International Journal of Grid and Distributed Computing
Vol. 9, No. 5 (2016), pp.55-68
http://dx.doi.org/10.14257/ijgdc.2016.9.5.06

An Adaptive Replica Selection Algorithm for Quorum based Distributed Storage System

Zhen Ye and Weijian Hu

Lishui Univeristy
yezhen@lsu.edu.cn

Abstract

In Quorum based cross datacenter distributed storage systems, data replication is a widely used technology. Such systems usually choose weak consistency, means they prone to select less number of replicas in read request in order to improve the performance, which lead to the possibility of getting stale data. In this paper, we propose an adaptive replica selection algorithm to make sure the system does not exceed specific stale read ratio while achieving good performance. Firstly, we choose a suitable distribution model from many candidates to estimate the time interval between current read request and nearest write request before it. Then we use Monte-Carlo method to simulate the processing order of read request and write request in different replicas. At last we use those estimated values to determine minimal number of replicas each read request needs to select in order to achieve a specific consistency level. We conduct a comprehensive experiments and the result shows our algorithm is effective.

Keywords*: Data Replication; Replica Selection; Quorum; Cross Datacenter; Replica Consistency*

1. Introduction

With the popularity of cloud computing and big data, cross datacenter storage system is widely adopted [1, 13]. In such system, data are stored in different datacenters, each of which can serve the users nearby thus balance the system load and provide redundancy. In order to improve system availability and get better performance, cross datacenter data replication is necessary. Data replication is a technology that replicates data into different copies and stores each of the copy into different location.

One of the most widely used data replication technology are Quorum based. Quorum system [6] is a redundancy set system that needs to satisfy $W+R>N$. N is the total number of data replica. W is the write quorum size. A write request will forward to all N replica nodes and wait the first W replicas return back. Other replica nodes will return the result in asynchronize way. R is the read quorum size that a read request needs to contact. $W+R>N$ means the sum of read replicas number and write replicas number is larger than total number of replicas, thus at least one replica node involved in both read request and write request. This strategy assures read request can always get the most updated data. However, Quorum system may unnecessarily reduce system performance. *e.g.* when the read/write ratio is high, once a read request comes, the corresponding recent write request may occur long ago and all N replicas are already updated. In such situation, this read request only needs to contact one replica rather than R replicas to get fresh data. Network latency between different datacenters has huge difference, thus in cross datacenter scenario this performance reduction issue is more obvious.

Since strict Quorum based system has relatively high response latency, cross datacenter systems often select Partial Quorum to get eventually consistency [3]. In eventually consistency, read request may get stale data, however if there is no more update request, finally all replicas will become consistency. In Partial Quorum system, $W+R<=N$, which

ISSN: 2005-4262 IJGDC

means read request and write request may has no replica node in common. If one replica receives the read request before update request synchronizes the fresh data, it will return stale data. If all R replicas return the stale data, this read request will return the stale data. For an application, the more stale data it returns, the worse user experience it has.

In Quorum based cross datacenter storage system, when requesting a data, the number of replicas you select has different impact on system performance, availability and the probability to get fresh data. The more replicas a read request chooses, the more result it needs to wait, thus it will increase response time and reduce performance. The advantage is it has more chance to get fresh data.

In this paper, we propose an adaptive replica selection algorithm to make the Quorum based cross datacenter distributed storage system performs as good as possible while assure it returns a specific percentage of fresh data. We first select a most suitable request distribution model, from many widely used candidate models, to fit current access pattern and estimate the time interval between each read request and most recent write request. Then we apply Monte-Carlo [8] method to simulate the processing order of read request and write request in different replicas. After that, we can use these estimated values to determine, in real time, minimal number of replicas the current read request needs to visit in order to reach a specific consistency. In this way we can reduce the unnecessary of cross datacenter visiting, thus can reduce response time and improve system throughput.

Our contributions include:

1. Describe the processes of read/write request in Quorum System and define in which scenario read request may get stale data.
2. Propose an adaptive replica selection algorithm to improve system performance as much as possible while still within the bounds of stale data rate.
3. Integrate the algorithm into representative and widely used distributed storage system Apache Cassandra.
4. Conduct a comprehensive experiment and the result shows our algorithm is quite effective.

The remaining paper are organized as follows: Chapter 2 surveys the related work; Chapter 3 introduces the background and in which scenario read request may get stale data; Chapter 4 proposes the replica selection algorithm; Chapter 5 describes the experiment and their result; Chapter 6 summarizes the paper.

2. Related Work

Quorum based cross datacenter storage system, replica selection and replica consistency are hot topics that has been widely studied.

VOGELS W. [3] divides replica consistency into strong consistency and weak consistency. In strong consistency, once the write finished, all following read request will get the most updated data. However, in weak consistency, system does not guarantee the following read request always get the fresh data. Based on how stale data is retrieved, weak consistency can be further divided into different types, among which causal consistency and eventually consistency are most researched. The causal consistency assures if the read request has causal relationship with its pre-write request, it always get fresh data. If there is no such causal relationship, this read request may retrieve the stale data. Eventually consistency only promises if there is no further update request, all the replicas will be consistency eventually.

Dynamo [10] and Cassandra [4] are two representative Quorum based cross datacenter storage systems. The data model of Dynamo is key-value, while Cassandra is Bigtable [5] style composed of sorted mapping table, which support more complex operation. In data replication related module, their strategy is similar. Both of them can contact different number of replicas according to the configuration. For each read request and write request, there are three types of replica consistency configuration: *ONE*, *QUORUM*, *ALL*.

ONE: the request can return the result to client as soon as one replica finishes it; QUORUM: the request can return the result only at least *N*/2 replicas finish it; *ALL*: the request must wait until all the replicas finish the operation. When one of write request or read request set replica consistency level to *ALL* or both of them set the level to *QUORUM*, the system has strong replica consistency. Since in such situations, at least one replica node is included in both read request and write request, thus must return the most updated data. Otherwise, the read request may get stale data. Such consistency configuration algorithm does not consider the dynamics of request model, and when the read/write ratio is high, it will visit some unnecessary additional replicas, which impact system performance and throughput.

There are some researches on adaptive replica consistency, which can adaptively support different consistency level according to system requirement. WANG X *et al*. [9] divides request access pattern into four categories: read frequency is low, update frequency is low; read frequency is high, update frequency is low; read frequency is low, update frequency is high; read frequency is high, update frequency is high. In different categories, it selects the most suitable consistency strategy respectively to balance the consistency, availability and system performance. The system is composed of one master node, three deputy nodes and several child nodes. All update requests are served by master node and synchronize to deputy nodes and child nodes. The disadvantage of this system is it uses fixed threshold to categorize different scenarios, which is not suitable while the access pattern changes or vary frequently. In addition, this system depends on a specific topology, which is not widely used in current cross datacenter distributed scenarios.

Harmony [11] is a system that can dynamically adjust replica consistency according to the requirement. It proposes an estimation model to predict the stale read. By collecting read/write access frequency, network latency, most recent read/write access time and other information, it can predict the stale read ratio in real time and achieve the required consistency level with relatively good performance by elastically increase or decrease the number of replicas involved in each read request. However, Harmony is a white box model, which decides the replicas number of each request by using mathematical formula derivation. However, since there are so many factors that can impact the result and lots of those factors change in real time, such white box analysis may not get precise result. Besides, Harmony assumes the request access pattern meets Poisson process, however, different applications' access patterns are different, which means Harmony has its usage limitation.

In most system, it defines the rate of stale read can be tolerated, and then try to improve system performance as much as possible while still not exceed such stale read rate. However ZHU Y *et al*. [12] takes another approach, it defines the longest response time it can tolerate and try to enhance consistency level as much as possible within this time. It breaks the read/write access into 6 steps: reception, transmission, coordination, execution, compaction and acquisition, each of which can further breaks into more small steps. Then it use linear regression to predict the cost time of next request for each step. When a request comes, it maximizes the number of steps this request covers within the tolerated time, thus achieve the maximize consistency. However, the stale read rate in this system is unpredictable.

[2] introduces Probabilistically Bounded Staleness (PBS) consistency. PBS describes two ways to estimate the staleness of data: version based and time based. Firstly, it derives a closed-form solution for version based data staleness. Then it models time based staleness and applies it in Dynamo style systems. PBS uses Monte Carlo simulation to describe the time based data staleness. Our paper is inspired by PBS and use the same Monte Carlo simulation to estimate the minimal replica number a read request needs to contact in order to get a specific fresh data rate.

3. Background

In our scenario, request will be sent to the nearest datacenter, within which a node will serve it and becomes coordinator of this request.

The process of an update request is as follows:

1. User send update request to write coordinator.
2. Write coordinator forward this request to all *N* replica nodes.
3. For each replica node, it serves the request and sends back the result.
4. Write coordinator wait until it receives first *W* response and return the result to the user.
5. The remaining (*N*-*W*) replicas reply to write coordinator in asynchronizes way.

Figure 1 shows this process.

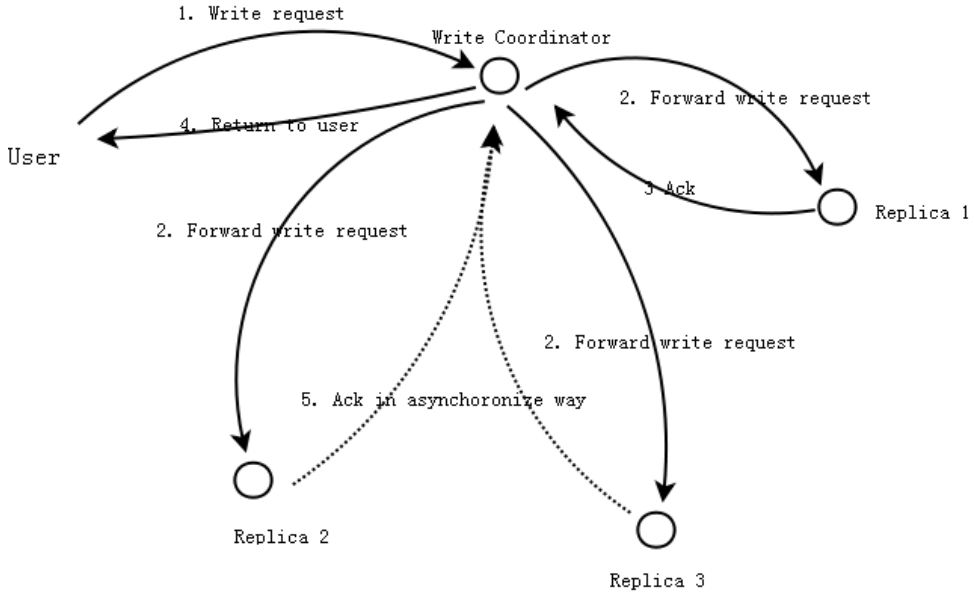


Figure 1. Quorum based Write Request

The process of a read request is as follows:

1. User sends read request to read coordinator
2. Read coordinator forwards this request to *R* replica nodes.
3. After receive this read request, each replica node retrieve the data and send back to read coordinator.
4. Read coordinator compares all *R* replies, picks the latest data and sends it back to user.

Figure 2 shows this process.

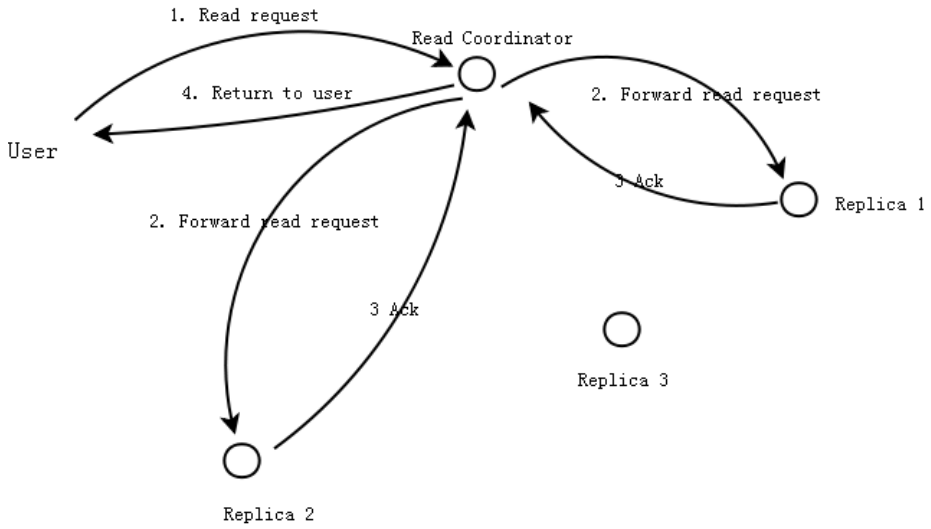


Figure 2. Quorum based Read Request

In the following situation user may get stale data:

1. The update request and the read request aim to the same data.
2. Read request arrives at read coordinator after corresponding update request finish step 4 and commit the result to the user.
3. There is no intersect node between *R* replica nodes that serve the read request and *W* replica nodes that return update request faster.
4. In all *R* replica nodes, there is no one return the most updated data.

Our goal is: within the bounds of stale read rate, according to current read/write access pattern, node load and network condition, adaptively select minimal number of replica nodes each read request needs, thus improve system performance and throughput as much as possible.

From above read/update request process and the scenario to get stale data we can see in order to determine the number of replicas nodes each read request chooses, the key is to accurately evaluate most recently update request commit time before this read request and the process sequence of read/update request in different replica nodes.

4. Replica Selection Algorithm

In this chapter, we introduce how to determine the number of replica nodes one read request should select in real-time, according to evaluate reach time of nearest update request and the processing order of read request and write request in different replica nodes. Section 4.1 shows how to select a suitable distribution model to fit and evaluate reach time of write request. Section 4.2 describes how to simulate reach process of read request and write request and then using this result to determine minimal number of replicas current read request need to wait in order to obey the stale data rate.

4.1. Select Suitable Distribution Model to Estimate Reach Time of Recent Update Request

When a read request arrive a read coordinator node, it needs to know the recent update request. However, the corresponding write coordinator may located in different

datacenters and its arrival time of update request has not synchronized to this read coordinator yet, thus need to estimate this time using some ways.

Usually, one application's read/write arrival times roughly fit one kind of distribution model. Different applications' request access model is different, thus should use different distributions to model. To one specific application, the access model is difficult to determine at the beginning and will not change much thereafter. Observing these characteristics, we first pick some common distribution models as the candidate, recording some arrival times at the beginning and using those candidates to fit such arrival times. After comparing those fitting rate, we can select the most suitable distribution model to predict update request arrival times of this application.

At first, there is no data to fit, so the algorithm uses Quorum based strong consistency to serve read request. In this strategy, all replicas can get fresh data. System will record each update arrival time T_{write} and add it into arrival time set S_{write}. Figure 3 presents this process.

initReplicaSelection():
1. Use Strong Quorum consistency
2. $S_{write} \leftarrow S_{write} \cup T_{write}$

Figure 3. Initial Replica Selection

After collecting some data, system has enough information to determine which kind of distribution model can be used to fit this application. Figure 4 describes the model selection process. We set some common distribution models as the candidate (Currently includes Poisson distribution, Zipf distribution, lognormal distribution and uniform distribution), by comparing each distribution's goodness of fit, we can select the most suitable distribution model. We use Kolmogorov-Smirnov [7] method to do the goodness of fit evaluation.

chooseSuitDistribution():
1. for each candidate model $canDis_i$
2. $fitRating_i \leftarrow fitting(canDis_i, S_{write})$
3. $fitRatings \leftarrow fitRatings \cup fitRating_i$
4. end of for
5. $selectedDis \leftarrow selectBestFit(fitRatings)$

adjustDistParameter():
1. $newParas \leftarrow MLE(selectedDis, S_{write})$
2. *updateParas(newParas, selectedDis)*

Figure 5. Adjust Model Parameters

Usually request access pattern will not change much once determined. However, as time goes by, this pattern still can have some small changes. So we need to adjust distribution model's parameters to adapt these changes. As Figure 5 shows, we use the maximum likelihood estimation method (MLE) to do such parameter estimation. Every once in a while system will run this parameter adjustment function to predict the future arrival times more accurately.

Figure 6 describes how to use selected distribution model to estimate recent update request arrival time. At first, it retrieves most recent write arrival time *nearestArrivalTime* and most recent commit time *nearestCommitTime* this node already knows. Then use

selected distribution model to estimate the arrival time of next write request. *CommitLatency(W)* can estimates time interval between write request reaches to the write coordinator and this write coordinator receives all *W* replies and commit. This estimation is processed by using Monte Carlo simulation and will be introduced in detail in the following section. This estimated latency is used to update the value of *nearestCommitTime*. If the new *nearestCommitTime* value does not exceed current time, it means this write request has been committed before current read request arrives. By repeating this process we can get the most recent write request arrival time.

```
forcastNearestWriteTime():
1.  nearestArrivalTime
        ←getNearestArrivalTime(S_write)
2.  nearestCommitTime
        ←getNearestCommitTime()
3.  while nearestCommitTime is less than currTime
4.    addedTime
        ← nextTimeInterval(selectedDis)
5.  nearestArrivalTime
        ←nearestArrivalTime + addedTime
6.  nearestCommitTime
            ←nearestArrivalTime + commitLatency(W)
7.  end of while
```

Figure 6. Evaluate most Recent Write Request

In this paper, we only consider the scenario when application's read/write access pattern fit single distribution model. In the future research we will consider the mixture distribution model. The way is assigning different weight to different distribution and mixed them into one model, or using generic algorithm to make the evolution and hybrid between different distribution models and finally get the suitable mixture model.

4.2. Using Monte-Carlo Method to Estimate Response Sequence of Read and Write Request

If a read request wants to get fresh data, in all its replies retrieve from *R* replica nodes, at least one reply must contain the most updated data. Towards each replica node, it can only return fresh data when read request handled after corresponding update request. It is very difficult to determine the handle order of read request and update request by using the model built through directly analysis of the whole process. The reason includes: read request and update request may from nodes located in different datacenters and network latencies between those nodes and replica nodes are quite different and changes dynamically; In order to accurately estimate the queue time of read request and update request, we need to build a very complex queueing model; Request handle time are much related with replica node's current workload.

In this paper we use simulation way to achieve the same goal. We use Monte-Carlo method to randomly select some read/write request latency from history data recorded in advance, and then uses such samples to estimate response time and the handle order in different replica nodes. Thus we can know the stale data rate for each replica number it selects. By comparing those stale rates with pre-defined *allowed_stale_ratio*, we can determine the minimal number of replica this request use. The process is inspired by WARS Monte-Caro method [2]. Figure 7 presents the detail algorithm.

At first, the algorithm sets read request replica number *R* as 1, and run *numberTrials* times Monte-Carlo simulation. $wLatency_i$ is the time interval of sending write request

from write coordinator to replica i. $wLatenciesList_i$ records all such write time intervals, from which the algorithm randomly select one as the value of current $wLatency_i$. $rLatency_i$ is the time interval of sending read request from read coordinator to replica *i*. The value is also picked from history records. After all *N* replica nodes finish selection, they will be ordered by their read time interval, and the first *R* replica nodes will be chosen as the nodes to be visited by current read request. In all *R* replica nodes, as long as one replica receive update request before read request, this read request can get most updated data. After *numberTrials* times simulation, we compare the stale data rate and allowed maximum stale rate. If it exceeds the bounds, it means current *R* number cannot meet consistency requirement and its value should be increased, repeat this process until *R* meet the requirement.

```
monteCarloSimulation(R)
1.  repeat run numberTrials times
2.      for each replica i in N
3.          wLatency_i ←randomSelect(wLatenciesList_i)
4.          rLatency_i ←randomSelect(rLatenciesList_i)
5.          wLatencyMap.add(i, wLatency_i)
6.          rLatencyMap. add(i, rLatency_i)
7.      end of for
8.      select R fastest replica from rLatencyMap
9.      for each replica i in R
10.         wLatency_i ← wLatencyMap.get(i)
11.         rLatency_i ←rLatencyMap.get(i)
12.         if rLatency_i + (currTime- nearestArrivalTime) > wLatency_i
13.             consistentRead add 1
14.             break out of for loop
15.         end if
16.     end of for
17. end of repeat run
18. if (1-consistentRead/totalRead) larger than allowed_stale_ratio
19.     R←R+1
20.     run monteCarloSimulation(R)
21. end if
```

Figure 7. Monte-Caro Simulation

5. Experiment

Our algorithm can apply in many Quorum based distributed storage systems. In this experiment we integrate it into Apache Cassandra, which is a NoSQL storage system widely used in industry domain. The version we use is Cassandra 1.2.4. We use 7 desktops to simulate 7 datacenters. Each machine contains a Cassandra node. The network latencies between different datacenters are simulated by adding some extra delay time. The delay times are real data collected by deploying nodes in Amazon EC2 datacenters and then ping each other.

We use YCSB as our benchmark [14]. YCSB is a standardized benchmark suit used to evaluate different NoSQL products and includes many workloads. In this experiment, we use Workload-A and Workload-B. Workload-A is update-intensive and write/read request

ratio is 1:1. Workload-B is read-intensive and write/read request ratio is 1:9. We deploy YCSB application in a dedicated desktop. One thread represents one user, thread number are raise from 1 to 100.

In each experiment, the total number of read/write request is 10 million, each write request includes 1k data, number of data replicas is 3 (N=3). The comparison algorisms are eventually consistency (W=1, R=1) and strong consistency (W=1, R=3). Since the allowed stale data rate has impact on our algorithm, we choose two rates, 5% and 10%, means the allowed bounds of stale data rate is 5% and 10% respectively. They are named as Adaptive-5% and Adaptive-10%. In our adaptive algorithm, W=1, number of R is dynamically changed according to current situation. The evaluation criteria include average network latency, throughput and actual number of stale data it returns. To see the result on different distribution models, we evaluate two scenarios: Zipf [15] distribution and Poisson distribution.

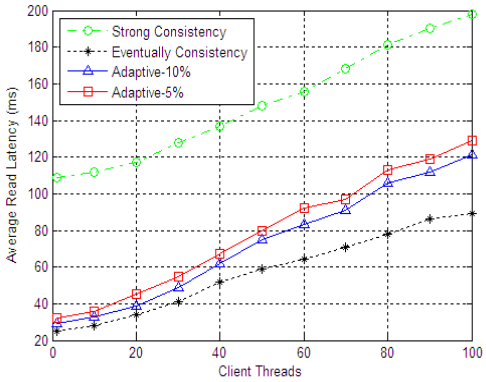
5.1. Impact on Network Latency

This set of experiment shows result of average latency. We conduct the experiment on both Workload-A and Workload-B. For each workload, we evaluate both the Zipf distribution and Poisson distribution. Figure 8.a to 8.d show the detail result.

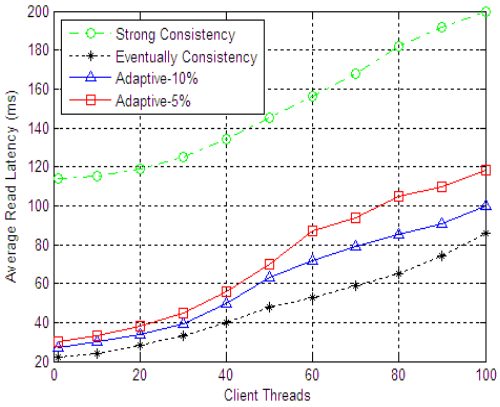
From Figure 8 we can see strong consistency causes the highest latency. Since in strong consistency, read coordinator needs to wait response from all R replicas and then commits the result. In eventually consistency, the coordinator only needs to wait a fastest response before it commits the result, thus the latency is lowest. In many of the situation, the fastest replica node and the coordinator are located in the same datacenter, which reduce the wait time a lot. For adaptive algorithm, the latency is only a little higher than eventually consistency and much lower than strong consistency. Since this algorithm can dynamically select the replica number according to current read/write ratio, network latency and system load, thus avoid waiting some unnecessary result from distant replicas, which will cause additional delay. Comparing with Adaptive-5%, Adaptive-10% can tolerant more stale data and in some requests it contacts less replica nodes, so the latency is lower.

As the number of thread increase, read latency also increase, this is because when there are more concurrent requests, the competition towards system resources is prone to cause the performance bottleneck.

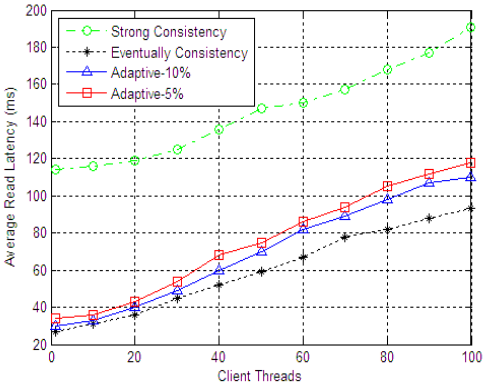
The adaptive algorithm performs well in both Poisson distribution and Zipf distribution. This result shows that according to the distribution of applications' read/write request arrival time, the algorithm can adaptively select the most suitable distribution model to fitting the result and thus improve the evaluation accuracy. Comparing the result of Workload-A and Workload-B, we can find within the same distribution model, the higher of read/write ratio, the better adaptive algorithm can perform. Since when the read request ratio is higher, the time interval between last update request and current time may become larger, and the more chance the read request wait for less number of replica node result. For algorithms using strong consistency and eventually consistency, latencies are remain the same, since the number of replicas coordinator need to contact is the same in two workloads.



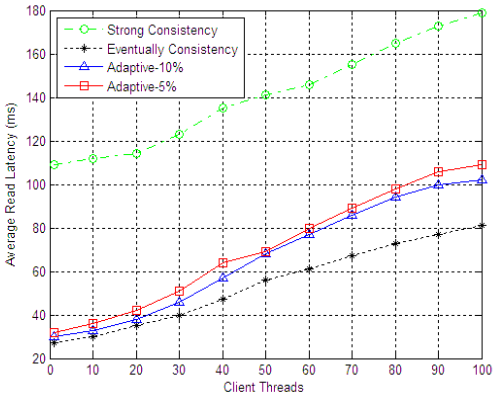
(a)Workload-A: latencies in Zipf



(b) Workload-A: latencies in Poisson



(c) Workload-B: latencies in Zipf



(d) Workload-B: latencies in Poisson

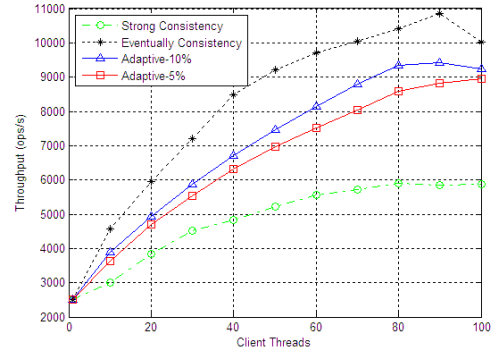
Figure 8. Impact on Network Latencies

5.2. Impact on Throughput

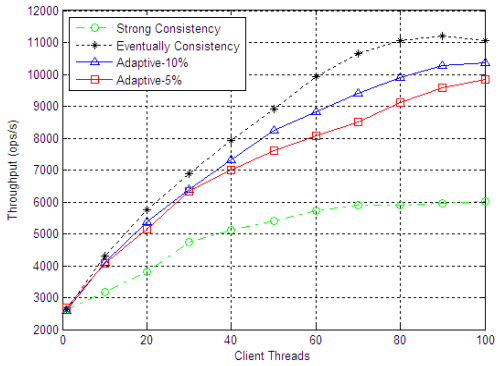
This set of experiment shows the throughput of adaptive consistency, strong consistency and the eventually consistency algorithms. Figure 9.a to 9.d describe the result.

In Figure 9 we can see, as the number of client threads increases, the throughput of all algorithms increase. The reason is within the bounds of system resource bottleneck, as the concurrent number increases, the number of finished request within unit time is also increase. However, as the client threads number continue increase, the growing speed of throughput become slowly and finally it even decreases. It is because when the system load reaches to the bottleneck, one node or some nodes may overload and thus has serious impact on the whole system's throughput.

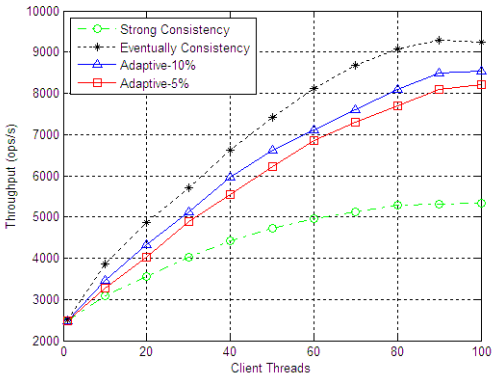
Among all algorithms, strong consistency has lowest throughput, since when coordinator waiting for replica nodes to return the result, it also consumes resources, and strong consistency's waiting time is longest and thus when the clients thread number are same, this strategy generate largest system load and prone to make node and system overload. With the same reason, eventually consistency has highest throughput. The adaptive algorithm performs relatively well in both read intensive and update intensive workloads. The overall throughput is only a little lower than eventually consistency and much higher than strong consistency. Adaptive-10% performs better than Adaptive-5%.



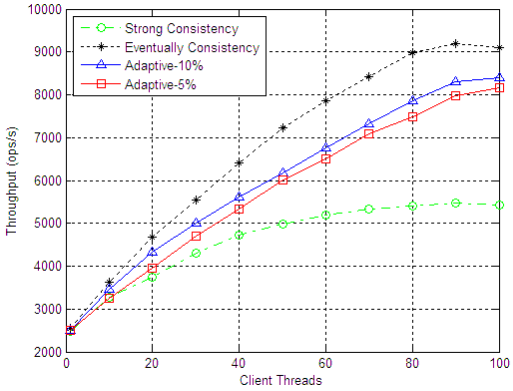
(a) Workload-A: throughput in Zipf



(b) Workload-A: throughput in Poisson



(b) Workload-B: throughput in Zipf



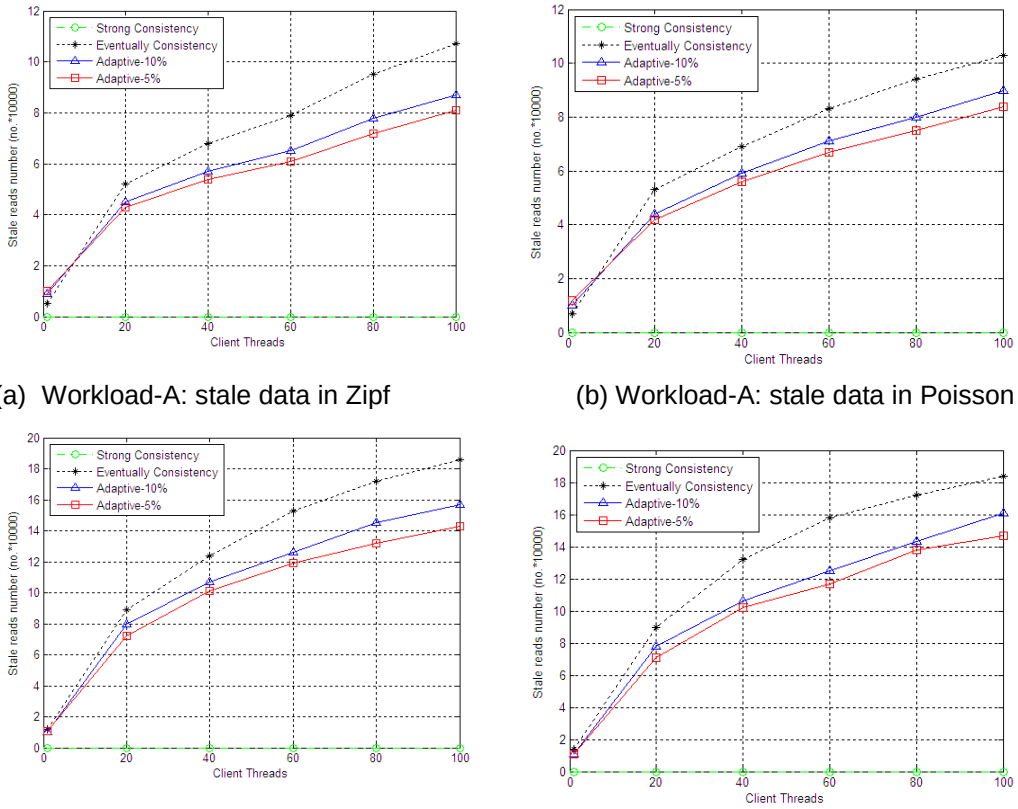
(c) Workload-B: throughput in Poisson

Figure 9. Impact on Throughput

5.3 Actual Number of Stale Data

In this set of experiment, we compare actual number of stale data adaptive consistency and eventually consistency algorithms retrieve, through which we show adaptive algorithm can provide higher consistency. Since strong consistency algorithm always retrieves fresh data, the number of stale data it gets is 0 in all scenarios. It is very difficult to judge if current read request retrieve fresh data or not. In this paper, the approaches we use is after adaptive read request and eventually read request immediately following a strong read request respectively, and then compare the result of strong request with the other two. Since strong read request always get fresh data, comparing with it can determine if pre-request retrieves the stale data. Although this approach will affect system performance, it can still show consistency level of adaptive consistency and eventually consistency. Figure 10 shows the result.

From Figure 10, in every workload and every distribution models, eventually consistency algorithm always gets the most number of stale data. For adaptive consistency, the more stale data rate it allowed, the more number of stale data it will get, so the number of stale date Adaptive-10% get is higher than those of Adaptive-5%. The higher the read/write ratio, if the total request number is same, the more number of read request it will have, thus the number of stale data it get is higher. This is the reason why Workload-B gets higher number of stale data than Workload-A.



(a) Workload-A: stale data in Zipf

(b) Workload-A: stale data in Poisson

(c) Worklad-B: stale data in Zipf

(d) Workload-B: stale data in Poisson

Figure 10. Actual Number of Stale Data

6. Conclusion

In Quorum based cross datacenter distributed storage system, there is a balance between consistency and system performance. Under the same conditions, the higher the consistency level, the worse the system performance. However, for a specific application, its read/write access pattern, network latency and system load are always changes dynamically. So at different time, to reach the same consistency level, the impact on system performance is different. However, current applications usually select fix number of replicas to visit for a read/write request. If this number is large, in some scenarios it will visit some unnecessary replicas, which reduce system performance. On the other side, if this number is small, sometimes it is prone to get stale data, which reduce the user experience.

By observing those variant environments, in this paper, we propose an adaptive replica selection algorithm for such quorum based distributed storage system. The algorithm can determine minimal number of replicas a read request needs to contact in real time and thus improve system performance as much as possible while still within the constriction of specific consistency level. This means, by using such adaptive replica selection algorithm, the system can reach high system performance while still maintain good user experience.

References

[1] S. Kadambi, J. Chen, B. F. Cooper, D. Lomax, R. Ramakrishnan, A. Silberstein, E. Tam and H. Garcia-Molina. Where in the World is My Data. Proceedings of the VLDB Endowment. 4, 11 (**2011**)
[2] P. Bailis, S. Venkataraman, J. M. Hellerstein, M. Franklin and I. Stoica. Probabilistically Bounded Staleness for Practical Partial Quorums. Proceedings of the VLDB Endowment. 5, 8 (**2012**)
[3] W. Vogels. Eventually Consistent. Communications of the ACM. 1, 52 (**2009**)
[4] A. Lakshman and P. Malik. Cassandra-A Decentralized Structured Storage System. SIGOPS Oper. Syst. Rev. 2, 44 (**2010**)
[5] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. ACM Trans. Comput. Syst. 2, 26 (**2008**)
[6] P. Song, J. Sun and Z. He. Survey of fault tolerant on Quorum based system [J]. Journal of Computer Research and Development. 4 (**2004**)
[7] A. Justel, D. Peha and R. Zamar. A multivariate Kolmogorov-Smirnov test of goodness of fit. Statistics & Probability Letters. 3, 35 (**1997**)
[8] J. Hammersley, D. Handscomb and G. Weiss. Monte carlo methods. Physics today. 2, 18 (**1965**)
[9] X. Wang, S. Yang, S. Wang, X. Niu and J. Xu. An Application-Based Adaptive Replica Consistency for Cloud Storage. International Conference on Grid and Cloud Computing, (**2010**) November 1-5; Nanjing, China
[10] G. Decandia, D. Hastrun, M. Jampani, G. Kakulapati and A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall and W. Vogels. Dynamo: Amazon's Highly Available Key-value Store. Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles, (**2007**) October 14-17; New York, USA
[11] H. Chihoub, S. Ibrahim, G. Antoniu and M. S. Perez. Harmony: Towards Automated Self-Adaptive Consistency in Cloud Storage. IEEE International Conference on Cluster Computing, (**2012**) September 24-28; Beijing, China
[12] Y. Zhu and J. Wang. Malleable Flow for Time-Bounded Replica Consistency Control. OSDI Poster, (**2012**) October 8-10; Hollywood, USA
[13] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang and D. Woodford.
Spanner: Google 's Globally-Distributed Database. Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation, (**2012**) October 8-10; Hollywood, USA
[14] B. F. Cooper, A. Silberstein, E. Tam, R. Ramkrishnan and R. Sears. Benchmarking Cloud Serving Systems with YCSB. Proceedings of the 1st ACM symposium on Cloud computing, (**2010**) June 10-11; Indianapolis, USA
[15] A. Siberstein, J. Terrace, B. F. Cooper and R. Ramakrishnan. Feeding Frenzy: Selectively Materializing Users' Event Feeds. Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, (**2010**) June 6-11; Indianapolis, USA

Authors

Zhen Ye. Zhen Ye receives his Ph.D degree in Computer Science and Technology in 2013 from Zhejiang University, China. Currently he is a lecturer in Lishui University, China. His areas of interest are Distributed System, Computer Vision and Machine Learning.

Weijian Hu. Weijian Hu receives his master degree in Software Engineering from Hangzhou Dianzi University, China. Currently he is a lecturer in Lishui University, China. His areas of interest are computer vision and virtual reality.

