Adaptive Hybrid Quorums in Practical Settings

Aaron Davidson, Aviad Rubinstein, Anirudh Todi, Peter Bailis, and Shivaram Venkataraman

December 17, 2012

Abstract

A series of recent works explore the trade-off between operation latency and data consistency in replicated data stores. In this paper, we build on the improved understandings of this trade-off and provide a new adaptive algorithm that modifies the behaviour of a replicated data store online. We use exploit consistency prediction techniques[3] available in Cassandra and implement the dynamic reconfiguration of the quorum sizes R and W. Furthermore, we extend the existing ideal model for the consistency-latency trade-off to include the case of a node failure. We demonstrate how our dynamic reconfiguration of the quorum sizes can be used to enhance the system performance (over 30% decrease in latency) with only a minor sacrifice in consistency.

1 Introduction

Traditionally, data stores use quorum systems to define a set of replicas for each item. However, several modern Dynamo-based data stores use quorums *within* a set of replicas to define weak semantics of eventually consistent transactions. In particular, every item is guaranteed to eventually be written to *all* replicas. This implies that the choice of quorum system is only in effect for the (typically) short period of time until an write has reached all replicas.

In this work, we exploit the transient nature of quorum systems in Dynamo-like data stores and dynamically modify them. We propose a *practical* algorithm that adaptively reconfigures quorum sizes based on the varying conditions of the system, including workload, online observed performance, and failures. In order to account for node failure we extend the existing WARS model [3] analytically and show the accuracy of our predictions in simulations. Finally we demonstrate how our dynamic reconfiguration of the quorum sizes can be used to enhance the system performance (over 30% decrease in latency) with only a minor sacrifice in consistency.

1.1 Quorum system

Many data store systems replicate data in quorums. Traditionally, a *quorum system* is a set of subsets of the replicas, chosen so that every two *quorums* have a non-empty intersection.

Most of the earliest works on quorum systems (e.g. [7]) defined quorums as each having more than half of the replicas, therefore easily guaranteeing that every two have quorums have a common replica; these quorum systems are sometimes also called vote quorums. Later works defined more complex quorums systems based for example on finite projective planes [10], trees [2], and paths on grids [13]. These systems achieve theoretical properties such as smaller quorums, capacity and (theoretical) availability [13]. However they are mostly impractical to implement with current technology and vote quorums and their variations are more often used in practice.

In systems where read and write operations are separate, one may consider a system of distinct *read quorums* and *write quorums*. In such a system every pair of a write quorum and a read quorum must have a common replica; thus any write that goes to all the replicas in a write quorum must be stored on one of the replicas in any read quorum. Read and write quorum systems maintain good durability and near-perfect consistency, while significantly improv-

ing availability and reducing the load on the system compared to writing and reading every data item to and from all replicas.

1.2 Partial quorums

Naor and Wool proved [13] that full, traditional quorum system cannot achieve both high capacity and low load. This motivated Malkhi et al. [11] to introduce the concept of *partial quorums* (also called "probabilistic quorums"). Partial quorum system does not satisfy the intersection condition of full quorums. Instead, it provides a probabilistic guarantee that if two quorums are chosen according to some specified probability distribution, they will intersect except with a small probability ϵ.

Partial quorums can achieve important properties that make them more practical such as lower load (which intuitively translates to smaller quorums). Additionally, the design of partial quorums tends to be much simpler and is commonly simply all the subsets larger than some threshold value. Thanks to the Birthday Paradox it suffices to take a threshold of $O(\sqrt{n})$ in order to get an arbitrarily small constant error probability ϵ.

1.3 Eventual consistency

Motivated by Brewer's celebrated CAP Theorem [4], researchers looked throughout the previous decade for appropriate alternatives to consistency. One such form of weak consistency is *eventual consistency*, which is the guarantee that if no further updates occur, *eventually* the storage will become consistent [16].

While many applications can tolerate a small delay in which a newly updated value may be inconsistent, eventual consistency per se does not provide any bound on this delay. A line of recent works such as [3, 15, 5] attempts to measure and even predict the time it will take the data to become consistent.

1.4 Cassandra

We chose to implement our work in Apache Cassandra, which is a Dynamo-based distributed data store. Cassandra is an ideal playground for academic work because it is open source and relatively simple; for those reasons it is also gaining popularity in production.

Similar to Dynamo, in Cassandra each key is randomly assigned to N replicas. Every write to the key is sent to all N replicas; however, after the coordinator issuing the write receives acknowledgements from W replicas, it sends an acknowledgement to the client. The rest of the N replicas receive the write asynchronously in the background. We will consider a read to be "consistent" only if it returns the latest value for which W acknowledgements have been received, or a later value.

Unlike in Dynamo, when issuing a read each coordinator only sends requests to R nodes. When all R nodes return their responses, the coordinator returns the latest value to the client. Another important deviation from Dynamo is that in Cassandra each key has only a single timestamp, rather than a clock-vector.

2 Related Works

There have been several theoretical works about dynamically reconfiguring quorums. Abraham and Malkhi [1] extended the design of probabilistic quorums from [11] to quorums that can expand or shrink over time. Naor and Weider [12] investigated the theoretical asymptotic complexity of finding a quorum in a setting where nodes may temporarily fail. leave, or join the system. Gramoli and Raynal [8] described "Time Quorum Systems", where two quorums are guaranteed to intersect with high probability if they are accessed within a short time.

Noble et al. [14] introduced "fluid replication", a theoretical method to adaptively choose the replication nodes. Ishikawa et al. [9] discuss dynamically changing quorums in the context of power grids. To the best of our knowledge we are the first paper that considers adaptively modifying quorums in practical data stores settings.

3 Fundamentals

3.1 Smaller quorum sizes (R and W): lower consistency and better performance

We note that the smaller we make W (number of write responses required), the smaller the number of nodes that are guaranteed to have the latest write. The writes will be asynchronously propagated to all the nodes and will *eventually* reach all the replicas. However, only a subset W is guaranteed to have the write. This implies that when we read from R replicas, we are less likely to read from a replica that has the latest write, thereby reading stale data. Moreover, by reducing R, we further reduce the number of nodes from which we read, thereby further reducing the chances of reading up to date data, and hence further reducing our consistency.

While reducing R and W may reduce our probability of consistency, it increases the performance of our system. This can be illustrated by a simple example: If our original value for W was 2, this meant that we would send the write request to all N replicas and block until we received a write acknowledgements from any 2 of the N replicas. If we now modify our W to be 1, we only block until we receive a write acknowledgement from 1 of the N replicas. Blocking on fewer acknowledgements implies that we spend less time waiting and mark the write as "committed" sooner, thus decreasing the write latency and increasing performance. A similar reasoning can be applied to show why reducing R improves performance.

3.2 Preserving durability

We note that by modifying the values of R and W, without changing N, the number of replicas, we have little or no effect on durability. Information can be lost during a write only in the extremely unlikely event that all of the following happen for the same transaction: (1) the coordinator receives acknowledgements from W nodes; (2) all W nodes suffer an unrecoverable failure; (3) all other $N - W$ messages are dropped; *and* (4) the coordinator also suffers an unrecoverable failure before realizing the other $N-W$ messages were dropped, yet after sending an acknowledgements to the client.

4 Adaptive Consistency

We propose a new algorithm that *adaptively* modifies the Consistently Level parameters to achieve significantly improved consistency and performance.

In particular, our algorithm can incorporate: (1) a user specified desired consistency level; (2) observed quality of the current workload, including latency distributions and reads vs writes distribution; and (3) failures of nodes in the cluster (as discussed in section 5).

4.1 Achieving the desired consistency level

Bailis et al. [3] proposed the WARS model for online prediction of consistency levels based on observed latency distributions. They also implemented a consistency predictor, PBS Predictor which predicts the probability that a read request will observe the data from a write request issued t ms earlier. Furthermore, the consistency predictor can output relatively accurate predictions for quorum settings other than the setting currently used (a "what if" analysis).

Our algorithm can accept as input a desired consistency level, and, using the consistency predictor, it finds the minimal quorum setting which achieves this consistency level. For example, consider a system that is in a configuration of $(R = 1, W = 2)$, and a desired consistency level of 99.9%. Suppose a large job just ended and the load on the system is reduced. The consistency predictor will observe the change in latencies and predict that a configuration of $(R = 1, W = 1)$ can now achieve a consistency level of 99.93%. Our algorithm will reduce the size of subsequent write quorums, thereby reducing latencies.

4.2 Finding the optimal configuration for current conditions

In addition to guaranteeing a desired consistency level, our algorithm also finds the optimal configu-

ration for the current workload and state of the network. Each node keeps track of the ratio of recent write request to read request. In particular our workload characterization χ variable is updated at each request according to:

$$\chi^{(t+1)} = \begin{cases} \gamma\chi^{(t)} + (1-\gamma) & \text{if request is a write} \\ \gamma\chi^{(t)} & \text{if request is a read} \end{cases}$$

where γ is a parameter set by default to 0.99. Using this measurement, if a workload has mostly writes χ will approach 1, whereas if the workload is read oriented χ will approach 0. Another important property of this estimator is that recent requests have a greater influence than older requests, thus giving an accurate description of the *current* workload on the system. Finally, the choice of γ can be thought of, intuitively, as giving most of the weight to the last $(1-\gamma)^{-1}$ requests (i.e. the last 100 request with the default setting of $\gamma = 0.99$).

In addition to the observed workload on the system, we have measurements of the recent latencies in the system, for each type of request. We can now aggregate this information, together with the consistency probability predictions to find which of the configurations that satisfy the consistency requirements gives optimal performance[1]. Let us go back to our example of a system in a configuration of $(R = 1, W = 2)$, and a desired consistency level of 99.9%. Suppose that the workload trend changed and the proportions of write requests increased from 10% to 30%; additionally, we observe that write latencies are significantly higher. Our algorithm will automatically switch from to a configuration of $(R = 2, W = 1)$. This will cause a small increase in the read latencies but will enable more influential latency-savings on the write latencies.

4.3 Handling failures

Handling failures of servers is a problem of growing importance as network sizes continue to scale. In Section 5 we propose a new model for accounting for node failures and analyze their effect on consistency. Our algorithm incorporates this analysis to achieve more accurate predictions of consistency probabilities, thereby improving the way our system handles failures.

In particular our analysis of node failures shows the surprising result that node failure *improves* consistency. Therefore, we can often reduce the quorum requirements in the case of a node failure. This is especially important because we reduce the latencies in a critical time when the load on the system is divided among less replicas.

4.4 Not only latencies

As discussed in the previous sections, our algorithm can improve latencies in several circumstances by reducing the quorum sizes. We note that while this is the main advantage of our algorithm, it can also improve other performance parameters.

In Dynamo, read and write requests are sent to all N replicas. The coordinator then waits for R or W responses (depending on the type of request) before continuing. However, in Cassandra only write requests are sent to all N replicas. Read requests, on the other hand, are sent only R replicas in first place. Thus in any of the cases where our algorithm reduces the read quorum requirement in the system, Cassandra will actually send less messages, thereby reducing the load on the system, and further improving the efficiency and throughput of the system.

4.5 Time vs space adaptiveness

Throughout this section, we described our algorithm as incorporating *recent* data on the performance and workload in the system. Thus our algorithm will adapt well to changes in those parameters *over time*.

We would like to note that in addition to *time adaptiveness*, our optimization algorithm also provides *space adaptiveness*. Our algorithm is run locally on each coordinator, according to the statistics observed locally by it. Therefore we will adapt the system optimally to the *current and local* conditions. For example, if one cluster has slower writes we may increase the quorum requirements locally for

[1] Optimal performance can be configured to provide either optimal mean latency, or optimal p^{th}-percentile latency.

that cluster, while keeping latencies low for the rest of the network.

5 Failure Modeling

As networks continue to grow, the probability of one server out of thousands failing increases. Additionally, as the industry trend towards using more cheaper and less reliable components continues the problem of node failing becomes ever more acute.

In this section we provide a simple three-phase model of how Cassandra behaves when a node fails ("fail-stop"), and analyze the effect on consistency and performance of the system: (1) the initial node's failure; (2) Cassandra discovers the node failures and begins to send writes to a *hinted* replica; (3) the failed node is fixed, and Cassandra initiates recovery by *hinted-handoff*. We will denote "Phase 0" as the base case where all the nodes are running.

5.1 Phase 2: writing to a hinted node

Eschewing chronological order, we begin with the analysis of Phase 2, which is the simplest phase to analyze, and also the most important.

In Phase 2, Dynamo discovers via local "gossip" that a node has failed. Writes are sent to a hinted replica, but the acknowledgements from a hinted replica do not count for the quorum requirements. Additionally, the hinted replica does not answer read requests. Thus, for consistency estimation purposes, this situation corresponds to having only $N-1$ replicas. In this case, the probability of consistency is given by:

$$P^A_{(2,N,R,W)} = P_{(0,N-1,R,W)}$$

The situation may seem a bit more complicated when trying to read a value that was written before the node failed. In this case there is a W/N probability that the failing node was one of the original W nodes to acknowledge that write, therefore leaving only $W-1$ live nodes that are guaranteed to have received the latest write. With probability $1-W/N$, this is not the case, i.e. the failing node did not acknowledge the original write, and therefore there are still W replicas that are guaranteed to have received the latest value.

A much simpler analysis of the case of an attempt to read a value written before the node fails, would be to notice that it is completely equivalent to a scenario where our pseudorandom hash function is broken, and the read request is always sent to R other nodes. Since we do not account in our model for the differences in consistency likelihood between nodes, it follows that this has exactly the same consistency probability as when all the nodes are alive.

Nevertheless, if we want the read to see a write that was issued before the node failed, we can assume it is at least as old as the time since the node failed. Therefore, t ms after the node failure, the probability of consistency for this case is given by:

$$P^B_{(2,N,R,W,t)} \quad = \quad P_{(0,N,R,W,t)}$$

When issuing a read, we cannot know whether the key in question was last written before or after the node failed, so we take the conservative estimate:

$$P_{(2,N,R,W,t)} = \min\left(P^A_{(2,N,R,W)}, P^B_{(2,N,R,W,t)}\right)$$

Finally, we plug this modified predicted consistency probability into the algorithm derived in Section 4. Notice that in both cases we expect *consistency to improve when a node fails!*

5.2 Phase 1: "what you don't know can't hurt you"

In Phase 1 of our model, a node crashed, but the other nodes have not discovered the problem yet. In this case it will neither receive nor acknowledge any write requests, and the coordinator will simply wait for acknowledgements from the other replicas. Moreover, the failed replica cannot reply to any read request, and the issuing coordinator will have to retry to read from another replica after waiting for a timeout.

Again, like in Phase 2, the system should behave like a system with $N-1$ replicas for writes that occur after the failure, and as usual for writes that occur before the failure. The only deviation from this

behaviour is that read requests that wait for timeout are even more likely to return a consistent value. Therefore consistency in Phase 1 may be even slightly better than in Phase 2. In particular, *consistency in Phase 1 is at least as good as the base case.*

Unfortunately, the coordinator of course cannot know that we are in Phase 1 - so we cannot take advantage of our analysis in this context.

5.3 Phase 3: recovery

In Phase 3, the server has been fixed and the node returns to the network. Cassandra initiates a recovery process using handoff of the hints from the hinted node. During this phase, consistency may be impaired by the recovering node attempting to answer requests for reads it has not yet recovered.

In more details, with probability $\frac{R}{N}$, a read request will be sent to the recovering node. In that case, only $R-1$ of the remaining $N-1$ nodes may have the most recent value - if it was written before Phase 3 began. Otherwise, with probability $1-\frac{R}{N}$, we again have R reads out of $N-1$ nodes that may have received the write. Again, this is only relevant if the write occured while the node was down, i.e. we can lower bound the time since such a write. Therefore, after t' ms from the beginning of Phase 3, the probability of consistency is given by

$$\begin{aligned} P^A_{(3,N,R,W,t')} &= \frac{R}{N} P_{(0,N-1,R-1,W,t)} \\ &\quad + \left(1-\frac{R}{N}\right) P_{(0,N-1,R,W,t)} \end{aligned}$$

Unfortunately, unlike in the analysis of Phase 2 in section 5.1, we cannot claim that this is the same as when all the nodes are running: When all the nodes are running, even if were guaranteed that the recovering node is not one of the first W nodes to acknowledge the write, it will receive it eventually; the recovering node, however can only receive it from the hinted node, which may take lower. In particular, note that this means that we expect consistency to be slightly worse during Phase 3.

The other situation, where one tries to read a value that was written during Phase 3, i.e. while all the nodes are alive, is trivially the case analyzed in [3] and computed by the PBS Predictor.

$$P^B_{(3,N,R,W)} = P_{(0,N,R,W)}$$

Finally, we again take the conservative, minimal estimate of the consistency probability:

$$P_{(3,N,R,W,t')} = \min\left(P^A_{(3,N,R,W)}, P^B_{(3,N,R,W,t')}\right)$$

We believe that this the observed consistency in this phase should not be significantly different from the base case where all nodes are running. Nonetheless, we hope to complete the implementation of this phase into the Adaptive Consistency algorithm in future work. It is interesting to note that this implementation may be more technically challenging because in order to incorporate Phase 3 into Adaptive Consistency one needs to keep track of completing of the hinted-handoff.

5.4 Implications

Our algorithm incorporates the analysis of Phase 2 to achieve more accurate predictions of consistency probabilities, thereby improving the way our system handles failures. In particular our analysis of node failures shows the surprising result that node failure *improves* consistency. Therefore, we can often reduce the quorum requirements in the case of a node failure. This is especially important because we reduce the latencies in a critical time when the load on the system is divided among less replicas.

6 Evaluation

We compare our predictions in the case of a node failure to those computed according to the ideal WARS model and show improved accuracy. We also measure latencies and consistency probabilities and demonstrate significantly improved performance using our adaptive reconfiguration of quorum sizes compared to the base case of current Cassandra trunk.

6.1 Implementation

We chose to implement our work in Apache Cassandra, which is a Dynamo-based distributed data store. Cassandra is an ideal playground for academic work because it is open source and relatively simple; for those reasons it is also gaining popularity in production.

We modified Cassandra-trunk to dynamically reconfigure the ConsistencyLevel property which specifies the quorum sizes R and W. We called the existing PBSPredictor() to generate consistency predictions and retrieve latency statistics. We adapted the consistency predictions from the PBSPredictor() to account for node failures according to our model in Section 5.

6.2 Evaluation methods

We ran all tests by setting up Cassandra nodes at different local IPs on a single machine. Each node was assigned an equal portion of the token ring. All queries were directed to the same test key and column to allow simple, predictable behavior from Cassandra. For our tests, we used 4 Cassandra nodes: 3 of the nodes owned the test key we used (since N=3), and we used the last as a "coordinator node." We sent all requests to the coordinator in order to ensure that none of the reads/writes were cached or local.

We utilized the Yahoo Cloud Serving Benchmark (YCSB) [6] to set up a workload for our system. YCSB was used in place of the standard Cassandra stress tool due to its greater configurability and functionality. We utilized a workload with 50% reads and 50% writes, interleaved. During our tests, we brought up the Cassandra nodes, waited for them to come alive, and then started YCSB. After 40 seconds, we killed one of the three nodes serving the test key. Another 40 seconds later, we restarted the node, and 40 seconds after that we ended the test.

One issue we noted through using YCSB was that it utilizes a fixed pool of threads to issue blocking requests. This meant that each thread has to wait for the request to complete for it to move on to the next request. Effectively, this meant that the test was self-repairing: If latencies increased dramatically, the test would slow down the requests to match, meaning that inconsistencies did not rise significantly. In order to circumvent this problem, we made all Cassandra requests return immediately, and continue the request in the background in a new thread. To prevent YCSB from overloading the system by creating thousands of threads, we additionally set the throttling parameters in YCSB to use 10 threads and only send 20 requests per second.

Latencies

We injected latencies according to the WARS model developed by Bailis et al. [3]. Each message requesting or acknowledging a write or read was independently assigned a random latency drawn from distribution measured on a real-world node in a LinkedIn service under peak traffic according to statistics provided by [3].

Write requests (W) latencies were drawn with 38% probability from a Pareto distribution with $x_m = 1.05, \alpha = 1.51$, and with 62% probability from an exponential distribution with $\lambda = 0.183$. Write acknowledgements (A) and read requests (R) and responses (S) were drawn with 91.22% probability from a Pareto with $x_m = 0.235, \alpha = 10$, and with 9.78% probability from exponential distribution with $\lambda = 1.66$.

6.3 Results

The plots in figures 1-4 are averages of multiple runs of the following scenario: a cluster of four nodes is initialized; at a specified point of time our simulator kills one of the nodes; then, after some time, the node is restarted.

Figures 1 and 2 show the base case, which is the current Cassandra-trunk (modified only to log the results), with quorum sizes configuration set to $(R = 1, W = 2)$. We can see (figure 1) that after approximately $60,000$ ms one node in the cluster is killed, and the proportion of observed inconsistencies decreases dramatically and even reaches 0; meanwhile

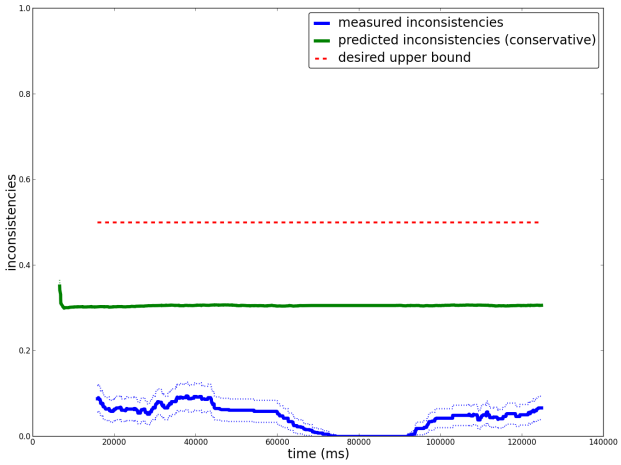

Figure 1: Inconsistency proportions for Cassandra-trunk. All our plots have lower and upper error bounds (± 3 standard deviations) in dotted lines; for some of the graphs (e.g. the predicted inconsistencies in this figure) the standard deviations are too small to be noticeable.

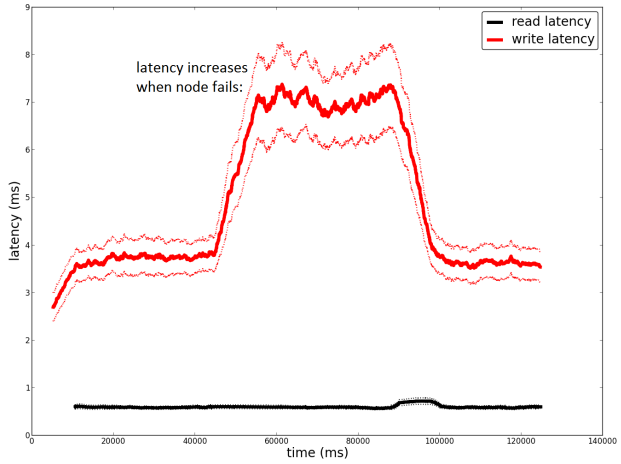

Figure 2: Average latencies for Cassandra-trunk

the unmodified PBS Predictor does not predict this decrease at all.

In figure 2 we notice that when a node is killed the latencies of write operations increases dramatically. We would like to minimize the impact of this slow down on the system in the case of a node failure.

Figures 3 and 4 show the modified Cassandra, which runs our adaptive consistency optimization algorithm. We can see in figure 3 that there is a slight decrease in inconsistencies immediately after the node is killed (Phase 1). After a short time (Phase 2) our algorithm automatically modifies the quorum sizes configuration from $(R = 1, W = 2)$ to $(R = 1, W = 1)$; the rate of inconsistency increases, but it is still below the designated inconsistency threshold. Finally, the node is restarted and the rate of inconsistencies return to normal.

Figure 4 demonstrates the importance of this work. Immediately after the node fails (Phase 1) there is a sharp increase in write latency. The system must not remain with this unacceptable performance for a long time. As soon as gossip detects that a node is down, the quorum sizes are decreased, and write latency become even better than during normal operation. This is especially helpful because it helps masking other slow downs in the system during a node failure.

Finally, we note that in both the base case and modified Cassandra the read latency remains unaffected because Cassandra, unlike Dynamo, only sends read requests to R (live) replicas - so those operations are practically unaffected by a fail of another node.

Quantitatively, for the entire time a node was down (Phase 1 and Phase 2), the average write latency for Cassandra-trunk was 7.06 ms (standard deviation 0.13 ms), while for modified Cassandra it was only 4.82 (standard deviation 0.11).

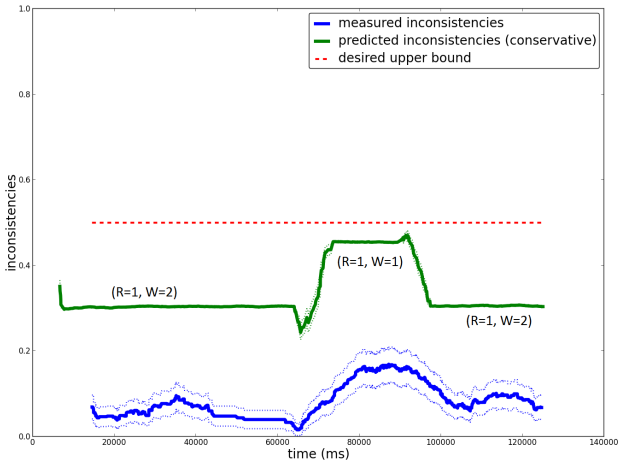

Figure 3: Inconsistency proportions for Cassandra with adaptive consistency algorithm

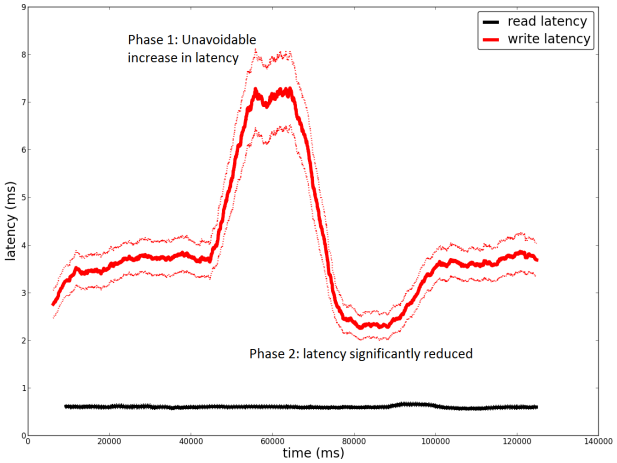

Figure 4: Average latencies for Cassandra with adaptive consistency algorithm

7 Conclusion

In this paper we consider the model of probabilistically bounded staleness (PBS). We modify Cassandra to adaptively change its replication configuration based on the PBS model. We further extend this model to support node failures, and include these enhancements in our implementation of the Adaptive Consistency Algorithm. We evaluate our modified branch of Cassandra and show that it can perform significantly better than the current Cassandra-trunk under varying workloads and node failures.

7.1 Future work

Further extensions of the model

In this work we considered crashes of a single server. It would be interesting to consider other failure modes, such as a failure of a link between two nodes, of a failure of a whole site, or a partition of the network. One failure mode that particularly is of particular interest is dropped messages. We hope to further investigate this issue and its implication on Cassandra's read-repair mechanism in future work.

Improved consistency predictions

There are several recent and ongoing works aimed toward improving predictions of consistency. Chow and Tan [5] use a graph-based network model to achieve improved predictions of latencies and consistency probability in Cassandra. Rahman et al. [15] use a client-oriented approach to devise a new measurement of consistency in Cassandra.

Improving our understanding of the way consistency behaves, and in particular improving the accuracy of our predictions of consistency could lead to more reliable and more efficient adaptive consistency.

Extending to other systems

Cassandra is an ideal playground for academic work because it is open source and relatively simple; for those reasons it is also gaining popularity in production. Nonetheless, it would be interesting to explore

how do the notions and algorithms described in this work extend to other storage systems.

References

[1] Ittai Abraham and Dahlia Malkhi. Probabilistic quorums for dynamic systems. *Distrib. Comput.*, 18(2):113–124, December 2005.

[2] Divyakant Agrawal and Amr El Abbadi. An efficient and fault-tolerant solution for distributed mutual exclusion. *ACM Trans. Comput. Syst.*, 9(1):1–20, February 1991.

[3] Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica. Probabilistically bounded staleness for practical partial quorums. *PVLDB*, 5(8):776–787, 2012.

[4] Eric A. Brewer. Towards robust distributed systems (abstract). pages 7–, 2000.

[5] Wesley Chow and Ning Tan. Investigation of techniques to model and reduce latencies in partial quorum system.

[6] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *SoCC*, pages 143–154, 2010.

[7] David K. Gifford. Weighted voting for replicated data. In *Proceedings of the seventh ACM symposium on Operating systems principles*, SOSP '79, pages 150–162, New York, NY, USA, 1979. ACM.

[8] Vincent Gramoli and Michel Raynal. Timed quorum systems for large-scale and dynamic environments. pages 429–442, 2007.

[9] Munetoshi Ishikawa, Koji Hasebe, Akiyoshi Sugiki, and Kazuhiko Kato. Dynamic grid quorum: a reconfigurable grid quorum and its power optimization algorithm. *Serv. Oriented Comput. Appl.*, 4(4):69:245–69:260, December 2010.

[10] Mamoru Maekawa. A n algorithm for mutual exclusion in decentralized systems. *ACM Trans. Comput. Syst.*, 3(2):145–159, May 1985.

[11] Dahlia Malkhi, Michael K Reiter, Avishai Wool, and Rebecca N Wright. Probabilistic quorum systems. *Information and Computation*, 170(2):184 – 206, 2001.

[12] Moni Naor and Udi Wieder. Scalable and dynamic quorum systems. *Distrib. Comput.*, 17(4):311–322, May 2005.

[13] Moni Naor and Avishai Wool. The load, capacity, and availability of quorum systems. *SIAM J. Comput.*, 27(2):423–447, April 1998.

[14] Brian Noble, Ben Fleis, and Minkyong Kim. A case for fluid replication. 1999.

[15] Muntasir Raihan Rahman, Wojciech M. Golab, Alvin AuYoung, Kimberly Keeton, and Jay J. Wylie. Toward a principled framework for benchmarking consistency. *CoRR*, abs/1211.4290, 2012.

[16] Werner Vogels. Eventually consistent. *ACM Queue*, 6(6):14–19, 2008.