The use of weak consistency models in cloud storage

Thilo Greiner
University of Kaiserslautern, Embedded Systems Group
t_greiner12@cs.uni-kl.de

Abstract

Weak consistency, more precisely eventual consistency, is the most common consistency provided by the majority of the cloud storage providers. But there are many other consistency guarantees that can be ensured without the use of strong consistency, also known as sequential consistency.

This paper gives a general overview of the most important weak consistency models and looks at their advantages, disadvantages, properties and additionally gives some general implementation ideas.

Finally, it will provide an insight into the works of some new approaches to improve the trade-off between performance, availability and consistency. Some of them change the consistency level dynamically during runtime. It also gives a general idea how these techniques work and what they bring to the table in contrast to static assigned consistency.

1 Introduction

As the Internet grew more and more in the last past few years, ensuring performance properties like scalability, availability, low latency and reliability has become a big problem. This is especially true for big online communities and web shops like Facebook, Google, Twitter, and Amazon just to name a few of them. The problem is that many users want to use the offered services at the same time. With only one big datacenter, this would lead to a impractical situation where the servers are overloaded. When this happens, users become dissatisfied, because they are not getting their information fast enough. This can result in high losses for these companies as customers may move to competitors.

Therefore, the data of these big websites are distributed on many servers. This use of several ”micro-data centers” instead of one big ”mega-data center” [8] is an important improvement regarding performance, e. g. lower network latency between server and client, availability and reliability. Such systems are much more robust against any network or server errors which would cause the web service to be unavailable. Using such cloud systems allow web services to scale limitless with little to no reduction of performance.

One problem these highly distributed systems have is the assurance of consistency of the data replications. According to the CAP theorem [4][11] it is impossible to simultaneously ensure consistency, availability and partition tolerance in a distributed system. As availability and partition tolerance are the more important aspects of a web service, most of the time consistency is relaxed in favor of the former two properties.

The rest of the paper is structured as follows. The next section gives an overview of recent related work. Section 3 explains why data replication and distribution are important aspects of today’s cloud storage systems. The 4th section introduces several different weak consistency models and their properties and gives some general implementation ideas. Section 5 then introduces different ideas of how to improve the cloud storage system by special approaches dealing with more than one kind of consistency at once. Finally, Section 6 concludes the paper.

2 Related Work

There are many papers dealing with the different levels of consistency. Eventual consistency in particular has been in focus of a lot of research in the past. The first part of this paper lists and explains some of the most important weak consistency models. The most similar work here is Tanenbaum et al. [18] as this paper adopts their classification of the different consistency models into client-centric and data-centric models. They explain in detail what each model provides and how to implement it with some examples.

Another paper that lists different consistency models is Dough Terry's ”Replicated data consistency explained through baseball” [20]. The authors approach is less formal and he tries to explain the different consistency models using baseball as an example. He shows that strong consistency is not always needed and you can save costs if you know what application needs which type of consistency.

Vogels (2009) [22] gives an overview about so called session guarantees that can be used to increase consistency based on eventual consistency. It also shows that some of these guarantees can be combined to increase consistency even more to satisfy demands that otherwise may need strong consistency.

Other papers giving a review of weak consistency models are Fekete and Ramamritham (2010) [10], Fuertes et al. (2012) [17] and Terry et al. (1994) [19]. The last one additionally gives some ideas for implementation and several use cases for the different models.

The second big part of this paper is the improvement of the consistency of the cloud storage system by using more than one kind of consistency. Most important papers here are Kraska et al. (2009) [13], Chihoub et al. (2013) [6], Esteves et al. (2012) [8] and Lu et al. (2007)[15] or Lu et al. (2008) [16] All are introducing a different approach to handle consistency.

The advantage of this paper over others it that it first gives a general overview of the most important weak consistency models for cloud computing, including their properties and their implementation in a general way. It then shows the different approaches how people try to increase performance even more. This paper can be used to get to know the techniques of cloud storage systems, how they work, how they are used, and why replicating data is so important.

3 Why Use Replicated Data

As already mentioned, using replicated data improves the performance of the cloud service. There are several servers geographically spread across a country or even the world, so there is always one server close to the client. Due to smaller distances, the transportation time is much smaller assuming that the client wouldn't be close to the single server. Furthermore, the fact that there is more than one hotspot allows the load of the network and the work of responding requests to be divided. This improves the speed of both the network and the server-system. As every server has a more or less fresh copy of the data, it is possible to access the data stored in the cloud even if one or more of the servers are down due to network errors, system failures etc. The client just connects to another server that can perform its request. These properties cause the system to be much more available and reliable than a single data center. Another advantage of distributed and replicated data is that corrupt writes can be undone easily. When a write fails and the system needs to recover the right value, it can just query other servers that are more or less up to date to determine the old value.

4 Weak Consistency Models

Unfortunately, the replication of data causes some problems, too. The most convenient way for the client would be to provide strong consistency as it assures actuality of the data in the cloud storage. That means every update on any server will be applied on any other replica. This way every client then sees the most recent value regardless of which data item, server, or client that updated that item. Providing strong consistency causes a lot of network traffic in data centers and between them. This is because the updates must be immediately propagated to the other servers and the protocols for distributing the data are much more complex.

Therefore, performance would be lowered causing higher response times, more computation costs and loss of quality of service which leads to monetary loss. So on the one hand replication increases the performance, but on the other hand this performance gain is decreased by the communication overhead of strong consistency provoking a higher cost per transaction. In addition to performance availability is reduced as well [3].

Fortunately, strong consistency is not needed in most cases. Therefore, many big solutions like Amazons Simple Storage Service [1] and Googles BigTable [5] use a weak consistency model like eventual consistency. Many other weak consistency models have been developed to allow the host to relax consistency and still satisfy the demands of its customers by ensuring some additional guarantees. Using weak consistency improves performance at the cost of the guarantee of consistent data. Although the returned value of a request may not be up to date, some applications can still use the data. Especially with some of these additional guarantees, many applications can run on top of a cloud storage system only providing weak consistency, without any limitations.

This paper adapts the division of the weak consistency models into two different types from Tanenbaum et al. [18]. They distinguish data-centric models from client-centric models. Data-centric models describe a setup where multiple processes try to access data simultaneously whereas client-centric models look at the consistency for a single client. Such a client may be mobile, so it can update and read data from any place. Which model to use strongly depends on the applications flexibility.

The following section gives an overview of the most important weak consistency models. Each subsection describes one model by its properties and shows a general way how to implement it.

4.1 Data-Centric Consistency Models

4.1.1 Sequential Consistency

Although, sequential consistency is not really a weak consistency model we will have a short look at it, as the next consistency level is based on this one. Sequential consistency originates from multiprocessor computing and was introduced by Lamport in 1979 [14]. The following condition for sequential consistency is taken from Tanenbaum et al.[18]:

> ”The result of any execution is the same as if the (read and write) operations by all processes on the data store were executed in some sequential order and the operations of-each individual process appear in this sequence in the order specified by its program”

That means that there is an order for all operations on the data in the cloud storage. All processes must see the same order of writes. But this order does not have to be the real time order an omniscient observer would see. The order is time independent and has to be determined and propagated to all servers.

Sequential consistency can be seen as a border between strong and weak consistency. All models that have weaker consistency than sequential consistency can be called weak consistency models.

4.1.2 Causal Consistency

As mentioned before, causal consistency is deduced by relaxing sequential consistency. This model uses the fact that there are causally related and unrelated operations. There is no need to have a strict order for unrelated operations as a disordered execution of operations will not cause any inconsistency of a stored value. But if there is a causal relation between two or more operations, their order must be identical for all servers similar to sequential consistency. As updates are propagated in the background from server to server, it may happen that the updates arrive in different ordering at the servers. To prevent inconsistencies because of a different execution order, a global order has to be ensured.

As an example of how to determine if events are causally related the system may proceed like this: If there is a read on data item x and then a write, these operations may be causally related. If there are two or more writes simultaneously on different data items, these operations may not be causally related, as they occur concurrent.

This division between causally related and unrelated operations increases the performance of the system, simply because less operations have to be kept in order. Therefore, less computation and administrative costs are needed for this unrelated data.

4.1.3 Bounded Staleness

The idea behind bounded staleness is to guarantee a client that the requested value will not be too far out of date. So with every read the client will see all updates that happened before a specific point. This point can be a time value, a number of missed updates, or a simple value. First one is called time bound, which means that the read includes all updates that were applied a certain amount of time before, for example 5 minutes. The second one ensures that the data item did not miss a specific number of updates. Last one, called value bound, assures that the value of the up-to-date data item differs no more than the bound from the current value on the requested server. The second and the third version of bounded staleness could be useful for example for a web-shop. To prevent overselling, the number of available products should be updated every 5 sales or so. The time bound version could be used for news pages, where all servers should show the latest news after a maximum delay of 5 minutes.

4.2 Client-Centric Consistency Models

Instead of looking at multiple processes and simultaneous accesses, client-centric consistency only looks at the guarantees given to a single client and their operations in the cloud storage. They provide no additional guarantees for concurrent operations of different clients.

First, we will have a look at eventual consistency. The weakest and simplest model of all of them. The consistency models following that section are all build upon eventually consistency by adding some more properties. As all these models use sessions as a basis of their improvements, they are called session guarantees. Each respective section will describe the model and then give an idea of how to implement it. For implementation we assume following:

Server side:

- Any server receiving a write assigns a globally unique identifier like a number to the operation. This number can be used to identify both the write itself and the server the write had been sent to.
- All servers have to store the write operations so it can be sent to other servers to update them.

Client side:

- Read and write operations can be sent to any server. The selection is random.
- There is no caching. Therefore we don't have to consider problems with too small caches e. g.
- Every client stores a write set with the writes committed by itself, and a read set containing the writes already seen by the clients former reads.
- A session begins with the start of the communication. Whenever the communication ends, intentionally or because of errors, the session is closed. When the client starts a communication again, a new session is created.

This paper also ignores problems with the amount of bandwidth between clients and the cloud servers. All these guarantees are bound to their session. There are no cross-session guarantees provided.

4.2.1 Eventual Consistency

Eventual consistency is the most common consistency model offered by the most providers of big cloud systems as its the easiest property to guarantee. The only thing that has to be considered is that if there are no writes for a period of time the distributed data will be consistent for all servers so that a read will return the latest update. Clients can send updates to any server. The update is then propagated in the background using a simple gossip/epidemic protocol that assures all servers will eventually see the update. That kind of operation that updates all replicas to the newest version is called anti-entropy. This explains why weak consistency causes the system to be so robust and still available in face of network partition. Because of the simple way eventual consistency spreads operations, it may happen that updates arrive at different times and in different orders. To solve such a conflict, a resolution method is needed that can undo operations and apply the operations in the correct order again. Especially in systems where only few clients write and many clients read the data, this model of weak consistency is recommendable as there will only be a small number of conflicts that have to be resolved. But in the end eventual consistency guarantees that all replicas converge to identical copies. But as presented in Wada et al. (2011) [23] this time is relatively short for current cloud services (see Figure 1). They ran tests with 26,500 writes and 3,975,000 reads for different consistency levels. The results show that for eventual consistency, about 33% reads already return consistent data from 0ms to 450ms. After that there is a sharp jump to 98% and after 507ms the data is already 100% consistent. Although the results of that test show that an eventually consistent system may reach a consistent state relatively quick, it theoretically can return any value that was written in the past. Because of that, it is difficult for software engineers to develop some applications using cloud storage systems with eventual consistency. Therefore, additional constraints will be introduced with the next models, based on session guarantees.

4.2.2 Monotonic Reads

The idea behind the monotonic reads model is relatively simple. It guarantees that if a client reads the same data item several times, the returned value of a read will not be older than the value of the former read. In other words, the client reads a data item and the server returns value x. Whenever the client reads the same data item again the returned value will be the same or if it has changed already, the newer value might be returned. So the client will never see a value that is older than the already seen value. With increasing time the cloud storage is more and more up-to-date. This can be done by only allowing the client to read from replicas that already have applied all previous updates seen by the first read.

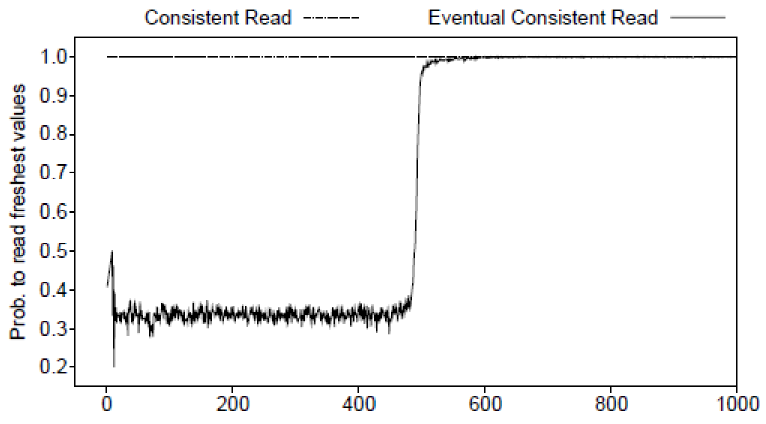

Figure 1: Time elapsed from completing write until consistent read . Wada (2011) Figure 2 [23]

To achieve monotonic read the client must send its read set to the server to which the request is directed to. The read set stores all the writes the client saw in former reads of the session. These writes are relevant for the reads in the same session. The server is then able to check if it has seen all writes the client already saw by checking the identifiers of the operations in the clients read set. If there are operations missing on server side, the request can be forwarded to the other servers that carried out the updates. Remember that the identifier of an operation can be used to determine the server which applied the update. Therefore, the server that noticed the missing update is even able to ask the specific servers for their updates in order to update its own database. There are cases where there are essential writes at the server responding to the client's request that were unseen by the client. But the client's read set is extended with these updates.

4.2.3 Monotonic Writes

This consistency model ensures that a clients writes will be applied to the database in the same order as they were committed. There may be writes from other clients in between, but it is guaranteed that new writes are done after the previous ones made within the same session. To make that possible the client is only allowed to write to replications that include all previous writes from the same session.

It's important to at least provide monotonic writes as it is quite hard for software engineers to write applications on top of a cloud without that guarantee.

The implementation idea for monotonic write consistency is pretty much the same as for monotonic reads. Just swap writes and reads. The client sends its write and write set to the server to which the write is directed to. The client's write set stores all the writes committed by itself in the same session. With the write set the server is able to check if all writes of the client have already been performed at its replica. The writes must have been performed in the right order as well. If the server is not up-to-date it can apply the missed writes from the client's write set or request them from other replicas. When the operation is applied the write is added to the write set of the performing client.

4.2.4 Read Your Writes

Read your writes consistency is roughly a mixture of both of the previous models. It states that the effects of all writes done in the same session have to be visible for a subsequent read within this session. So the client is only allowed to read from servers containing all write changes from that session. There may be other writes from other clients, too. Therefore, the returned value is not necessarily the up-to-date-value. Only if the globally last write of the data item has been done by the same client in the same session the read sees the latest value.

Implementing read your writes consistency uses the same approach as monotonic writes except that the writes from other clients do not have to be considered.

4.2.5 Writes Follow Reads

This consistency model is build upon the monotonic read consistency model. Tanenbaum et al. [18] defined the writes follow reads property as follows:

> ”A write operation by a process on a data item *x* following a previous read operation on *x* by the same process is guaranteed to take place on the same or a more recent value of *x* that was read.”

That means that a write is never applied to an older version of a data item than the former read operation returned. The writes are always applied to the read value, or if the last change has been sent from an other client it may be applied on this newer version.

Instead of only reading from a replication that contains the same or a newer value seen in the last read similar to monotonic reads, it is only allowed to write to those replicas that contain the same or a newer value seen by the last read.

The implementation for writes follow reads consistency uses the same pattern. Together with the write the client's read set is sent to a server. This server can use the set to update its own database if some writes are missing. After the operation is finished, the identifiers of the write and the writes applied from the read set are added to the write set.

The described way of implementation of any model is a simple approach and therefore not a perfect solution. For example, applying missing updates during a request unnecessarily delay the response time for the client. Therefore, it's better to redirect the request to the right server for the operation instead of updating a random one and performing the operation there. The server containing the outdated replica can then update its data in the background.

5 Improving Consistency Cost Trade-Off

As a famous example, Facebook uses a cloud system called Cassandra [1] using eventual consistency to provide high performance, scalability, and availability. But as the experiments in Wada et al. (2011) [23] show, the stale read rate under heavy access is higher than 60%. As an example, 1.39 billion people use Facebook each month and 890 million people daily [7]. Therefore, the term ”heavy access” can be assumed to be valid at all times. Because 60% of stale reads is not enough for such a big company they search for ways to improve the usage of weak consistency. Not all data or all operations need to be treated with the same level of consistency. Therefore, several methods were devised that use these possibilities to improve the trade-off between consistency and cost in money and performance.

The following approaches can be developed on top of an existing cloud storage database providing eventual consistency.

[1] http://cassandra.apache.org/

5.1 Static Consistency

Instead of providing only one consistency model for the whole cloud service, the data is statically separated into different groups. These groups are treated by different levels of consistency. For a web-shop, as an example, there exists data that has to be handled with a stronger consistency like transaction information. Other groups of data like profile settings of users won't be changed or accessed very often and therefore only need weak consistency. Also, there is data that has to be handled between those two groups. For example, the stock amount of products is accessed and changed by many users simultaneously. Therefore, additional properties have to be provided and a model like causal consistency is needed.

5.2 IDEA - Infrastructure For Detection-Based Adaptive Consistency Guarantees

Another approach that tries to improve performance and consistency is presented in Lu et al. (2007) [15] and Lu et al. (2008) [16]. They suggest to let the user decide which level of consistency is needed. Their system design, named IDEA, then guarantees that all requests will be processed with at least this level. As the users might not know what consistency is needed, they can give feedback to the system if they are not satisfied or if their needs have changed. The system then tries to adapt consistency to fulfill the demands of the user.

They improved the achieving of consistency, performance and availability by quickly detecting and resolving inconsistency among the replicas of the cloud storage. They implemented their system as a two-layer overlay for every data item. All nodes that frequently update this item are in the top layer. The other nodes form the bottom layer. To check inconsistency, they first look at the top layer which is where 95% of all inconsistencies can be found. Because the number of nodes in the top layer is relatively small, this check can be done very quickly.

To detect inconsistency, version vectors are utilized. Each data item in the cloud has its own version vector on each replica. The standard version vector stores information about which client updated it how often. They expand these vectors by a timestamp, some metadata showing the difference between the replicas and a triple of errors: <numerical error, order error, staleness>. The numerical error is the difference in the metadata values. The order error shows the difference between the amount of updates, and the staleness is the difference of the timestamps of the replicas. If the version vectors of different replicas are not the same, their data is inconsistent.

The consistency check of the replicas in the IDEA system starts frequently in the background by itself. It is also initiated whenever a user commits a write, as every write causes the other replicas to become inconsistent, and by some reads. For example, if a client requests a data item it never read, the system checks if it is consistent enough to satisfy the users demands. If the consistency level is good enough for the requesting client, then there is no resolution happening until the system is under low load so the resolution process won't influence the performance of the systems work. But in the case that the consistency provided does not satisfy the users demands, then the nearest replica sends a message to all top layer nodes that it wants to resolve a conflict. If there is no other resolution in progress all top layer nodes acknowledge this request and the initiating node starts the resolution process. The acknowledge is used to avoid having more then one resolution run at the same time. If one is already running the requesting node waits a random amount of time before it sends the request again.

In the resolution process the system chooses a so called reference consistent state. In other words it determines which replica is set to be the correct state.

There are three different ways to chose a node:

- Invalidate Both: Independent of their current values both replicas undo the applied updates and are rolled back to the last known consistent state.
- ID Based: Each node is assigned an unique ID. The node with the larger ID wins and becomes the reference state.
- Priority Based: Every node gets a priority. The node with the highest priority is chosen as the reference node.

After the reference status is determined the errors can be calculated. As already mentioned above, the numerical error is calculated by the metadata values, the order error is the difference between the number of updates and the staleness is the timestamp difference. These errors can be used to calculate the consistency using the maximal values of the elements of the triple and the errors as follows:

$$Consistency = \frac{\text{max\_num} - \text{num\_error}}{max\_num} \times num\_weight$$

$$+ \frac{\text{max\_order} - \text{order\_error}}{max\_order} \times order\_weight$$

$$+ \frac{\text{max\_staleness} - \text{staleness\_error}}{max\_staleness} \times staleness\_weight$$

The weights can be used to uniquely customize the weight of the errors depending on the wishes of the user. When the reference consistency state node is chosen the missing updates can be determined. The requesting node can then ask the top nodes and other involved nodes for those updates to get an up-to-date replica.

As the users may not be able to tell the system the needed consistency level, there are different methods to adjust the consistency:

- Hint-Based: The consistency is given by the user before the application starts. In this case the user needs to know how much consistency is needed.
- On-Demand: The wanted consistency can be adjusted by the client any time if it's not contented or its needs have changed.
- Fully Automatic: The system automatically determines the needed consistency level based on the balance between consistency and the message overhead. This maximal overhead is used to determine the frequency of the background resolution processes.

As the users may not be able to find the right setting of the three error weights these values can automatically be adapted by the system by giving feedback about the satisfaction as well.

The next two approaches try to improve the IDEA design by preventing inconsistency instead of trying to quickly detect and resolve it. They try to adjust the consistency level during runtime, too. The purpose of this method is to minimize the monetary costs for the provider of the cloud storage system. It is necessary to find a balance between the consistencies. Using too weak consistency levels leads to increased penalty costs because the servers have to be updated. This causes inter replication network traffic that cost time and money. On the other side, providing strong consistency where it is not needed involves unnecessary propagation of updates among the servers, which causes avoidable costs of time and money. There are different approaches for that solution. This paper presents two of them in the following subsections.

5.3 Consistency Rationing

The first approach explained in Kraska et al. (2009) [13] assumes that consistency is defined on the data-items in the cloud instead at the transaction level. Therefore, consistency models can be assigned to the data depending on the costs for operations on them. It divides the server data into three different priority groups (A, B, C) treated with the proper consistency model.

- A: Data that needs to be up-to-date and therefore has to be treated by strong consistency. If inconsistency occurs there would be a large penalty cost to resolve that value discrepancy.
- C: As temporally inconsistency is acceptable, this group of data can be treated with weak consistency such as read your writes consistency. Data in this group is never read or written concurrently. If inconsistency occurs almost no penalty cost in monetary or administrative way has to be expected.
- B: This group is in between of the other ones. Data in this group is accessed concurrently by many clients. Therefore, the needed consistency level can change anytime depending on the current situation.

The focus in this paper is on the B category, as category A and C provide nothing new compared to the idea of statically assigning different consistency models. As too high consistency may avoid penalty costs but implicates higher operation costs, less availability and because too relaxed consistency may lead to high penalty costs but provides cheaper operations, the system has to find the right trade-off between the costs and the consistency. In Kraska et al. (2009) [13] the price of inconsistency is calculated as the percentage of incorrect operations causing overhead for resolving the conflict leading to monetary cost and lower user experience. They monitor the data stored in the cloud system and use temporal statistics to find that balance that minimizes the costs for communication and penalty. They define different policies describing when to switch consistency level:

- General policy: Calculates the probability for inconsistency by measuring the frequency of operations on the data items of the cloud storage.
- Time policy: switches consistency after a certain amount of time
- Fixed Threshold policy: depends on the value of the data
- Demarcation policy: Uses several values to consider a switch on the base of the demarcation protocol [2]
- Dynamic policy: Probabilistic guarantee depending on the frequency of updates like general policy, but also using the data values. In simple terms it uses the probability that an update will cause the value to exceed its constraint.

The problem with most of these policies is finding suitable values for the thresholds and limits that determine when to change the consistency level to high or low.
In the general policy this can be described with a simple mathematical inequation saying: Switch to strong consistency whenever the savings from weak consistency are lower than the costs of inconsistency and switch back to weak consistency if the savings are higher than the penalty costs.
For time and value fixed policy the threshold has to be found by experimenting until a proper trade-off between runtime cost and penalty cost is found.
The dynamic policy doesn't have this problem. Statistics for any data item in the cloud are set up and used to adjust the thresholds during runtime.

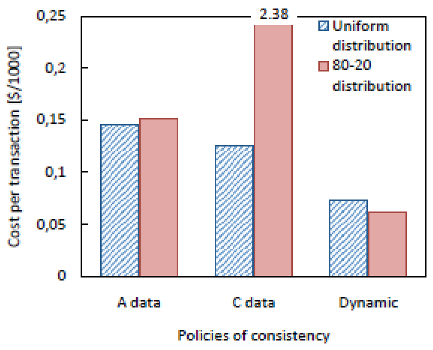

Figure 2: cost per transaction [$/1000]. [13] Figure 3

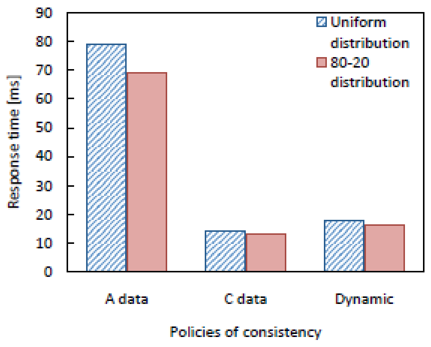

Figure 3: Response time [ms]. [13] Figure 5

In Kraska et al. (2009) [13] they used the TPC-W benchmark [21] to check their implementation of the consistency rationing concept. Using the dynamic policy for adjusting the consistency level emerged as the best solution providing performance and low costs as it's dynamic and reactive. They ran the test with a 80-20 rule distribution as explained in Gray et al. (1994) [12] and a uniform one. Some of their results are shown in Figure 2 and 3.

Figure 2 shows that the dynamic adaption of consistency ("Dynamic") is much more efficient than both, constant weak ("C data") and constant strong consistency ("A data") for both distributions. This chart also shows that weak consistency is more efficient than strong efficiency. But in the biased 80-20 distribution it is way more expensive due to the high penalty costs.

Figure 3 shows the response times for the different consistencies. It illustrates how much faster weak consistency is than strong consistency for both distributions regarding response time showing the strength of weak consistency in cloud computing.

As a conclusion it is proved that dynamic consistency is way more effective than static consistency as it clearly reduces response time and monetary cost of transactions. The improvements are so big that the costs of rationing the data into different consistency categories can be neglected.

5.4 Consistency Adaption With Bismar

As defining consistency on data level causes an overhead for large data items, the second approach described in Chihoub et al. (2013) [6] defines consistency at transaction level. They introduce an economical consistency model called Bismar which changes the consistency level at runtime. This model uses the current operation frequency and the network latency to determine the rate of stale reads and relative costs. They segment the costs into three different groups:

- Computing: Charged for virtual machine hours needed to run the operations on the cloud.
- Storage: Charged for amount of storage leased per month and the number of I/O requests to or from the storage. Increasing with a higher number of replicas and requests.
- Network: Network traffic among replicas for updating and repair. Grows with higher consistency and higher amount of replicas.

Due to cheap storage the computing part takes up to 90% of the costs while storage and networking costs only contribute 9% and 0.4%.

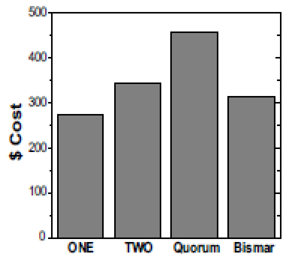

Figure 4: Monetary Cost

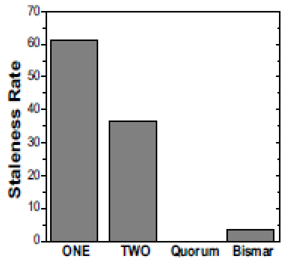

Figure 5: Staleness

To have values as a reference they ran tests to determine the costs for different consistency levels from eventual consistency (”ONE”) over ”TWO” and ”QUORUM” to strong consistency (”ALL”). Their results show that weak consistency is almost 48% cheaper than strong consistency.

To control their Bismar model they introduced a metric called consistency-cost efficiency, showing how much a consistency level costs. Their model selects the consistency providing the maximum consistency-cost efficiency to minimize the monetary costs. To determine the needed level of consistency they calculate a probabilistic estimation model from the current read/write frequency and the network latency. The arrival of transactions is considered as a Poisson process.

In the end they came up with a formula that can be used to compute the consistency-cost relation depending on the consistency level. The only thing their system has to do is to continuously calculate the possible values for that metric and choose the consistency level delivering the maximum efficiency.

As shown in Figure 4 the results of their tests reveal that the costs of the Bismar model are slightly higher than the costs of eventual consistency (”ONE”). But the number of stale reads in Bismar (3%) is much lower than in eventual consistency (61%).

As these two approaches show the dynamic adaption of consistency during runtime can obviously improve the monetary costs and the response time in cloud storage systems.

6 Conclusion

This paper showed that there are many different ways to handle consistency in cloud storage systems. Eventual consistency is the most important and is used most often. It is easy to provide and it increases performance, availability and reliability. As mentioned in Fuertes et al. (2012) [17] and Vogels (2009) [22] the system even provides these properties in the face of network or system errors causing network partition. The system continues working. When the error is resolved the system may have several different states of the data as some replicas may have continued running isolated from the rest of the cloud storage system. Eventual consistency easily can resolve these states and therefore deserves its status as the most popular consistency model.

In this paper I showed that eventual consistency alone is not a good way to run a cloud storage system. Because of the unpredictable results of requests, the use of such a weak consistency is rather unhandy. Especially for the software developers that have to write the software using the cloud.

We learned that there are several additional constraints for eventual consistency. These constraints provide some additional guarantees to the returned data from a request and the order of execution of the write operations from the same client in the same session. These so called session guarantees significantly make it easier for developers to write applications that deal with the cloud data. The choice of the right model strongly depends on the application it is designed for. As these different session models can be combined arbitrary, the variety of possible consistency models is huge.

In the second part this paper showed that a cloud storage system is not limited to only provide one level of consistency. Different kinds of data and different kinds of transactions can be treated with different levels of consistency. This simple idea facilitates some great improvement for the performance of the cloud system.

Some works like Lu et al. (2007) [15] or Lu et al. (2008) [16], Kraska et al. (2009) [13] and Chihoub et al. (2013) [6] pursued this idea of using more than one consistency level in the same cloud even more and developed some novel approaches for handling consistency. They all try to adapt the level of consistency during runtime and try to adjust the consistency to the needs of the user. Their intention is to find the weakest possible level of consistency that satisfies the client's demands. Therefore, they automatically use the method with the least costs of money possible.

In Lu et al. (2007) [15] and Lu et al. (2008) [16] they try to detect and resolve inconsistency as fast as possible and then change the level of consistency to prevent repeating needs for resolution processes. Kraska et al. (2009) [13] and Chihoub et al. (2013) [6] go one step further and predict the costs of a operation in respect to a certain level of consistency. Then they chose the model with the best efficiency, calculated by the costs per consistency.

The results of the tests run on the different approaches show the obvious improvement in operation costs and response time. They make clear that such novel ideas can heavily boost the performance of commercial cloud storage systems.

References

[1] Amazon (2008): *Simple Storage Service S3*. Available at `http://aws.amazon.com/s3/`.

[2] Daniel Barbar-Mill & Hector Garcia-Molina (1994): *The demarcation protocol: A technique for maintaining constraints in distributed database systems*. *The VLDB Journal* 3(3), pp. 325–353, doi:10.1007/BF01232643. Available at `http://dx.doi.org/10.1007/BF01232643`.

[3] Matthias Brantner, Daniela Florescu, David Graf, Donald Kossmann & Tim Kraska (2008): *Building a database on S3*. In: *Proceedings of the 2008 ACM SIGMOD international conference on Management of data - SIGMOD '08*, Association for Computing Machinery (ACM), doi:10.1145/1376616.1376645. Available at `http://dx.doi.org/10.1145/1376616.1376645`.

[4] Eric A. Brewer (2000): *Towards robust distributed systems (abstract)*. In: *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing - PODC '00*, Association for Computing Machinery (ACM), doi:10.1145/343477.343502. Available at `http://dx.doi.org/10.1145/343477.343502`.

[5] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes & Robert E. Gruber (2006): *Bigtable: A Distributed Storage System for Structured Data*. In: *Proceeding of OSDI*, Google, pp. 205–218. Available at `http://static.googleusercontent.com/media/research.google.com/de//archive/bigtable-osdi06.pdf`.

[6] H.-E Chihoub, S. Ibrahim, G. Antoniu & M. S. Perez (2013): *Consistency in the Cloud: When Money Does Matter!* In: *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, Institute of Electrical & Electronics Engineers (IEEE), doi:10.1109/ccgrid.2013.40. Available at `http://dx.doi.org/10.1109/CCGrid.2013.40`.

[7] Deborah Crawford (2015): *Facebook Q4 2014 Earnings Call Transcript* Available at `http://files.shareholder.com/downloads/AMDA-NJ5DZ/0x0x805927/BA45C6A6-6D93-4DF6-BA81-D3C0B352FF4E/FacebookQ42014EarningsCallTranscript.pdf`.

[8] Srgio Esteves, Joo Silva & Lus Veiga (2012): *Quality-of-Service for Consistency of Data Geo-replication in Cloud Computing*. In Christos Kaklamanis, Theodore Papatheodorou & PaulG. Spirakis, editors: *Euro-Par 2012 Parallel Processing*, *Lecture Notes in Computer Science* 7484, Springer Berlin Heidelberg, pp. 285–297, doi:10.1007/978-3-642-32820-6_29. Available at `http://dx.doi.org/10.1007/978-3-642-32820-6_29`.

[9] Alan Fekete (2009): *Weak Consistency Models for Replicated Data*. In LING LIU & M.TAMER ZSU, editors: *Encyclopedia of Database Systems*, Springer US, pp. 3451–3455, doi:10.1007/978-0-387-39940-9_1537. Available at `http://dx.doi.org/10.1007/978-0-387-39940-9_1537`.

[10] AlanD. Fekete & Krithi Ramamritham (2010): *Consistency Models for Replicated Data*. In Bernadette Charron-Bost, Fernando Pedone & Andr Schiper, editors: *Replication*, *Lecture Notes in Computer Science* 5959, Springer Berlin Heidelberg, pp. 1–17, doi:10.1007/978-3-642-11294-2_1. Available at `http://disi.unitn.it/~montreso/ds/papers/replication.pdf`.

[11] Seth Gilbert & Nancy Lynch (2002): *Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services*. *SIGACT News* 33(2), p. 51, doi:10.1145/564585.564601. Available at `http://dx.doi.org/10.1145/564585.564601`.

[12] Jim Gray, Prakash Sundaresan, Susanne Englert, Ken Baclawski & Peter J. Weinberger (1994): *Quickly generating billion-record synthetic databases*. *ACM SIGMOD Record* 23(2), pp. 243–252, doi:10.1145/191843.191886. Available at `http://dx.doi.org/10.1145/191843.191886`.

[13] Tim Kraska, Martin Hentschel, Gustavo Alonso & Donald Kossmann (2009): *Consistency rationing in the cloud: Pay only when it matters*. *Proc. VLDB Endow.* 2(1), pp. 253–264, doi:10.14778/1687627.1687657. Available at `http://www.vldb.org/pvldb/2/vldb09-759.pdf`.

[14] Lamport (1979): *How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs*. *IEEE Transactions on Computers* C-28(9), pp. 690–691, doi:10.1109/tc.1979.1675439. Available at `http://dx.doi.org/10.1109/TC.1979.1675439`.

[15] Yijun Lu, Ying Lu & Hong Jiang (2007): *IDEA:*. In: *Proceedings of the 16th international symposium on High performance distributed computing - HPDC '07*, Association for Computing Machinery (ACM), doi:10.1145/1272366.1272401. Available at `http://digitalcommons.unl.edu/cgi/viewcontent.cgi?article=1061&context=csetechreports`.

[16] Yijun Lu, Ying Lu & Hong Jiang (2008): *Adaptive Consistency Guarantees for Large-Scale Replicated Services*. In: *2008 International Conference on Networking, Architecture, and Storage*, Institute of Electrical & Electronics Engineers (IEEE), doi:10.1109/nas.2008.64. Available at `http://www.researchgate.net/publication/4359882_Adaptive_Consistency_Guarantees_for_Large-Scale_Replicated_Services`.

[17] M.I. Ruiz-Fuertes, M.R. Pallard-Lozoya & F.D. Muoz-Esco (2012): *Consistency in Scalable Systems*. In Robert Meersman, Herv Panetto, Tharam Dillon, Stefanie Rinderle-Ma, Peter Dadam, Xiaofang Zhou, Siani Pearson, Alois Ferscha, Sonia Bergamaschi & IsabelF. Cruz, editors: *On the Move to Meaningful Internet Systems: OTM 2012*, *Lecture Notes in Computer Science* 7566, Springer Berlin Heidelberg, pp. 549–565, doi:10.1007/978-3-642-33615-7_7. Available at `http://dx.doi.org/10.1007/978-3-642-33615-7_7`.

[18] Andrew Tanenbaum & Maarten Van Steen (2006): *Distributed systems: principles and paradigms*, second edition, chapter VII, pp. 273–320. Pearson Education, Upper Saddle River, NJ. Available at `https://vowi.fsinf.at/images/b/bc/TU_Wien-Verteilte_Systeme_VO_%28G%C3%B6schka%29_-_Tannenbaum-distributed_systems_principles_and_paradigms_2nd_edition.pdf`.

[19] D.B. Terry, A.J. Demers, K. Petersen, M.J. Spreitzer, M.M. Theimer & B.B. Welch (1994): *Session guarantees for weakly consistent replicated data*. In: *Proceedings of 3rd International Conference on Parallel and Distributed Information Systems*, Institute of Electrical & Electronics Engineers (IEEE), doi:10.1109/pdis.1994.331722. Available at `http://dx.doi.org/10.1109/PDIS.1994.331722`.

[20] Doug Terry (2013): *Replicated data consistency explained through baseball*. *Communications of the ACM* 56(12), pp. 82–89, doi:10.1145/2500500. Available at `http://research.microsoft.com/pubs/157411/ConsistencyAndBaseballReport.pdf`.

[21] TPC (2002): *TPC-W Benchmark 1.8* Available at `http://www.tpc.org/tpcw/spec/tpcw_v1.8.pdf`.

[22] Werner Vogels (2009): *Eventually consistent*. *Communications of the ACM* 52(1), p. 40, doi:10.1145/1435417.1435432. Available at `http://dx.doi.org/10.1145/1435417.1435432`.

[23] Hiroshi Wada, Alan Fekete, Liang Zhao, Kevin Lee & Anna Liu (2011): *Data consistency properties and the trade-offs in commercial cloud storages: the consumers perspective*. In: *Fifth Biennial Conference on Innovative Data Systems Research, Online Proceedings*, Lecture Notes in Computer Science, Asilomar, CA, USA, pp. 134–143. Available at `http://www.cidrdb.org/cidr2011/Papers/CIDR11_Paper15.pdf`.