Consistency in Distributed Storage Systems
An Overview of Models, Metrics and Measurement Approaches

David Bermbach and Jörn Kuhlenkamp

Karlsruhe Institute of Technology,
Karlsruhe, Germany
firstname.lastname@kit.edu

Abstract. Due to the advent of eventually consistent storage systems, consistency has become a focus of research. Still, a clear overview of consistency in distributed systems is missing. In this work, we define and describe consistency, show how different consistency models and perspectives are related and briefly discuss how concrete consistency guarantees of a distributed storage system can be measured.

Keywords: Consistency, Distributed Systems

1 Introduction

In distributed storage systems replication can be used to increase durability and availability of data as well as to enable fault tolerance and low latencies for distributed clients. This comes with a price, though, as multiple copies add the burden of keeping replicas identical. With the advent of the Internet and lately Cloud Computing, replication has become the number one mechanism to deal with scalability issues, load variance and large numbers of parallel requests. This in turn has brought *consistency* to the attention of both businesses and researchers as there is now a multitude of storage systems each offering different consistency guarantees which cannot be easily measured. As the CAP theorem and the PACELC model[18, 1] mandate, there are direct trade-offs between consistency and availability as well as consistency and request latency of a replicated storage system and each system chooses a different spot on the continuum of consistency guarantees between strict consistency and no consistency.

There is much work on consistency models, their implementations and measurements, but the relationships between models and measurement approaches is not always clear. Furthermore, researchers from the database community have an entirely different understanding of consistency than researchers from the distributed systems community.

In this work, we try to shed some light on these issues. We start with a brief description of what consistency means in both research communities before taking the distributed systems view and analyzing different perspectives on consistency as well as the relationship between various consistency models. Next, we combine this with our previous work on consistency benchmarking and describe continuous metrics and setups for experimental consistency measurements.

The main contribution of this work is, hence:

– A comprehensive overview of consistency models and guarantees, including discussions why particular guarantees are useful in different use cases.
– An analysis how consistency models can be ordered by their strictness and how client-centric guarantees relate to data-centric models.
– A discussion of different metrics to describe consistency guarantees as well as a brief analysis of existing benchmarking approaches.

2 Definitions

The term consistency is derived from the latin word *consistere* which means "standing together" or also "stopping together". Hence, consistency generally describes relationships between items that are somehow connected. When considering consistency of data, a consistent state requires that all relationships between data items and replicas are as they should be, i.e., that the data representation is correct. This focus on the correctness can be seen in both the database as well as the distributed systems community – but on different levels.

2.1 Database Systems

Researchers from the (relational) database community focus on consistency in the context of ACID guarantees. A set of operations is abstracted into a transaction that will execute entirely or not at all (atomicity). If it executes, its changes will be permanently visible (durability). Multiple transactions may execute concurrently but use suitable mechanisms to show the same behavior as if all transactions were executed according to some global serializable order. This can be done in various degrees of isolation[1] ranging from true *serializability* (which is rarely offered in actual products [5]) to *read uncommitted*. Transactions guarantee that the database always adheres to a global schema (independent of whether the database is replicated or not) where all integrity constraints, a set of conditions and requirements, are observed [15] (consistency).

Hence, the consistency focus of database systems is on the relationships between data items and the overall correctness of the entire database. They can be guaranteed in a distributed setting but it is expensive to do so as consistency and isolation are typically guaranteed via locking mechanisms which create an extensive communication overhead in a distributed setting. This is also caused by the fact, that transactions were intended as "a simple programming abstraction to cope with concurrent executions, rather than to address the challenges of a distributed setting." [33].

[1] Isolation describes the degree to which concurrent transactions are aware of each other, e.g., by accessing the same data items.

2.2 Distributed Systems

Researchers from the distributed systems community investigate state shared by multiple replicas, i.e., several copies of a datum exist which may or may not be identical. Executions of operations on these replicas may read or change the state at one or more replicas. Essentially, "a consistency criterion [or consistency model] defines which executions of a distributed system are considered correct" [21], i.e. which order of operations leaves the data in a correct state.

In the following, we will focus on the distributed systems perspective on consistency which can be defined as follows:

> A system is in a consistent state, if all replicas are identical and the ordering guarantees[2] of the specific consistency model are not violated.

3 Perspectives and Consistency Models

3.1 Perspectives on Consistency

In a distributed storage system there are two perspectives on consistency [34]: the provider (i.e., the entity responsible for the deployment and operation of a storage system) views the internal state of the system. His focus is on the synchronization processes among replicas and the ordering of operations. Hence, this perspective is called data-centric. The other perspective is the one of a client of the storage system. Here, a client refers to the process that interacts with the storage system which can be any kind of application, middleware or even software running on the end user's machine or mobile device. This client-centric perspective views the system from the outside as a black box. Hence, its focus is on the guarantees of the distributed storage system that could also be captured as part of a service level agreement (SLA). Based on these two perspectives, there are various consistency models either taking a client-centric or data-centric perspective. Still, there is a relation between those models so that some models and combinations thereof mean exactly the same thing while still bearing different names.

Both perspectives have advantages and disadvantages for the analysis of consistency guarantees – depending on the issue of interest. While data-centric consistency models do not address concrete implementations or algorithms, they certainly describe ordering properties that allow to develop a corresponding synchronization protocol. The downside is that data-centric consistency models are not really helpful to application developers. Client-centric consistency models describe the effects of such a synchronization protocol. While this is very helpful to an application developer, it ignores completely how this could be implemented, i.e., what internal synchronization protocols might deliver such a guarantee.

[2] Ordering guarantees in this context describe how requests may be reordered on different replicas.

3.2 Consistency Models and Implementations

Both data-centric and client-centric consistency guarantees have two dimensions: ordering and staleness[3]. Staleness describes how much a given replica is lagging behind, either expressed in terms of time (t-visibility) or versions (k-staleness)[4]. Again, k-staleness is a function of t-visibility and the update patterns of the application so that, for application-independent information, t-visibility suffices to characterize the staleness guarantees of a storage system. Low, bounded staleness values can often be tolerated by applications as long as the corresponding real-world events would have the same or higher staleness values without an IT system. For example, when person A wires money to person B, the account of A will be charged right away. Person B in contrast might not be credited for some time. In the EU this time window is limited to three days which is far longer than any replica synchronization protocol might take. Hence, small staleness values will often not be noticed.

Ordering on the other hand is more critical. In a setting with strict consistency, all requests must be executed on all replicas in their chronological order which is hard to implement in distributed databases due to clock synchronization issues as the replica servers might disagree on the actual chronological order of events. The standard database mechanism of locking offers poor performance levels in a distributed setting. Based on this, data-centric consistency models exist that relax certain ordering requirements while keeping those that are essential to applications. These models can be ordered by the "strictness" of their guarantees. Client-centric consistency models take a different approach: While there will almost certainly be cross effects between the models, the guarantees itself are disjunct in their promises and complement each other. We will start by describing client-centric consistency models before continuing to data-centric models and how those two are related.

Client-centric Consistency The first model, *Monotonic Read Consistency (MRC)*, guarantees that a client that has read a version n will thereafter always read versions $\geq n$ [34, 37]. This is helpful as from an application perspective data visibility might not be instantaneous but versions at least become visible in chronological order, i.e., the system never "goes backward" in time. For example, imagine person B from our bank scenario above. If this person first sees the credited amount on his bank account statement and then tries to transfer the money to a person C which fails due to "insufficient funds", this will at least cause severe customer irritation if not more.

Read Your Writes Consistency (RYWC) guarantees that a client that has written a version n will thereafter always be able to read a version that is at least as new as n, i.e., $\geq n$ [34, 37]. This helps, for example, to avoid user irritation when person A checks his bank account statement, does not see the

[3] Yu and Vahdat[39] propose an additional dimension *numerical error* to describe replica differences based on the semantics of the respective data item. From our point of view, this is first not always applicable and second a numerical error is essentially a function of ordering, staleness and application access patterns.

transaction and consequently wires the same amount of money again. Generally, RYWC avoids situations where a user or application issues the same request several times because it gets the impression that the request failed the first time. For idempotent operations reissuing requests causes only additional load on the system, while reissuing other requests will create severe inconsistencies.

Monotonic Writes Consistency (MWC) guarantees that two updates by the same client will be serialized in the order that they arrive at the storage system [34, 37]. This is useful to avoid seemingly lost updates when an application first writes and then updates a datum but the update is executed before the initial write and is, thus, overwritten. In the bank scenario above, person A might have corrected the account number of person B before finalizing the transfer. If MWC is not guaranteed, the money might end up in the wrong account.

Write Follows Read Consistency (WFRC) guarantees that an update following a read of version n will only execute on replicas that are at least of version n [34]. This, also, helps against seemingly lost updates where the update is overwritten by a delayed update request for versions $\leq n$. This model essentially extends MWC guarantees to updates by other clients that have at least been seen.

In NoSQL and Cloud storage systems, these client-centric properties are typically not guaranteed explicitly. Benchmarks can be used to determine the probability of violations or to measure the second dimension staleness [8, 38].

Data-centric Consistency In this section, we will present data-centric consistency models ordered by the strictness of their guarantees and discuss for each model how it can be translated into a client-centric consistency model. As already discussed, there are two consistency dimensions: staleness and ordering. The following consistency models (apart from Linearizability) do *not* consider staleness [34]. In fact, increasing strictness of ordering guarantees often leads to higher staleness values as updates may not be applied directly but are required to fulfill dependencies first (e.g.,[3]).

The lowest possible ordering guarantee is typically described as *Weak Consistency* [34, 37]. As the name states, guarantees are very weak in that they do not really exist. Essentially, weak consistency translates to a colloquial "replicas might by chance become consistent". While an implementation may or may not have a protocol to synchronize replicas, a typical usecase can be found in the context of a browser cache: it is updated from time to time but replicas will rarely (if ever) be consistent. As Weak Consistency does not provide any ordering guarantees at all, there is no relation to client-centric consistency models.

Eventual Consistency (EC) is a little stricter. It requires convergence of replicas, i.e., in the absence of updates and failures the system converges towards a consistent state. Updates may be reordered in any way possible and a consistent state is simply defined as all replicas being identical [34, 37]. EC is very vague in terms of concrete guarantees but is very popular for web-based services. Most NoSQL systems implement EC [16, 11, 26, 17].

In terms of client-centric consistency guarantees, EC often fulfills these guarantees for a majority of requests but does not guarantee to do so. As an example, Amazon S3[4] currently delivers MRC for about 95% of all requests whereas it still violated MRC in about 12% of all requests in 2011 [8].

While there are certainly some usecases where EC cannot be applied, it often suffices as the real world itself is inherently eventually consistent. The difference is, that more conflict resolution is necessary at the application layer [16] requiring a higher skill set from application developers. Instead of pessimistically locking data items "guesses and apologies" are used [22].

Causal Consistency (CC) is the strictest level of consistency that can be achieved in an always available storage system [30] based on the tradeoffs of the CAP theorem [18]. In a causally consistent storage system, all requests that have a causal relationship to another request must be serialized (i.e., executed) in the same order on all replicas while unrelated requests may be serialized in arbitrary order. A request $r2$ causally depends on a request $r1$

- if both requests are issued by the same client and $r1$ was received at the storage system before $r2$,
- if $r2$ is a read that returns the result of $r1$ which is an update or
- if there is a transitive relation between both requests [34, 37, 9].

Of course, CC captures potential causality so that systems like COPS [29] have to evaluate large dependency trees before applying an update. This both adds an overhead and increases staleness as updates cannot become visible right away. Bailis et al. [3] propose to minimize this impact by having the application explicitly define dependencies that need to be considered. A typical implementation uses vector clocks to identify (potential) causal dependencies.

CC can also be defined via the client-centric guarantees discussed above: If all four are fulfilled, the system is causally consistent [9]. It is also possible to create the client-side illusion of CC with the combination of version caching and vector clocks [7].

As Guerraoui and Hari point out, CC does not require replica convergence [21]. Convergence is only asserted when the latest update is causally dependent on all previous writes since the last idempotent replace-update[5] and staleness is bounded.

Sequential Consistency (SC) is a very strict consistency model and cannot be achieved in always available systems[6]. It requires that all requests are serialized in the same order on all replicas and that requests by the same client are executed in the order that they are received by the storage system [34]. While this model

[4] aws.amazon.com/s3

[5] i.e., some request like $x := 5$ which does not depend on any previous value.

[6] In CC only requests with causal dependencies must be executed in the same order on all replicas. For SC, this extends to all requests so that replicas need to agree on the ordering of requests for non-causally related requests. This is not possible in the presence of failures so that the system either becomes unavailable or violates its consistency model.

does not guarantee anything about the recentness of values read by clients, it mandates that all updates become visible to clients in the same order. Often, SC is described as strict consistency which is not entirely true as staleness is not addressed. But since real-world staleness values are often very small SC usually suffices even for applications seemingly requiring strict consistency.

SC could, for example, be implemented using the Paxos algorithm [27]. Generally, vector clocks that define causal relationships can be in conflict (e.g., for unrelated concurrent updates). If vector clocks are used for request ordering and an approach exists that defines a transitive, global order for all conflicting vector clocks, then a causally consistent system becomes sequentially consistent.

When focusing on client-centric consistency guarantees, the main difference between CC and SC is that WFRC becomes global in so far as reads by all clients are considered. This means that as soon as a client has seen a particular version n, all updates by other clients will only be executed on replicas that have already processed the update to version n. This guarantee can be provided as SC guarantees that all replicas execute all updates in the same order. So, once a version n has been read, it is guaranteed to have been finally serialized as that version so that any updates will be serialized with a higher version number.

Linearizability (LIN) describes what is typically meant with strict consistency. It does not only consider ordering but also staleness, i.e., it requires that all requests are ordered chronologically by their arrival time in the system and that all requests always see the effects of the preceding request. This can be visualized as all operations happening instantaneously at a single point in time and not during an interval of time [23].

LIN is hard to implement in distributed systems as there is always the issue of clock synchronization (which is necessary to determine a chronological order of requests). In practice, however, sufficiently high precision is achieved to guarantee that violations are highly improbable to occur. Furthermore, in case of violations LIN becomes SC between which applications will rarely notice a difference. While Consensus protocols can guarantee that all replicas serialize requests in the same order, they cannot guarantee that all replicas execute requests in the actual chronological order of arrival in the system. An implementation using distributed locking is likely to show poor performance.

Expressed in terms of client-centric consistency guarantees, the difference between SC and LIN is that both RYWC and MWC become global guarantees. This means that a client will always see all updates and that all writes will be executed in the (global) chronological order. MRC then also becomes global as a side effect.

Beyond the data-centric consistency models discussed here, there are a few other models (e.g., PRAM consistency [10]) which we leave out as no implementations exist and space within this paper is limited. Table 1 gives an overview of the relationship between different client-centric and data-centric consistency models. Entries “N/A” mean that the guarantee may be reached for single requests from time to time but only based on chance. In contrast, “Often” specifies that such a guarantee is fulfilled for a large number of requests. “Single

Client" describes that the guarantees from section 3.2 are fulfilled, whereas we use "Global" to describe when such a guarantee is extended to all clients at the same time.

Data-centric Model	MRC	RYWC	MWC	WFRC
Weak Consistency	N/A	N/A	N/A	N/A
Eventual Consistency	Often	Often	Often	Often
Causal Consistency	Single Client	Single Client	Single Client	Single Client
Sequential Consistency	Single Client	Single Client	Single Client	Global
Linearizability	Global	Global	Global	Global

Table 1. Relationship Between Data-centric and Client-centric Consistency Models Ordered by the Strictness of their Guarantees

Other Consistency Models Beyond the models already discussed, there are also a few other consistency models that do not quite fit the categorization used so far.

Multi-dimensional Consistency: Yu and Vahdat [39] introduce the concept of a conit, a consistency unit, which is a three dimensional vector that describes tolerable deviations from LIN along the dimensions staleness, order error and numerical error. As already mentioned, numerical error is often not applicable and semantically overlaps with staleness and order error. When ignoring numerical error, their work becomes comparable to the work of Torres-Rojas et al.(e.g., [36, 35]) who coined the term *timed consistency.* Timed consistency models are also sometimes known as delta consistency and essentially describe a combination of ordering and staleness in that the inconsistency window (defined by the time period between the commit of an update and reaching a consistent state) is bound. This means that the guarantees of a particular consistency model are not reached right away but rather after a fixed period of time Δt. If replicas fail to synchronize during that period of time, the item becomes unavailable until consistency has been reached. This is particular useful for guaranteeing Service Level Agreements (SLAs) and increases the transparency of the consistency availability trade-off.

Sadly, to our knowledge no implementations of timed consistency models exist apart from TACT [39] and the work of Krishnamurthy et al. [25] who guarantee bounds on k-staleness (based on version count). It is possible, though, to specify a *timed* version for each of the data-centric consistency models where the guarantees become visible before the specified time window is over. In that case, the models discussed above become a special case of their timed equivalent (i.e., with a time window of infinity) which also affects the timeliness of client-centric guarantees.

Coherence: In their original definition, data-centric consistency models provide ordering guarantees for all data items, i.e., in CC, for example, two updates by the same client on two different data items must be serialized in correct order. This also implies that an eventually consistent datastore can only be in a consistent state if all replicas of all data items are identical. Depending on the size of the datastore deployment this may never be the case and it is also more difficult to coordinate updates on large numbers of servers than for just a few. So, for reasons of scalability it often makes sense to provide the guarantees of the consistency model only per key. In the case of our example above, those two updates could then be executed in arbitrary order, thus, granting more flexibility to the storage system. Guarantees per key often suffice as it is then up to the application developer to persist all items, which need guarantees amongst each other, under the same key.

Those models are named coherence, i.e., eventual coherence, causal coherence, sequential coherence. It is common practice, though, to use consistency for both coherence and consistency models alike. To add some clarity, we propose to add a "per key" prefix if coherence is meant, i.e., per key CC instead of causal coherence.

Ramakrishnan [33] argues that the "unit of consistency" should also be considered as a continuum where guarantees are not only provided either for the entire data set or for just one key but also for groups of keys like, e.g., the entity groups in Google's Megastore [6].

Adaptable Consistency: Kraska et al. [24] propose Consistency Rationing where data items are in a first step clustered based on importance (e.g., for a web shop credit card numbers vs. comments on reviews) into types A, B and C. While types A and C are always handled at LIN or EC respectively, B data continuously changes its consistency requirements based on an external cost function. This means that B data is handled at LIN whenever the costs of inconsistencies exceed the cost of opportunity caused by unavailability or high latencies. Consistency Rationing could, for example, be implemented via the much older GARF library [20].

Chihoub et al. [12, 13] present approaches that allow the user to specify maximum stale read rates or a consistency cost efficiency level as part of SLAs. The system then dynamically uses different consistency levels in Apache Cassandra [26] while guaranteeing the SLAs.

Li et al. [28] propose the concept of RedBlue Consistency where operations are broken down into very small commutative suboperations that are then categorized as either red or blue meaning that they are either synchronously or asynchronously replicated while guaranteeing dependencies between suboperations. While Consistency Rationing uses different consistency levels based on the data type, RedBlue Consistency adaptively tunes the consistency level based on the kind of operation.

4 Measuring Consistency Guarantees

4.1 Continuous Consistency Metrics

According to thefreedictionary.com, a metric is "A standard of measurement". When measuring a certain aspect, a measurement always comprises a value and a corresponding unit (e.g., for the height of a building this could be the value "5" and the unit "meter"). If it is not possible to find two values which do not have any value in between them, the metric is continuous. Otherwise the metric is discrete. An example for a continuous metric would be the height of a person, whereas clothing sizes are an example for a discrete metric.

When the ultimate goal is to compare consistency guarantees of two storage systems, it is desirable to either use a continuous metric or at least use a discrete metric with a large number of potential measurement values. Otherwise, it might not be possible to rank systems according to their consistency guarantees. In the following, we will discuss metrics for data-centric and client-centric consistency. Depending on the perspective (storage system provider or application developer), different metrics may be the best fit.

Data-centric Consistency Metrics Zellag and Kemme [40] extend their previous work on transactional datastores to non-transactional datastores. They propose to build a global dependency graph based on operation logs and count cycles in the graph as a metric for "consistency anomalies". This is a discrete metric and one of their main assumptions is that the storage system guarantees at least CC which is very restrictive and does not allow to analyze consistency guarantees of most NoSQL systems which only offer EC. Table 2 lists their approach as "Anomalies".

Rahman et al. [32], Golab et al. [19] and Anderson et al. [2] at Hewlett Packard Labs also propose to build dependency graphs based on operation logs and to count cycles in the graph as a metric for consistency guarantees. They distinguish the three properties safeness, regularity and atomicity for which they each count violations. A storage system that has no cycles in its atomicity graph fulfills LIN. The other two properties also consider staleness as well as ordering. Regularity is, thus, stricter than SC whereas Safety cannot be compared to existing consistency models. Regularity mandates that "a read not concurrent with any writes returns the value of the most recent write, and a read concurrent with some writes returns either the value of the most recent write, or the value of one of the concurrent writes" [2]. Safeness in contrast relaxes the last requirement so that reads concurrent with writes may return arbitrary values. We do not believe that the latter guarantee is very helpful as it basically requires LIN for non-concurrent requests and Weak Consistency for concurrent requests. Chockler et al. [14] seem to share that opinion. Furthermore, real-world systems may or may not return the value of the most recent write but, to our knowledge, no system exists that may return values that have never been written. All three metrics can also be expressed as k-property or Δ-property (e.g., k-atomicity and Δ-atomicity) which describes the maximum number of time units or versions a particular system has been found to lag behind during a violation. This is a rather coarse-grained discrete metric. Table 2 lists their approach as "k-Atomicity", "Δ'-Atomicity" etc.

We propose to again distinguish the two consistency dimensions ordering and staleness and measure them separately. Staleness can be expressed either based on time (t-Visibility) or operation count (k-Staleness) [4]. We believe that these two (continuous) metrics are best suitable to describe data-centric staleness. It probably makes sense to aggregate them into a distribution function, i.e., a function describing the probability of a particular staleness "level". Staleness can be measured independent of concrete application workloads. For ordering on the other hand, it makes sense to mine the replicas' operation logs to determine the number of violations for each of the consistency models; i.e., in a SC system violations of LIN will be counted, in a CC system violations of SC will be measured and in an EC system violations of CC could be counted. This, obviously, highly depends on the distribution of requests regarding time, target key, originator and kind (read, insert, update, delete). Hence, for a comparison of two systems' consistency guarantees it is a hard requirement to replay exactly the same client workload which will often be problematic[7]. Ordering can then be reported as number of violations of consistency model X per unit of time. Table 2 gives an overview of data-centric consistency metrics.

Metric	Staleness	Ordering	Continuous	Discrete	Unit & Description
Anomalies	X	X	-	X	Number of cycles
k-Atomicity	X	X	-	X	Max. version lag in violation
Δ-Atomicity	X	X	-	X	Max. time lag in violation
k-Regularity	X	X	-	X	Max. version lag in violation
Δ-Regularity	X	X	-	X	Max. time lag in violation
k-Safeness	X	X	-	X	Max. version lag in violation
Δ-Safeness	X	X	-	X	Max. time lag in violation
t-Visibility	X	-	X	-	Prob. distr. of time lag
k-Staleness	X	-	X	-	Prob. distr. of version lag
Violations	-	X	X	-	No. of violations per time unit

Table 2. Overview of Data-centric Consistency Metrics

Client-centric Consistency Metrics Wada et al. [38] as well as Bermbach and Tai [8] propose to take a client-centric perspective for measuring consistency. This is of particular interest for application developers who can this way get concrete information to act upon. For client-centric consistency, there are again the two consistency dimensions ordering and staleness which both papers consider. Patil et al. [31] also propose to measure client-centric staleness in terms of time.

[7] This is a common problem for consistency metrics: Ordering cannot be considered properly without analysis of the request workload.

Staleness is best expressed either in terms of time (t-Visibility) or version lag (k-Staleness) in both cases the corresponding data-centric value is an upper bound for the client-centric one, as a system may employ additional mechanisms to hide staleness from the application. For example, in a quorum system with an (N,R,W) configuration of (5,2,2) data-centric t-Visibility will be determined by the time difference between the start of the update in replica 1 (or the commit timestamp – this depends on when updates become visible: right away or upon commit) and the end of the update in replica 5. The client-centric t-Visibility, in contrast, is determined again by the same start timestamp but ends when replica 4 completes the write as afterwards no request will ever again return the old value. Hence, data-centric staleness values are an upper bound for client-centric staleness values. Staleness can either be expressed as a density function (probability distribution of inconsistency window sizes) or as a cumulative density function (probability of reading fresh data Δt time units after the last update).

Ordering is best expressed in terms of the client-centric consistency models, i.e., the likelihood of a request violating a particular guarantee. Table 3 gives an overview of client-centric consistency metrics.

Metric	Staleness	Ordering	Continuous	Discrete	Unit & Description
MRC Violations	-	X	X	-	Prob. distr. of violation
MWC Violations	-	X	X	-	Prob. distr. of violation
RYWC Violations	-	X	X	-	Prob. distr. of violation
WFRC Violations	-	X	X	-	Prob. distr. of violation
t-Visibility	X	-	X	-	Prob. distr. of time lag
k-Staleness	X	-	X	-	Prob. distr. of version lag

Table 3. Overview of Client-centric Consistency Metrics

4.2 Consistency Benchmarking Approaches

After identifying the metrics most useful for measuring consistency in the last section, we will now describe benchmarking approaches for these metrics.

Data-centric Consistency All data-centric metrics require access to the actual replicas of the storage system. A test application creates load on the system. Results are then achieved by mining replica logs which should for each request contain the following information: start and end timestamp at each replica, some unique request id and the request type (read, write). Based on this, it is then possible to calculate t-Visibility, k-Staleness as well as the number of ordering violations and the corresponding consistency model[8].

[8] Some additional information like the (N,R,W) configuration for a quorum system may be necessary.

Client-centric Staleness and MRC Violations Both t-Visibility as well as k-Staleness can be benchmarked via the approach of [38] and its extension by [8]: Several geographically distributed machines interact with a storage system. A single writer periodically writes a timestamp and a version number to the storage system. The remaining machines continuously read the same data item from the storage system. Based on this the distribution of staleness (both based on time and version lag) can be calculated. The probability of MRC violations can be calculated by analyzing the results of each individual reader machine.

MWC Violations A single machine inserts a value into the storage system and directly updates it afterwards. After waiting for a sufficiently long period (all replicas need to synchronize) the key is read again and the result is compared to the updated value. If this is repeated for a large number of keys, the probability distribution for violations of MWC can be calculated.

RYWC Violations A single machine writes a value into the storage system and directly starts to continuously read it afterwards and logs the time difference to the end of the update as well as whether it was possible to read the new value or not. If this is repeated a statistically significant number of times, then it is possible to calculate the probability distributions for violations of RYWC as a function of the duration since the last update.

WFRC Violations So far, no benchmarking approach exists for WFRC violations. This can be explained by the fact that a violation cannot be directly observed by a client. One approach could be to use the replica logs of the storage system to identify if and how often WFRC has been violated.

Another approach could rely on the fact that WFRC violations mainly cause the effect that a delayed update message of an older version replaces the update that was executed on an older replica. If, for example, a client reads version $n+10$ and then issues an update which executes on a replica still at version n, then (depending on the storage system's implementation) either a delayed update message for version $n+10$ may replace the client's update (which leads to a lost update) or a conflicting version will be created which needs to be reconciled at a later point in time. If neither effect becomes visible, it still does *not* imply that WFRC is always guaranteed.

Finally, a third approach might work for storage systems which offer update operations beyond a CRUD interface. For example, a record append operation like in the Google File System [17] could be used followed by an analysis of the update order within the file.

5 Conclusion

In this work, we have provided an comprehensive overview of consistency in distributed systems. We started with a brief comparison of consistency in databases and distributed systems before focusing on the two perspectives on consistency in distributed systems. Next, we continued with a detailed discussion of data-centric and client-centric consistency models, their usecases and the relationships between those models before describing metrics and benchmarking approaches that help to determine consistency guarantees of distributed storage systems.

References

1. Abadi, D.: Consistency tradeoffs in modern distributed database system design: Cap is only part of the story. Computer 45(2), 37 –42 (feb 2012)
2. Anderson, E., Li, X., Shah, M., Tucek, J., Wylie, J.: What consistency does your key-value store actually provide. In: HotDep (2010)
3. Bailis, P., Fekete, A., Ghodsi, A., Hellerstein, J., Stoica, I.: The potential dangers of causal consistency and an explicit solution. In: Proceedings of the Third ACM Symposium on Cloud Computing. p. 22. ACM (2012)
4. Bailis, P., Venkataraman, S., Hellerstein, J., Stoica, I.: Probabilistically bounded staleness for practical partial quorums. VLDB Endowment (2012)
5. Bailis, P.: When is "acid" acid? rarely. http://www.bailis.org/blog/when-is-acid-acid-rarely (accessed Jan 28,2013) (2013)
6. Baker, J., Bond, C., Corbett, J., Furman, J., Khorlin, A., Larson, J., Léon, J., Li, Y., Lloyd, A., Yushprakh, V.: Megastore: providing scalable, highly available storage for interactive services. In: Proceedings of Conference on Innovative Data Systems Research
7. Bermbach, D., Kuhlenkamp, J., Derre, B., Klems, M., Tai, S.: A middleware guaranteeing client-centric consistency on top of eventually consistent datastores. In: IC2E. IEEE (2013)
8. Bermbach, D., Tai, S.: Eventual consistency: How soon is eventual? an evaluation of amazon s3's consistency behavior. In: Proceedings of the 6th Workshop on Middleware for Service Oriented Computing. p. 1. ACM (2011)
9. Brzezinski, J., Sobaniec, C., Wawrzyniak, D.: From session causality to causal consistency. In: PDP (2004)
10. Brzezinski, J., Sobaniec, C., Wawrzyniak, D.: Session guarantees to achieve pram consistency of replicated shared objects. PPAM (2004)
11. Chang, F., Dean, J., Ghemawat, S., Hsieh, W., Wallach, D., Burrows, M., Chandra, T., Fikes, A., Gruber, R.: Bigtable: A distributed storage system for structured data. ACM Transactions on Computer Systems (TOCS) 26(2), 1–26 (2008)
12. Chihoub, H., Ibrahim, S., Antoniu, G., Pérez, M., et al.: Consistency in the cloud: When money does matter! (2012)
13. Chihoub, H., Ibrahim, S., Antoniu, G., Pérez, M., et al.: Harmony: Towards automated self-adaptive consistency in cloud storage. In: IEEE CLUSTER (2012)
14. Chockler, G., Guerraoui, R., Keidar, I., Vukolic, M.: Reliable distributed storage. Computer 42(4), 60–67 (2009)
15. Codd, E.F.: The relational model for database management: Version 2. Addison-Wesley, Reading, Mass. (1990)
16. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: amazon's highly available key-value store. In: Proc. SOSP (2007)
17. Ghemawat, S., Gobioff, H., Leung, S.: The Google file system. ACM SIGOPS Operating Systems Review 37(5), 29–43 (2003)

18. Gilbert, S., Lynch, N.: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. ACM SIGACT News 33(2), 59 (2002)
19. Golab, W., Li, X., Shah, M.: Analyzing consistency properties for fun and profit. In: Proceedings of the 30th annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing. pp. 197–206. ACM (2011)
20. Guerraoui, R., Garbinato, B., Mazouni, K.: The garf library of dsm consistency models. In: Proceedings of the 6th workshop on ACM SIGOPS European workshop: Matching operating systems to application needs. pp. 51–56. ACM (1994)
21. Guerraoui, R., Hari, C.: On the consistency problem in mobile distributed computing. In: Proceedings of the second ACM international workshop on Principles of mobile computing. pp. 51–57. ACM (2002)
22. Helland, P., Campbell, D.: Building on quicksand. CIDR (2009)
23. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. 12(3), 463–492 (Jul 1990)
24. Kraska, T., Hentschel, M., Alonso, G., Kossmann, D.: Consistency rationing in the cloud: Pay only when it matters. Proceedings of the VLDB Endowment (2009)
25. Krishnamurthy, S., Sanders, W., Cukier, M.: An adaptive framework for tunable consistency and timeliness using replication. In: DSN. IEEE (2002)
26. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. ACM SIGOPS Operating Systems Review 44(2), 35–40 (2010)
27. Lamport, L.: Paxos made simple. ACM SIGACT News 32(4), 18–25 (2001)
28. Li, C., Porto, D., Clement, A., Gehrke, J., Preguiça, N., Rodrigues, R.: Making geo-replicated systems fast as possible, consistent when necessary. Tech. rep., Technical report, MPI-SWS. http://www. mpi-sws. org/chengli/rbTR. pdf (2012)
29. Lloyd, W., Freedman, M., Kaminsky, M., Andersen, D.: Don't settle for eventual: scalable causal consistency for wide-area storage with cops. In: SOSP. ACM (2011)
30. Mahajan, P., Alvisi, L., Dahlin, M.: Consistency, availability, and convergence. Technical Report TR-11-22, University of Texas at Austin (2011)
31. Patil, S., Polte, M., Ren, K., Tantisiriroj, W., Xiao, L., López, J., Gibson, G., Fuchs, A., Rinaldi, B.: Ycsb++: benchmarking and performance debugging advanced features in scalable table stores. In: SOCC. ACM (2011)
32. Rahman, M., Golab, W., AuYoung, A., Keeton, K., Wylie, J.: Toward a principled framework for benchmarking consistency. In: Proceedings of the 8th Workshop on Hot Topics in System Dependability (2012)
33. Ramakrishnan, R.: Cap and cloud data management. Computer (2012)
34. Tanenbaum, Andrew S. ; Steen, M.v.: Distributed systems : principles and paradigms. Pearson, Prentice Hall, Upper Saddle River, NJ, 2. ed. edn. (2007)
35. Torres-Rojas, F., Ahamad, M., Raynal, M.: Timed consistency for shared distributed objects. In: Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing. pp. 163–172. ACM (1999)
36. Torres-Rojas, F., Meneses, E.: Convergence through a weak consistency model: Timed causal consistency. CLEI ELECTRONIC JOURNAL 8(2) (2005)
37. Vogels, W.: Eventually consistent. Queue 6, 14–19 (October 2008)
38. Wada, H., Fekete, A., Zhao, L., Lee, K., Liu, A.: Data consistency properties and the trade offs in commercial cloud storages: the consumers' perspective. In: 5th biennial Conference on Innovative Data Systems Research, CIDR. vol. 11 (2011)
39. Yu, H., Vahdat, A.: Design and evaluation of a conit-based continuous consistency model for replicated services. ACM TOCS (2002)
40. Zellag, K., Kemme, B.: How consistent is your cloud application? In: Proceedings of the Third ACM Symposium on Cloud Computing. p. 6. ACM (2012)