all of the capabilities of their distributed systems and protocols. Unfortunately, it requires significant effort to produce an actual consistability measure because it is difficult to determine the probability distributions over all failure scenarios. This is similar to the situation designers are in when they work with fault tolerance and reliability today: the former is easy to quantify and reason about during design; whereas the latter is difficult to evaluate and measure, and applies only narrowly to a specific, actual system instance.

4 Case studies

In this section, we present a key-value store system that we are building as a case study to illustrate the various consistability concepts presented in the previous section. We also discuss other prior systems that make interesting consistability choices.

4.1 Key-value store: Design

Figure 2 illustrates the system model for our key-value store (KVS). It shows three data centers (X, Y, Z) with their respective storage and proxies. It also shows six clients (a to f) that usually access the system through their closest proxy. In this position paper we ignore the design decisions about metadata (i.e., how clients determine which storage nodes host fragments for some key), but we do discuss whether the metadata service is available or not.

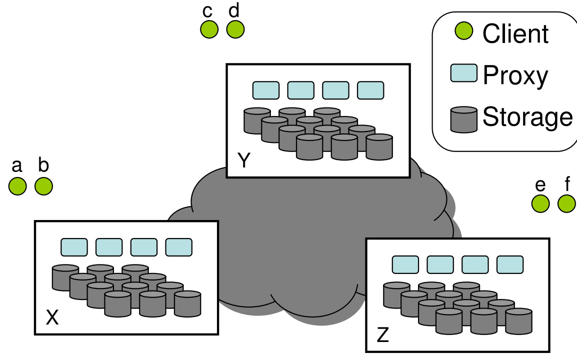

Figure 2: System model.

For the sake of availability, the KVS must allow get and put operations during a network partition and handle reconciliations when the network reconnects. To cost-effectively achieve reliability, values are erasure-coded across multiple data centers in such a manner so that each data center has sufficient erasure-coded fragments to recover the value. The put operation offers best effort consistency and disaster-tolerance. Once the metadata service selects the storage nodes for a particular operation, that operation will not complete until it has written to or timed out on all the selected nodes. The get operation offers best effort availability: if it cannot retrieve the latest version of the value put into the system, it attempts to retrieve prior versions until a value is returned (or all prior versions have been tried). There is a clear tension between the availability and disaster-tolerance requirements. Roughly speaking, we have designed a protocol that is usually consistent, and only degrades to worst-case consistency when specific failures occur.

4.2 Key-value store: Consistability

For the KVS, we consider the following failure scenarios: (1) There are at most m disks unavailable (and the erasure code tolerates up to m disk failures). (2) More than m disks are unavailable. (3) A data center has failed catastrophically. (4) The data centers are partitioned into multiple groups. (5) The metadata service is unavailable. For the sake of brevity, we do not discuss composite failure scenarios (e.g., failure scenarios (2) and (4) occurring simultaneously).

Failure scenario (1) is effectively the "no failures" scenario because the erasure code tolerates up to m disk failures. Failure scenarios (1) and (3) map to the regular consistency class. In both cases, a get operation concurrent to a put operation will return some value being put concurrently, or the most recent completed put value. For case (1), the get operation will return either the prior value or the value being put concurrently, depending on message ordering at each server. For case (3), regularity is achieved because each data center has sufficient erasure-coded fragments to recover a value and so the most recently put value is available after such a catastrophe. Even though only regularity can be provided in these failure scenarios, there are many executions that will achieve atomicity. For example, if every client preferentially selects the same node for metadata services, then the operations will achieve atomicity.

Failure scenarios (2) and (4) map to the same consistency class: K-regularity, a get returns one of the K most recent values put [1]. In both scenarios, some recently put values may be lost because too few erasure-coded fragments are available for get to recover them. Unfortunately, without bounding the number of new, or concurrent, put operations, $K = \infty$. For failure scenario (4), given a bound of p incomplete put operations, and d data centers, we can set $K = p \cdot d$, as each data center could be in its own partition.

Failure scenario (5) maps to the null set. In general, mapping to the null set means that the system is either unavailable or inconsistent. For failure scenario (5), the system is simply unavailable since no operations can complete.

We do not have probability distributions for the possible failure scenarios and so cannot compute the overall consistability of the KVS system. Instead, our current focus is on showing that for each of the failure scenarios, the KVS protocol does indeed map to the consistency class we have identified. We first use the TLC [16] model-checker to verify that the consistency class semantics are obeyed for small instances of the protocol, and then use hand-proofs to prove that the consistency class is satisfied. We have described the KVS protocol along with some consistency classes and failure scenarios in TLA+ and model-checked them using TLC. Our success in model checking the specification has been limited by the size of the system. For the failure-free case with a single data center, we have verified that put and get operations are atomic. However, verifying the extended specification for multiple data centers with distributed metadata service takes too long to check all possible states. We have used TLC to verify that failure scenarios (1) and (3) map to regularity, and used hand-proofing to prove it. For failure scenarios (2) and (4) we hand-proved that the protocol only provides $\infty-$regular consistency class, unless the number of disconnected put operations are bounded. Further, we hand-proved that by using failure detectors, or by restricting the number of disconnected put operations, the KVS protocol provides K-regularity for an appropriately chosen value of K.

4.3 Dynamo and PNUTS

There are other systems that try to offer best-effort consistency and so provide better consistency when networks are mostly-connected and responsive than when failures are present, or believed to be present. Unfortunately, the worst-case nature of most weak consistency guarantees discourages discussion of how much consistency a system provides in the common case.

Dynamo [7] uses a *hinted handoff* mechanism to tolerate temporary node or network failures. If one of the nodes responsible for storing an object (say A) is unavailable, data is stored at a different node (say B). B will later attempt to deliver the object to A, when it sees that A has recovered. In the meanwhile, if a reader attempts to read the object from A, before A receives the updated version from B, the reader can get an out-of-date value. We believe that the failure-free scenario in Dynamo maps to an atomic consistency class. The failure scenario involving a bounded number of node failures/unavailability maps to regular consistency class. Finally, the failure scenario involving a number of node failures that exceeds the bound leads to unavailability.

PNUTS [5] explicitly exposes the consistency classes it offers via an extensive storage interface: read-any, read-critical, read-latest, write, and test-and-set-write. In PNUTS, applications implicitly specify portions of their consistability requirements via their use of the storage interface. The failure scenarios that map to each consistency class and that ensure the availability of certain storage interfaces depends on how PNUTS is configured and used.

4.4 Beyond one third faulty BFT

Two recent Byzantine fault tolerant (BFT) systems provide interesting consistability. Li and Mazieres [18] developed BFT2F, which is a BFT protocol that offers different guarantees depending on whether fewer than one third of the replicas are faulty (linearizability), or if fewer than two thirds of the replicas are faulty (fork* consistency). Similarly, Chun et al. [4] developed Attested Append-only Memory (A2M). A2M provides safety and liveness if fewer than one third of the replicas are faulty, but only safety if fewer than two thirds of the replicas are faulty. Liveness in this sense could be thought of as performability requirement. The set of failure scenarios are the same for both of these systems, but the mapping to consistency classes for a single failure scenario differs.

5 Discussion

An open problem with consistability is how to reason about the consistency classes provided as the system transitions between failure scenarios. It is simple to reason about all operations that occur during a specific failure scenario because the consistency class remains constant. If, for example, a *put* operation occurs during some failure scenario involving a network partition, and then a *get* operation occurs immediately after the network merges, then which consistency class applies? If the consistency classes are comparable, then the weaker class should apply. If the consistency classes are not comparable, then it is unclear what should happen.

Beyond understanding which consistency class applies after a transition, we need to better understand when the consistency class does in fact apply. We expect that a system which transitions to a worse failure scenario immediately switches to a consistency class which maps to that failure scenario. However, we expect that a system which transitions to a better failure scenario experiences a transition period during which increasingly more objects achieve the better consistency class.

One of the benefits of consistability is that clients can more precisely articulate their requirements. Consistability simply complements any performability measure (i.e., consistability is not a generalization of performability). Given models of system performance in each failure scenario and a model of the workload, we believe it is possible to develop a unified measure of performance, consistency, and availability. By combining failure frequency information with the consistability achieved by a system, a client could write a service level agreement (SLA) that would allow them to articulate the value they place on achieving "more consistency, more often". Unfortunately, both clients and servers need to be able to verify that an SLA is being met. Ideally, a system can tell a client which consistency class it provides, and potentially, what consistency class it is currently achieving (if it is offering better than worst-case consistency). We refer to this ability as *introspection*.

We believe that prior work on *adaptive* protocols may provide some guidance on how to implement introspection. For example, Hiltunen et al. developed methods for adaptive fault tolerance [13] and customizable failure models [12]. The fail-awareness work of Mishra, Fetzer, and Cristian [8, 21] is also adaptive in nature since servers trigger an exception that clients catch if the server detects that it is not operating under normal conditions. Introspection and adaptive protocols may complicate the specification and evaluation of a system's consistability.

6 Summary

In this position paper, we argued that there is a need for a richer language with which to discuss the consistency of *usually* consistent systems. Traditionally, understanding of consistency is limited to the *worst-case* guaran-

tees provided to clients. We introduced the concept of *consistability* to facilitate describing and understanding systems that can provide or achieve better than worst-case consistency under some conditions. Consistability is based on mapping possible failure scenarios to consistency classes that the system can provide during the failure scenario. Our motivation for enriching the language with which to reason about consistency was our ongoing effort to design a global-scale key-value store (KVS). A key design goal is for the KVS is to offer best effort consistency during failure-free periods and in the face of network partitions, data center catastrophes, and correlated failures. Consistability allows us to compare techniques and discuss trade-offs more precisely. We used the consistability definition to understand that mapping of failure scenarios to consistency classes for the KVS and other recent distributed systems such as Dynamo [7], PNUTS [5], BFT2F [18], A2M [4]. Finally, we discussed open problems with consistability. For example, understanding the mapping of failure scenario transitions to consistency classes, and if it is possible, via introspection, to determine that the consistency class achieved exceeds the worst-case consistency class.

References

[1] A. Aiyer, L. Alvisi, and R. A. Bazzi. On the availability of non-strict quorum systems. In *Proc. 19th DISC*, pages 48–62, September 2005.

[2] O. Babaoğlu, R. Davoli, and R. Montresor, Alberto and-Segala. System support for partition-aware network applications. *ACM SIGOPS Operating System Review*, 32(1):41–56, 1998.

[3] N. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. PRACTI replication. In *Proc. 3rd NSDI*, pages 59–72, May 2006.

[4] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. In *Proc. 21st SOSP*, pages 189–204, October 2007.

[5] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s hosted data serving platform. In *Proc. 34th VLDB*, pages 1277–1288, August 2008.

[6] S. B. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in partitioned networks. *ACM Computing Surveys*, 17(3):341–370, September 1985.

[7] G. DeCandia et al. Dynamo: Amazon's highly available key-value store. In *Proc. 21st SOSP*, pages 205–220, October 2007.

[8] C. Fetzer and F. Cristian. Fail-awareness in timed asynchronous systems. In *Proc. 15th PODC*, pages 314–322, May 1996.

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

[10] J. Gray and A. Reuter. *Transaction Processing : Concepts and Techniques*. Morgan Kaufmann, October 1992.

[11] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[12] M. A. Hiltunen, V. Immanuel, and R. D. Schlichting. Supporting customized failure models for distributed software. *Distributed Systems Engineering*, 6(3):103–111, December 1999.

[13] M. A. Hiltunen and R. D. Schlichting. Adaptive distributed and fault-tolerant systems. *Computer Systems Science and Engineering*, 11(5):275–285, September 1996.

[14] N. Krishnakumar and A. J. Bernstein. Bounded ignorance in replicated systems. In *Proc. 10th PODS*, pages 63–74, May 1991.

[15] L. Lamport. On interprocess communication, Part I: Basic formalism and Part II: Algorithms. *Distributed Computing*, 1(2):77–101, June 1986.

[16] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[17] H. Lee and J. L. Welch. Randomized registers and iterative algorithms. *Distributed Computing*, 17(3):209–221, March 2005.

[18] J. Li and D. Mazieres. Beyond one-third faulty replicas in Byzantine fault tolerant systems. In *Proc. 4th NSDI*, pages 131–144, April 2007.

[19] D. Malkhi, M. K. Reiter, A. Wool, and R. N. Wright. Probabilistic quorum systems. *Information and Computation*, 170(2):184–206, November 2001.

[20] J. F. Meyer. On evaluating the performability of degradable computing systems. *IEEE Transactions on Computers*, C-29(8):720–731, August 1980.

[21] S. Mishra, C. Fetzer, and F. Cristian. The Timewheel group communication system. *IEEE Transactions on Computers*, 51(8):883–899, August 2002.

[22] S. Pleisch, O. Rütti, and A. Schiper. On the specification of partitionable group membership. In *Proc. 7th EDCC*, pages 37–45, May 2008.

[23] Y. Saito and M. Shapiro. Optimistic replication. *ACM Computing Surveys*, 37(1):42–81, March 2005.

[24] D. P. Siewiorek and R. S. Swarz. *Reliable computer systems (3rd ed.): design and evaluation*. A. K. Peters, Ltd., Natick, MA, USA, 1998.

[25] H. Yu and A. Vahdat. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Transactions on Computer Systems*, 20(3):239–282, August 2002.