

Eventual Consistency: How soon is eventual?

An Evaluation of Amazon S3's Consistency Behavior

David Bermbach and Stefan Tai
Karlsruhe Institute of Technology
Karlsruhe, Germany
firstname.lastname@kit.edu

ABSTRACT

Over the last few years, Cloud storage systems and so-called NoSQL datastores have found widespread adoption. In contrast to traditional databases, these storage systems typically sacrifice consistency in favor of latency and availability as mandated by the CAP theorem, so that they only guarantee eventual consistency. Existing approaches to benchmark these storage systems typically omit the consistency dimension or did not investigate eventuality of consistency guarantees. In this work we present a novel approach to benchmark staleness in distributed datastores and use the approach to evaluate Amazon's Simple Storage Service (S3). We report on our unexpected findings.

Categories and Subject Descriptors

H.3.4 [Information Systems]: Systems and Software—*Distributed systems, Amazon S3*; D.2.8 [Software Engineering]: Metrics—*Performance Measures, Consistency*

General Terms

Measurement, Performance, Experimentation

Keywords

Cloud Computing, Amazon S3, Eventual Consistency

1. INTRODUCTION

The Web with its continuously growing user and application base is producing increasingly large amounts of data. Cost-efficiency and elasticity of data storage consequently have become a key requirement on storage solutions, giving rise to the development of NoSQL (Not Only SQL) data stores in the Cloud. Offerings include simple key-value stores such as Amazon S3¹ and Amazon SimpleDB², and

¹aws.amazon.com/s3

²aws.amazon.com/simpledb

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

MW4SOC '11, December 12, 2011, Lisboa, Portugal

Copyright 2011 ACM 978-1-4503-1067-3/11/12 ...\$10.00.

other schema-less offerings such as the Google App Engine datastore³ and Apache Cassandra⁴. Common to these diverse offerings is the creation and management of multiple geographically distributed replica of the data to be stored. A behind-the-scenes replication architecture is fundamental in ensuring high availability.

Cloud storage systems typically trade high availability against strong data consistency, and take advantage of very large numbers of commodity machines that fail frequently. Hence, NoSQL Cloud storage systems often exhibit eventually consistent [20] behavior. That is, a client may observe stale data for some time, and data consistency is only ensured eventually. Not all eventually consistent systems expose the same consistency characteristics, though. Tanenbaum and Steen [18], for instance, report on different non-strict classes of consistency guarantees that might be fulfilled by storage systems. As developing applications on top of an eventually consistent datastore requires a higher effort compared to traditional databases (if it is possible at all), as also pointed out by Wada et al. [21], any help in determining the actual consistency guarantees of a system is advantageous. Beyond the consistency classes, an immediate question then is: how soon (how late, respectively) is 'eventual' and is there actually a point in time where consistency is reached? In this paper, we report on experimental findings in the pursuit to answer this question. Knowing about consistency properties of a system can also help with the decision whether an application can use a particular datastore [14].

In section 2, we start by describing different perspectives on consistency. Next, in section 3 we provide an overview of our considerations on how to answer the question as stated above. Afterwards, in section 4 we experimentally validate the feasibility of our approach before evaluating the consistency behavior of Amazon S3 in section 5. After a brief discussion of our results in section 6 we report on related work before ending with a conclusion.

2. BACKGROUND

Literature, e.g., Tanenbaum and Steen [18], basically distinguishes two main classes of consistency: data-centric consistency and client-centric consistency.

Data-centric consistency models focus on the internal state of a storage system, i.e., consistency has been reached as soon as all replica of a given data item are identical. These

³code.google.com/appengine

⁴cassandra.apache.org

models come in slightly different flavors, ranging from traditional strict consistency, which requires all replica of all data items to be identical as well as all semantical relationships between data items to be observed, to consistency guarantees which can be found in systems like the Google File System [8], where replica are treated as consistent once every copy includes every single update *at least* once.

On the other hand, there are client-centric consistency models that do not care about the internal state of a storage system. Instead they focus on the consistency guarantees which can actually be observed by one or more clients, e.g., whether stale data is returned or not.

In consequence, when measuring how soon eventual consistency is (that is, measuring the length of the inconsistency window), there are again two different perspectives on this. A Cloud storage provider or anyone with access to the source code of a storage system (in the following just *provider*) would rather focus on a data-centric perspective. To a consumer, in contrast, it really does not matter whether internally a Cloud storage system contains a huge number of stale replica as long as the provider has implemented mechanisms to deal with those. As long as no stale data is observed, the customer is satisfied.

3. APPROACH AND IMPLEMENTATION

Measuring the length of the inconsistency window is trivial from a provider perspective: By adding detailed logging or notification functionality to the storage system it is easily possible to have the actual timestamps of each replica update readily available. By calculating the difference between the latest and the first timestamp it is, hence, possible to get the desired result.

From a customer perspective, in contrast, who might only have black-box access to the storage system (e.g., in case of Cloud storage systems) or might not have the means or knowledge to change the source code of a storage system, it is more important to know how long it takes from issuing an update to being able to still read the old version. For eventually consistent storage systems, this is typically a value expected to be greater than zero. This value can be experimentally determined by the following steps:

1. Create a timestamp.
2. Write a version number to the storage system.
3. Continuously read until the old version number is no longer returned, then create a new timestamp.
4. Calculate the difference between the write timestamp and the second timestamp (time of the last read of the previous version).
5. Repeat these steps to achieve statistical significance.

Depending on the latency of step 2, an alternative approach might create another timestamp between steps 2 and 3 and use the mean of those for the calculation in step 5 (instead of the timestamp from step 1).

Please, note that it is necessary to use the last read of the old version and not the first read of the new version as – for systems where monotonic read consistency⁵ is violated

⁵Monotonic read consistency is defined as follows: After having returned version n to a specific client the system guarantees to return only versions $\geq n$ [20]

– the timestamp of the last read of the old version may be long after the timestamp of the first read of the new version. Our system identifies the last read of a particular version using an internal buffer: for every version each reader remembers the last time it could read that version. Once the buffer is full, the inconsistency window is calculated for the oldest version only, before it is removed from the buffer. For our experiments we have chosen combinations of buffer size and write interval which guarantee that the highest observed inconsistency window easily fits into the buffer, i.e., $bufferSize * writeInterval \gg maxInconsistencyWindow$. E.g. a peak inconsistency window of about 33s combined with a configuration which enables us to capture values as large as 100s.

Independent from our work, Wada et al. [21] propose a very similar approach, already with interesting results. In our opinion, their approach has a fundamental flaw, though: only one reader is used in their experimental setup. By using only one reader, especially when running in the same datacenter as the writer or even worse running on the same machine, it is improbable to actually discover staleness. This is due to two facts:

1. A distributed storage system usually uses some kind of load balancer. Depending on the intelligence of the load balancer it is not unlikely that all requests from the same IP range are forwarded to the same replica or that there is even a caching layer in between. Accuracy can be greatly increased by running additional geographically distributed readers.
2. One reader can, depending on the latency L of the storage system, only achieve a resolution of $1/L$, i.e., send only $1/L$ requests per unit of time. Anything that happens in between is unknown. This resolution of the results can be almost linearly improved by adding more reader instances.

For these reasons, we have implemented a system where one writer periodically writes a local timestamp plus a version number to the storage system. Next, there is a number of readers (the actual number depends on the storage system) which are geographically distributed. These reader instances continuously poll the storage system and remember for each version the latest point in time where they could still read that specific version. After collecting this data from all readers, we then consider the difference between the latest read timestamp of version n and the write timestamp of version $n + 1$. This is, because the client-observable inconsistency window is the period of time after submitting an update where it is still possible to read the previous version.

Figure 1 shows an example which shall serve to better explain how we derive our results. The data used is not real data as we usually have about 1,000 reads in between two writes. We have observed similar logs in real monitoring data, though. In this example, the storage system violates monotonic read consistency.

The figure shows a timeline in the left column, the data the writer wrote in the second column, and what the two readers read at different points in time in the other two columns. Based on the highlighted last reads for a given version it is then possible to calculate the table in the right part of the figure.

For example, after 5 units of time (TU) the writer writes version B to the storage system. Reader 1 reads the old

Time	Writer	Reader 1	Reader 2	Version	Observed Inconsistency Window
0	(A,0)	-	-	A	$\max\{3;5\} = 5$
1		(A,0)	(A,0)	B	$\max\{2;4\} = 4$
2		(A,0)	(A,0)
3		(A,0)	(A,0)		
4		(A,0)	(A,0)		
5	(B,5)	(A,0)	(B,5)		
6		(A,0)	(B,5)		
7		(B,5)	(B,5)		
8		(A,0) ← 3	(B,5)		
9		(B,5)	(B,5)		
10	(C,10)	(B,5)	(A,0) ← 5		
11		(C,10)	(C,10)		
12		(B,5) ← 2	(C,10)		
13		(C,10)	(C,10)		
14		(C,10)	(B,5) ← 4		
...			

Figure 1: Consistency Monitoring Example

version *A* the last time after 8 TU, reader 2 does so after 10 TU. So, reader 1 observed that version *A* lingered on for 3 TU while reader 2 measured 5 TU. Both send their respective values to the collector. Since only the longest difference matters, this results in an inconsistency window of 5 TU for version *A*.

4. FEASIBILITY OF THE APPROACH

As already discussed in the previous sections, there are two general perspectives on consistency: data-centric and client-centric. Furthermore, the accuracy of our client-centric consistency monitoring highly depends on the number of readers used. This becomes also clear when looking at probabilities. For example, in a scenario with three replica where a load balancer forwards requests to a randomly chosen replica (uniformly distributed) we assume that all readers poll only once. Then, there is an 83% probability of reading from all three replica when using seven readers. When using nine readers there is a 93% probability. This in turn implies that the client-observable inconsistency window has an upper bound in the data-centric inconsistency window since there is always a chance of missing one replica, i.e., the probability of reading all replica will always be less than 100%. Furthermore, as already discussed, adding more readers increases the resolution of the results. Hence, using our approach from the previous section will always underestimate the actual client-observable inconsistency window. But how far off are these estimates from the actual data-centric consistency?

To address this question, we implemented a very simple distributed key-value store called MiniStorage with a replication level of three. In MiniStorage, a read request is served by just one replica. A write request, in contrast, is sent to one replica which persists the data locally or in memory. Next, it responds to the requester. After this, we added an artificial 1000ms delay⁶ before forwarding the write request to the other two replica. This corresponds to an (N,R,W) configuration of (3,1,1) [20]. Each MiniStorage replica logs the exact point in time when it executes an update request plus the content of that update.

For our evaluation, we deployed MiniStorage on three Amazon EC2⁷ small instances within the region us-east. At

⁶Originally, observed system latencies and inconsistency windows were close to the accuracy range of NTP which we use for clock synchronization so that no meaningful results could be achieved.

⁷aws.amazon.com/ec2

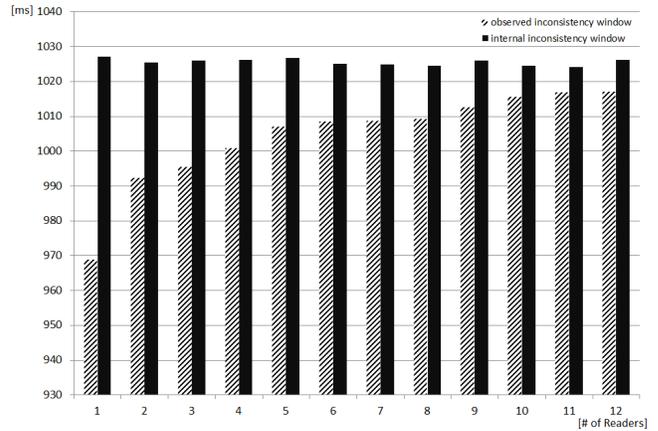


Figure 2: Data-centric vs. Client-observable Inconsistency Windows in MiniStorage

the time of our experiments this region offered four so-called availability zones which are each independent datacenters located in close proximity to each other. We could not start instances in availability zone 1a as it was full, so we distributed the replica over the zones 1b, 1c and 1d. Next, we deployed our consistency monitoring tool (again using only EC2 small instances). Our writer, the so-called collector (which is just responsible for collecting logs from the readers and the writer) and the first reader were deployed in region 1b as well and we started the test with an update interval of 5s and a poll interval of 10ms. Afterwards, we added additional readers every 10 minutes. While the first reader was in zone 1b, the second was in 1c, the third in 1d, the fourth in 1b again, and so on. We did this until we had 12 readers running, which is when we stopped adding readers but kept the system running for another two hours. This test was run on August 17, 2011.

The test results show a fairly stable inconsistency window of slightly above one second based on the MiniStorage logs. Our consistency monitoring instead slowly approaches that curve asymptotically which proves the validity of our considerations from sections 2 and 3. Figure 2 shows our results; the black bar stands for the inconsistency window calculated from MiniStorage logs while the striped bar shows our measured results by adding more and more readers. To remove small random fluctuations the figure only shows the mean values for each period between changing the number of readers. The actual values nevertheless also showed that the observed inconsistency window never exceeds the data-centric inconsistency window. It also shows that after a certain number of readers it becomes highly inefficient to achieve higher accuracy.

5. EVALUATION OF AMAZON S3

The first actual Cloud storage service which we evaluated via our consistency monitoring was Amazon’s Simple Storage Service (S3). S3 is a key-value store guaranteeing eventual consistency. Files are placed in buckets for which a location can be chosen from a set of regions which is identical to the regions of Amazon EC2. Also, files are replicated in multiple availability zones. It, hence, seems to be a valid assumption that S3 uses the same data centers as its EC2

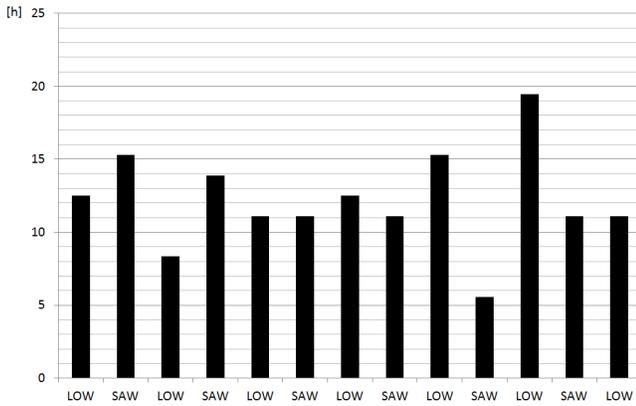


Figure 3: Length of LOW and SAW Periods over Time on S3

counterparts.

For our purposes we placed a bucket in the region eu-west (Ireland) since we had, during our MiniStorage tests, observed that we could not start EC2 instances in us-east 1a whereas we could start instances in all availability zones of eu-west. When we repeated our MiniStorage test for S3 starting additional readers in certain intervals, we observed that our results were fairly constant beyond 8 readers. To nevertheless play it safe, we deployed 12 readers – 4 per availability zone. Our writer as well as the collector were deployed in zone a, all instances again were small instances. We chose an update interval of 10s to give each update enough time (in our mind) to propagate without interfering with older updates. The poll interval per reader was set to 10ms. We started the test on August 29, 2011 8.30h AM (UTC) and kept it running for a week.

In contrast to the findings of Wada et al. [21] who could not observe any inconsistencies at all, and in contrast to our expectations of seeing a normal distribution of inconsistency window lengths, our results show some strange periodicities. First, there is a long-term periodicity: Roughly every 12 hours the behavior of S3 abruptly changes between what we will call a LOW phase and a SAW phase. Figure 3 shows the length of those periods in comparison.

During the LOW phase we actually find a random⁸ distribution with a mean value of 28ms and a median of 15ms. Please, note that these values may be exact but could be off by at least a factor 2 due to the accuracy limitations of NTP [15] which we use for clock synchronization. We believe, though, that median and mean values between 0 and 100ms are realistic.

During our SAW periods we can observe a curve which resembles a sawtooth wave – hence, the name. It really does not matter which SAW phase we select an excerpt from, the periodicity follows always the same pattern: First, the inconsistency window’s length is close to zero. Then, it increases by about one or two seconds with every test until it peaks at about eleven seconds before dropping straight down to the next minimum. The only difference that can be found is that the minimum can be found in the interval between zero and five seconds and the maximum can be found

⁸The distribution has three local maxima: the absolute maximum at 7ms, next smaller local maximum at 26ms and another small local maximum at 90ms.

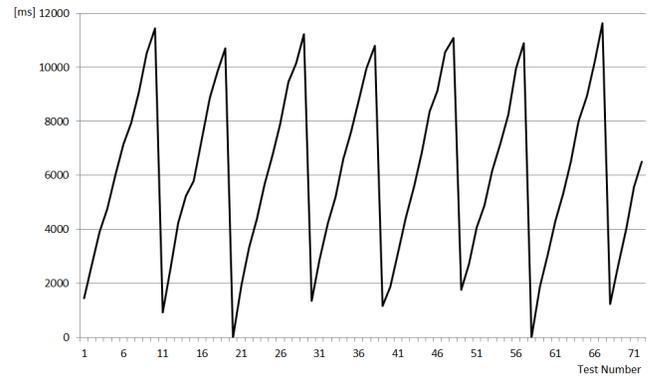


Figure 4: Observed Inconsistency Window Length during SAW Periods Over Time on S3 (Excerpt)

between ten and twelve seconds. The wavelength of this pattern fluctuates between eight and twelve tests, i.e., for our test setup the pattern restarts every 80 to 120s. Figure 4 shows an excerpt from one of the SAW phases.

We have been researching the question of consistency monitoring for quite a while now. Repeated tests on S3 showed the exact same results. Already in July and August 2010, we experimentally analyzed consistency guarantees of S3 via an independent implementation which also used a slightly different algorithm. Even back then (where it was only a by-product of our evaluation of [3]) we observed remarkably similar behavior. Figure 5 shows the full results of our one week evaluation of S3. Due to the sheer number of test runs and, hence, the density of the curve, it is not possible to see the sawtooth pattern during the SAW phases but it is still easily possible to distinguish SAW and LOW phases.

Another finding was that the availability zones seem different in terms of accessing the latest version. While our writer was in zone a, the longest inconsistency window length was observed in 28% of all tests in zone a. The same is true for zone c while zone b had the maximum in 49% of all tests⁹. This indicates that zone b seems to have a slightly poorer connection to the other two zones, e.g., by being located in a different building.

Furthermore, regarding locations we could not see differences between the zones: They all did the same sawtooth wave and had their maxima and minima at the exact same time only the amplitudes were slightly different which creates the results from the last paragraph.

We also tested our results for violations of monotonic read consistency. From a total of 353,357,884 reads 42,565,840 or about 12% of all requests violated monotonic read consistency [20]. In exchange, we observed an availability of more than eight nines (99.999997% – only one request returned an error).

6. DISCUSSION

In summary, we observed an unexpected, very interesting consistency behavior of Amazon S3, but have so far not been able to come up with a satisfying explanation of our experimental findings. Possible explanations could be caching effects or measurements to counter DDoS attacks

⁹The total is not 100% as for about 5% of all tests two zones observed the same inconsistency window.

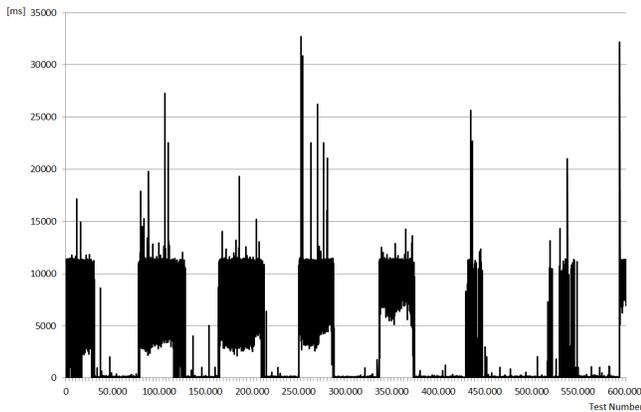


Figure 5: Observed Inconsistency Window Length Over Time on S3

(which both would not explain why there are LOW phases) or internal rebalancing processes which are triggered about every twelve hours and somehow cause this phenomenon. The latter would explain the SAW vs. LOW periodicities but still does not explain why the SAW phase comes in such a shape.

Another possible explanation for the periodicities could be the NTP protocol (and specifically the `ntpd` linux demon) which we use for clock synchronization. Our approach naturally relies on a tight synchronization of all monitoring clocks. If the clocks continuously drift apart a resynchronization about once per minute may occur; the result may then be a behavior as shown in figure 4. We believe, though, that this is not the case for several reasons:

1. Before running our experiments we tested the accuracy achieved by `ntpd`. This was done by opening `ssh` connections to several EC2 instances which had `ntpd` running. Those instances then continuously printed their local timestamp every second. All values which we could observe were less than one second apart so that delays of about 12s can not be explained.
2. If NTP were the root cause of the periodicities it would still only explain the SAW but not the LOW phases. Hence, the behavior of `ntpd` would need to change completely every 12 hours which seems highly unlikely.
3. Preliminary results of experiments with Apache Cassandra (with or without additional load) as well as our MiniStorage experiments do not show any periodicities at all. Instead Apache Cassandra seems to follow a geometric distribution and MiniStorage shows the random distribution already mentioned. If we monitor two files on S3 at the same time the second file shows an entirely different behavior while the first file behaves as discussed above. These results were achieved using the exact same configuration running `ntpd`.

Still, while NTP effects cannot explain the periodicities it heavily affects the accuracy of our measurements. For future versions we will, hence, also look at alternative clock synchronization approaches, e.g., the coupling-based algorithm by Baldoni et al. [2].

Of course, we also considered that some parts of our implementation might have caused this behavior. For this reason

we had several people cross-check our approach as well as the source code. Also, we could not observe a similar behavior when benchmarking other storage systems and our approach includes no long-term periodicities which could explain the change between LOW and SAW phases. Furthermore, an independent earlier prototype using even a slightly different approach created similar results. For these reasons, we believe that our results reveal an interesting behavior of S3 which is caused by Amazon’s internal design choices. In the end all interpretation on our side is guesswork so that the final explanations need to be provided by Amazon. Still, we are looking forward to discussing our findings at the conference which might bring up other possible explanations.

7. RELATED WORK

There is a huge number of publications on distributed storage systems with only eventually consistent guarantees. Examples among others are [6, 13, 8, 4, 12, 17, 19].

Also there is work on benchmarking those data stores [5, 23, 10]. Some of those even call for the necessity of consistency monitoring in their future works part, but to our knowledge so far only [21, 1] actually evaluate consistency guarantees. While the results of Wada et al. [21] show the implications of *not* using multiple readers their approach is otherwise very similar.

Anderson et al. [1] in contrast require detailed operation logs. After execution they run an offline check to analyze whether consistency violations have occurred. Due to the complexity of their calculations it is impossible to provide live monitoring. Furthermore, it is not clear how their approach can be integrated into a concrete application and the results highly depend on the application workload or interaction pattern with the datastore that is used. Their results are, hence, more application-specific than datastore-specific.

Klems et al. [9] also propose consistency benchmarks but their approach using Fox and Brewer’s harvest and yield metrics [7] does not consider staleness of results, rather measuring availability and completeness of answers of a distributed queueing system.

Consistency Rationing like [16, 11] or ongoing research within our research group towards the same question requires means to tune consistency guarantees. For this purpose, it is necessary to be able to measure the actual consistency output. Our work can, hence, serve as input for those tools.

8. CONCLUSION

In this paper we started by discussing different perspectives on consistency of distributed storage systems, namely a provider (or data-centric) and a consumer (or client-centric) view. We then explained how these two relate to each other and introduced an approach which allows to measure the staleness of data, or how soon ‘eventual’ in eventual consistency is. Also, we compared it to a similar approach, independently developed by Wada et al. [21], and showed how our design corrects a fundamental flaw of their approach. Afterwards, we validated our approach using a simple key-value store called MiniStorage.

After these initial considerations, we evaluated Amazon S3 in terms of consistency guarantees and found, in stark contrast to the findings by Wada et al. [21], that S3 frequently violates monotonic read consistency. Also, we en-

countered strange periodicities, namely our so-called SAW and LOW phases which alternate approximately twice a day. Furthermore, we described the sawtooth wave-like behavior of S3 during SAW phases before discussing potential explanations.

Our approach of geographically distributed readers combined with a writer fits into current research regarding benchmarking of distributed datastores as well as systems building on top of that. Our results provide concrete data that serves as criteria for an application developer to determine whether an eventual consistency data store provides acceptable consistency guarantees.

In future endeavors, we will try to determine dependencies between files on S3, e.g., how periodicities of files within the same bucket or across multiple buckets correlate. Furthermore, we are currently benchmarking Apache Cassandra and the Google App Engine datastore. We plan to publish these results as well as to extend our efforts to additional storage systems in a follow-up paper.

Finally, Yu and Vahdat [22] as well as similar models know other consistency dimensions beyond staleness, e.g., order error. We are investigating means to also measure these dimensions.

9. REFERENCES

- [1] E. Anderson, X. Li, M. Shah, J. Tucek, and J. Wylie. What consistency does your key-value store actually provide. In *Proceedings of the Sixth Workshop on Hot Topics in System Dependability (HotDep)*, 2010.
- [2] R. Baldoni, A. Corsaro, L. Querzoni, S. Scipioni, and S. Tucci-Piergiovanni. An adaptive coupling-based algorithm for internal clock synchronization of large scale dynamic systems. In *Proceedings of the 2007 OTM Confederated international conference on On the move to meaningful internet systems-Volume Part I*, pages 701–716. Springer-Verlag, 2007.
- [3] D. Bermbach, M. Klems, M. Menzel, and S. Tai. Metastorage: A federated cloud storage system to manage consistency-latency tradeoffs. In *Proceedings of the 4th International Conference on Cloud Computing (IEEE Cloud 2011)*. IEEE, 2011.
- [4] B. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!’s hosted data serving platform. *Proceedings of the VLDB Endowment*, 1(2):1277–1288, 2008.
- [5] B. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.
- [6] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Voshall, and W. Vogels. Dynamo: amazon’s highly available key-value store. In *Proc. SOSP*, 2007.
- [7] A. Fox and E. Brewer. Harvest, yield, and scalable tolerant systems. In *Proceedings of the 7th Workshop on Hot Topics in Operating Systems, 1999*, pages 174–178. IEEE, 2002.
- [8] S. Ghemawat, H. Gobioff, and S. Leung. The Google file system. *ACM SIGOPS Operating Systems Review*, 37(5):29–43, 2003.
- [9] M. Klems, M. Menzel, and R. Fischer. Consistency benchmarking: Evaluating the consistency behavior of middleware services in the cloud. In *Proceedings of the 8th International Conference on Service Oriented Computing (ICSOC)*. Springer, Dec. 2010.
- [10] D. Kossmann, T. Kraska, and S. Loesing. An evaluation of alternative architectures for transaction processing in the cloud. In *Proceedings of the 2010 international conference on Management of data*, pages 579–590. ACM, 2010.
- [11] T. Kraska, M. Hentschel, G. Alonso, and D. Kossmann. Consistency Rationing in the Cloud: Pay only when it matters. *Proceedings of the VLDB Endowment*, 2(1):253–264, 2009.
- [12] J. Kubiawicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, C. Wells, et al. Oceanstore: An architecture for global-scale persistent storage. *ACM SIGARCH Computer Architecture News*, 28(5):190–201, 2000.
- [13] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.
- [14] M. Menzel, M. Schoenherr, and S. Tai. (mc2)2: criteria, requirements and a software prototype for cloud infrastructure decisions. *Software: Practice and Experience*, 2011.
- [15] ntp.org. *NTP Algorithm*. <http://www.ntp.org/ntpfaq/NTP-s-algo.htm> (accessed on September 6, 2011).
- [16] S. Sakr, L. Zhao, H. Wada, and A. Liu. Clouddb autoadmin: Towards a truly elastic cloud-based data store. In *The 9th IEEE International Conference on Web Services (ICWS 2011)*, Washington DC, USA, July 2011.
- [17] M. Satyanarayanan, J. Kistler, P. Kumar, M. Okasaki, E. Siegel, and D. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on computers*, pages 447–459, 1990.
- [18] A. S. Tanenbaum and M. V. Steen. *Distributed Systems - Principles and Paradigms*. Pearson Education, Upper Saddle River, NJ, 2nd edition, 2007.
- [19] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. *ACM SIGOPS Operating Systems Review*, 29(5):172–182, 1995.
- [20] W. Vogels. Eventually consistent. *Queue*, 6:14–19, October 2008.
- [21] H. Wada, A. Fekete, L. Zhao, K. Lee, and A. Liu. Data consistency properties and the trade offs in commercial cloud storages: the consumers’ perspective. In *5th biennial Conference on Innovative Data Systems Research, CIDR*, volume 11, 2011.
- [22] H. Yu and A. Vahdat. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Transactions on Computer Systems (TOCS)*, 20(3):239–282, 2002.
- [23] L. Zhao, A. Liu, and J. Keung. Evaluating cloud platform architecture with the care framework. In *2010 Asia Pacific Software Engineering Conference*, pages 60–69. IEEE, 2010.