Consistency Management in Cloud Storage Systems

Houssem-Eddine Chihoub[◇], Shadi Ibrahim[◇], Gabriel Antoniu[◇], María S. Pérez[‡]

[◇]INRIA Rennes - Bretagne Atlantique

Rennes, 35000, France

{*houssem-eddine.chihoub, shadi.ibrahim, gabriel.antoniu*}*@inria.fr*

[‡]Universidad Politécnica de Madrid

Madrid, Spain

mperez@fi.upm.es

March 1, 2013

Contents

Chapter 1

Consistency Management in Cloud Storage Systems

Abstract: *With the emergence of cloud computing, many organizations have moved their data to the cloud in order to provide scalable, reliable and high available services. As these services mainly rely on geographically-distributed data replication to guarantee good performance and high availability, consistency comes into question. The **CAP** theorem discusses tradeoffs between consistency, availability, and partition tolerance, and concludes that only two of these three properties can be guaranteed simultaneously in replicated storage systems. With data growing in size and systems growing in scale, new tradeoffs have been introduced and new models are emerging for maintaining data consistency. In this chapter, we discuss the consistency issue and describe the **CAP** theorem as well as its limitations and impacts on big data management in large scale systems. We then briefly introduce several models of consistency in cloud storage systems. Then, we study some state-of-the-art cloud storage systems from both enterprise and academia, and discuss their contribution to maintaining data consistency. To complete our chapter, we introduce the current trend toward adaptive consistency in big data systems and introduce our dynamic adaptive consistency solution (**Harmony**). We conclude by discussing the open issues and challenges raised regarding consistency in the cloud.*

1.1 Introduction

Cloud computing has recently emerged as a popular paradigm for harnessing a large number of commodity machines. In this paradigm, users acquire computational and storage resources based on a pricing scheme similar to the economic exchanges in the utility market place: users can lease the resources they need in a *Pay-As-You-Go* manner [1]. For example, the *Amazon Elastic Compute Cloud* (EC2) is using a pricing scheme based on virtual machine (VM) hours: *Amazon* currently charges $\$0.065$ per small instance hour at [2].

With data growing rapidly and applications becoming more data-intensive, many organizations have moved their data to the cloud, aiming to provide scalable, reliable and highly available services. Cloud providers allow service providers to deploy and customize their environment in multiple physically separate data centers to meet the ever-growing user needs. Services therefore can replicate their state across geographically diverse sites and direct users to the closest or least loaded site. Replication has become an essential feature in storage systems and is extensively leveraged in cloud environments [3][4][5]. It is the main reason behind several features such as fast accesses, enhanced performance, and high availability (as shown in Figure 1.1).

- For **fast access**, user requests can be directed to the closest data center in order to avoid communications' delays and thus insure fast response time and low latency.
- For **enhanced performance**, user requests can be re-directed to other replicas within the same data center (but different racks) in order to avoid overloading one single copy of the data and thus improve the performance under heavy load.
- For **high availability**, failure and network partitions are common in large-scale distributed systems; by replicating we can avoid single points of failure.

A particularly challenging issue that arises in the context of storage systems with geographically-distributed data replication is how to ensure a consistent state of all

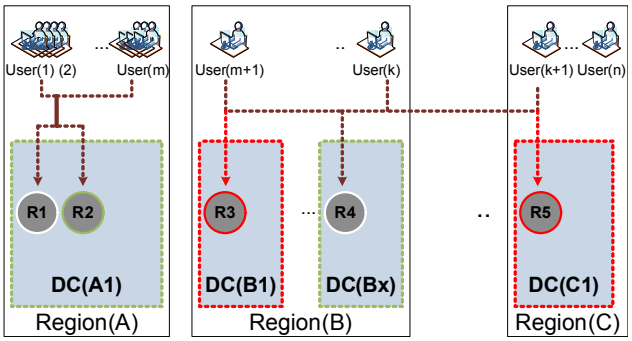

Figure 1.1: **Leveraging Geographically-distributed data Replication in the Cloud:** *Spreading replicas over different datacenters allows faster access by directing requests to close replicas (User(K+1) accesses R5 in DC(C1) instead of a remote replica); Under heavy load, replicas can serve users requests in parallel and therefore enhance performance (eg. R1 and R2 in Region (A)); When a replica is down, requests can failover to closer replicas (The requests of replicas within DC(B1) and DC(C1) failover to DC(Bx))*

the replicas. Insuring strong consistency by means of synchronous replication introduces an important performance overhead due to the high latencies of networks across data centers (the average round trip latency in *Amazon* sites varies from 0.3ms in the same site to 380ms in different sites [6]). Consequently, several weaker consistency models have been implemented (e.g., casual consistency, eventual consistency, timeline consistency). Such relaxed consistency models allow the system to return some stale data at some points in time.

Many cloud storage services opt for weaker consistency models in order to achieve better availability and performance. For example, *Facebook* uses the eventually consistent storage system *Cassandra* to scale up to host data for more than 800 million active users [7]. This comes at the cost of a high probability of stale data being read (i.e., the replicas involved in the reads may not always have the most recent write). As shown in [8], under heavy reads and writes some of these systems may return up to 66.61% stale reads, although this may be tolerable for users in the case of social network. With the ever-growing diversity in the access patterns of cloud applications along with the unpredictable diurnal/monthly changes in services loads and the variation in network latency (intra and inter-sites), static and traditional consistency solutions are not adequate for the cloud. With this in mind, several adaptive consistency solutions have been introduced to adaptively tune the consistency level at

run-time in order to improve the performance or to reduce the monetary cost while simultaneously maintaining a low fraction of stale reads.

In summary, it is useful to take a step back, consider the variety of consistency solutions offered by different cloud storage systems, and describe them in an unified way, putting the different use and types of consistency in perspective; this is the main purpose of this book chapter. The rest of this chapter is organized as follows: In Section 1.2 we briefly introduce the ***CAP*** theorem and its implications in cloud systems. Then we present the different types of consistency in Section 1.3. Then we briefly introduce the four main cloud storage systems used by big cloud vendors in Section 1.4. To complete our survey, we present different adaptive consistency approaches and detail our approach Harmony in Section 1.5. A conclusion is provided in Section 1.6.

1.2 The *CAP* Theorem and Beyond

1.2.1 The *CAP* theorem

In his keynote speech [9], *Brewer* introduced what is know since as *The* ***CAP*** Theorem. This theorem states that at most only two out of the three following properties can be achieved simultaneously within a distributed system: *Consistency*, *Availability* and *Partition Tolerance*. The theorem was later proved by *Gilbert and Lynch* [10]. The three properties are important for most distributed applications such as web applications. However, within the ***CAP*** theorem, one property needs to be forfeited, thus introducing several tradeoffs. In order to better understand these tradeoffs, we will first highlight the three properties and their importance in distributed systems.

Consistency: The consistency property guarantees that an operation or a transaction is performed atomically and leaves the systems in a consistent state, or fails instead. This is equivalent to the atomicity and consistency properties (***AC***) of the ***ACID*** (Atomicity, Consistency, Isolation and Durability) semantics in relational database management systems (***RDBMs***), where a common way to guarantee (strong) consis-

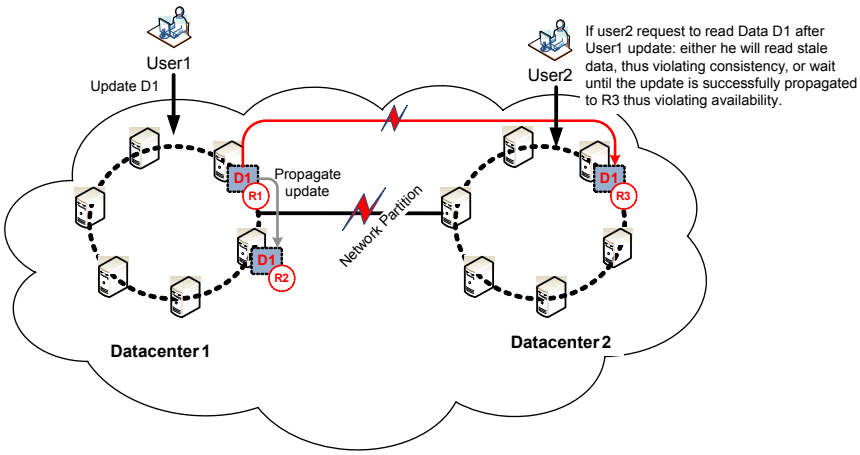

Figure 1.2: **Consistency *vs* Availability in Geo-replicated Systems**

tency is linearizability.

Availability: In their ***CAP*** theorem proof [10], the authors define a distributed storage system as continuously available if every request received by a non-failing node must result in a response. On the other hand, when introducing the original ***CAP*** theorem, *Brewer* qualified a system to be available if *almost* all requests receive a response. However, in these definitions, no time bounds on when the requests would receive a response were specified, which left the definition somewhat vague.

Partition Tolerance: In a system that is partition tolerant, the network is allowed to loose messages between nodes from different components (datacenters for instance). When a network partition appears, the network communication between two components (partitions, datacenters etc.) is off and all the messages are lost. Since replicas may be spread over different partitions in such a case, this property has a direct impact on both consistency and availability.

The implications of the ***CAP*** theorem introduced challenging and fundamental tradeoffs for distributed systems and services designers. Systems that are designed to be deployed on single entities such as an RDBM aim to provide both availability and consistency properties with partitions were not an issue. However for distributed systems that rely on networking, such as geo-replicated systems, partition tolerance is a must for a big majority of them. This in turn introduces, among other tradeoffs

derived from the ***CAP*** theorem, the *Consistency vs. Availability* as a major tradeoff in geo-replicated systems. As shown in Figure 1.2, user requests can be served from different replicas in the system. If partitions occur, an update on one replica can not be propagated to other replicas on different partitions. Therefore, those replicas could be made either *available* to the clients, thus violating consistency, or otherwise, made *unavailable* until they converge to a *consistent* state, which can happen after recovering from the network partition.

1.2.2 Beyond the *CAP* Theorem

The proposal of the ***CAP*** theorem a few years ago had a huge impact on the design of distributed systems and services. Moreover, the ever-growing volume of data along with the huge expansion of distributed systems scales makes the implications of the ***CAP*** theorem of even more importance. Twelve years after the introduction of his ***CAP*** theorem, *Brewer* still ponders its implications [11]. He estimates that the theorem achieved its purpose in the past in the way it brought the community's attention to the related design challenges. On the other hand, he judges some interpretations of the implications as misleading, in particular, the 2 out of 3 tradeoff property. The general belief is that the partition tolerance property P is insurmountable for wide-area systems. This often leads designers to completely forfeit consistency C or availability A for each other. However, partitions are rare. *Brewer* states that the modern goal of the ***CAP*** theorem should be to maximize combinations of C and A. In addition, system designers should develop mechanisms that detect the start of partitions, enter an explicit partition mode with potential limitations of some operations, and finally initiate partition recovery when communication is restored.

Abadi [12] states as well that the ***CAP*** theorem was misunderstood. ***CAP*** tradeoffs should be considered under network failures. In particular, the Consistency-Availability tradeoff in ***CAP*** is for when partitions appear. The theorem property P implies that a system is partition-tolerant and more importantly, is enduring partitions. Therefore, and since partitions are rare, designers should consider other tradeoffs that are arguably, more important. A tradeoff that is more influential, is the

latency-consistency tradeoff. Insuring strong consistency in distributed systems requires a synchronized replication process where replicas belong to remote nodes that communicate through a network connection. Subsequently, reads and updates may be costly in terms of latency. This tradeoff is ***CAP**-Independent* and exists permanently. Moreover, *Abadi* makes a connection between latency and availability. When latency is higher than some timeout the system becomes unavailable. Similarly, the system is available if the latency is smaller than the timeout. However, the system can be available and exhibit high latency nonetheless. For these reasons, system designers should consider this additional tradeoff along with ***CAP***. *Abadi* propose to unify the two in a unified formulation called *PACELC* where PAC are ***CAP*** tradeoffs during partitions and ELC is the latency consistency tradeoff.

After they proved the ***CAP*** theorem, *Gilbert et Lynch* reexamined the theorem properties and its implications [13]. The tradeoff within ***CAP*** is another example of the more general tradeoff between *safety* and *liveness* in unreliable systems. Consistency can be seen as a *safety* property for which every response to client requests is correct. In contrast, availability is a *liveness* property that implies that every client request would eventually receive a response. Hence, viewing ***CAP*** in the broader context of safety-liveness tradeoffs provide insight into the feasible design space for distributed systems [13]. Therefore, they reformulate the ***CAP*** theorem as follows: "***CAP*** *states that any protocol implementing an atomic read/write register cannot guarantee both safety and liveness in a system prone to partitions*". As a result, the practical implications dictate that designers opt for best-effort availability, thus guaranteeing consistency, and best-effort consistency for systems that must guarantee availability. A pragmatic way to handle the tradeoff is by balancing consistency-availability tradeoff in an adaptive manner. We will further explore this idea in Section 1.5.

1.3 Consistency Models

In this section we introduce the main consistency models adopted in earlier single-site storage systems and in current geo-replicated systems and then we summarize them in Table 1.1.

1.3.1 Strong consistency

In traditional distributed storage and database systems, the instinctive and correct way to handle replicas consistency was to insure a strong consistency state of all replicas in the system all the time. For example, the RDBMs were based on ACID semantics. These semantics are well defined and insure a strong consistency behavior of the RDBM based on the atomicity and consistency properties. Similarly, the POSIX standard for file systems imply that data replicated in the system should always be consistent. Strong consistency guarantees that all replicas are in a consistent state immediately after an update, before it returns a success. Moreover, all replicas perceive the same order of data accesses performed by different client processes.

In a perfect world, such semantics and the strong consistency model are the properties that every storage system should adopt. However, insuring strong consistency requires mechanisms that are very costly in terms of performance and availability and limit systems scalability. Intuitively, this can be understood as a consequence of the necessity to exchange messages with *all* replicas in order to keep them synchronized. This was not an issue in the early years of distributed storage systems as the scale and the performance needed at the time were not as important. However, in the era of big data and cloud computing, this consistency model can be penalizing, in particular if such a strong consistency is actually not required by the applications!.

Several mechanisms and correctness conditions to insure strong data consistency were proposed over the years. Two of the most popular approaches are *serializability* [14] and *linearizability* [15].

Serializability: A set of concurrent actions execution on a set of objects is serializable if it is equivalent to a serial execution. Every action is considered as a serialization unit and consists of one or more operations. Each operation may be performed concurrently with other operations from different serialization units. Serialization units are equivalent to transactions in RDBMs and a single file system call in the case of file systems.

A concurrent execution on a set of replicated objects is *one-copy equivalent* if it is equal to an execution on the same set of objects without replication. As a result, concurrent actions execution is said to be one-copy serializable if it is serializable and one-copy equivalent. Moreover, a one-copy serializable execution is considered global one-copy serializable if the partial orderings of serialization units, which perceived by each process, are preserved.

Linearizability: The linearizability or the atomicity of a set of operations on a shared data object is considered as a correctness condition for concurrent shared data objects [15]. Linearizibility is achieved if every operation performed by a concurrent process appears instantaneously to the other concurrent processes at some moment between its invocation and response. Linearizability can be viewed as a special case of global one-copy serializability where a serialization unit (a transaction) is restricted to consist of a single operation [15]. Subsequently, linearizability provides locality and non-blocking properties.

1.3.2 Weak Consistency

The implementation of strong consistency models imposes serious limitations to designers and clients requirements. Moreover, insuring strong global total ordering has a heavy performance overhead. As to overcome these limitations, *Dubois et al.* [16] first introduced the weak ordering model. Data accesses (read and write operations) are considered as weakly ordered if they satisfy the following three conditions:

- All accesses to a shared synchronization variables are strongly (sequential) ordered. All processes perceive the same order of operations.
- Data accesses to a synchronization variable are not issued by processors before all previous global accesses have been globally performed.
- A global data access is not allowed by processors until a previous access to synchronization variable is globally performed.

From these three conditions, the order of read and write operations, outside critical sections (synchronization variables), can be seen in different orders by different processes as long as they don't violate the aforementioned conditions. However, in [17] [18], it has been argued that not all the three conditions are necessary to reach the intuitive goals of weak ordering. Numerous variation models have been proposed since. *Bisiani et al.* [19] proposed an implementation of weak consistency on distributed memory system. Timestamps are used to achieve a weak ordering of the operations. A synchronization operation is completed only after all previous operations in the systems reach a completion state. Various weaker consistency models derived from the weak ordering. The following client-side models are weak consistency models, but provide some guarantees to the client.

Read-your-writes : This model guarantees that a process that commits an update will always be able to see the updated value with the read operation but not an older one. This might be an important consistency property to provide with weakly ordered systems for a large class of applications. As will be seen further in this section, this is a special case of causal consistency.

Session consistency : Read-your-writes consistency is guaranteed in the context of a *session* (which is a a sequence of accesses to data, usually with an explicit beginning and ending). As long as the users access data -during the same session, they are guaranteed to access their latest updates. However, the read-your-writes property is not guaranteed to be spanned over different sessions.

Monotonic reads : A process should never read a data item value older than what it has read before. This consistency guarantees that a process's successive reads returns always the same value or a more recent one than the previous read.

Monotonic writes : This property guarantees the serialization of the writes by one process. A write operation on a data object or item must be completed before any successive writes by the same process. Systems that do not guarantee this property are notoriously hard to program [20].

1.3.3 Causal Consistency

Causal consistency is a consistency model where a sequential ordering is always preserved only between operations that have causal relation. Operations that execute concurrently do not share a causality relation. Therefore, causal consistency does not order concurrent operations. In [21][22], two operations a and b have a potential causality if one of the two following conditions are met: a and b are executed in a single thread and one operation execution precedes the other in time; or if b reads a value that is written by a. Moreover, a causality relation is transitive. If a and b have a causal relation, b and c have a causal relation as well, then a and c have a causal relation.

In [22], a model that combines causal consistency and convergent conflict handling is presented and called causal+. Since concurrent operations are not ordered by causal consistency, two writes to the same key or data object would lead to a conflict. Convergent conflict handling aims at handling all the replicas in the same manner using a handler function. To reach handling convergence, all conflicting replicas should consent to an agreement. Various conflict handling methods were proposed such as last-writer-wins rule, through user intervention, or versioning mechanism as in Amazon's Dynamo storage system.

1.3.4 Eventual Consistency

In a replicated storage system, the consistency level defines the behavior of divergence of replicas of logical objects in the presence of updates [23]. Eventual consistency [24] [23] [20], is the weakest consistency level that guarantees convergence. In the absence of updates, data in all replicas will gradually and *eventually* become consistent.

Eventual consistency ensures the convergence of all replicas in systems that implement lazy, update-anywhere or optimistic replication strategies [25]. For such sys-

tems, updates can be performed on any replica hosted on different nodes. The update propagation is done in a lazy fashion. Moreover, this update propagation process may encounter even more delays considering cases where network latency is of a high order such as for geo-replication. Eventual consistency is ensured through mechanisms that will guarantee the propagation process which will successfully end at a future (maybe unknown) time. Furthermore, Vogels [20] judges that, if no failures occur, the size of the inconsistency window can be determined based on factors such as communication delays, the load on the system, and the number of replicas in the system.

Eventual consistency by the mean of lazy asynchronous replication may allow better performance and faster accesses to data. Every client can read data from local replicas located in a geographically close data center. However, if an update is performed on one of the replicas and is yet to be propagated to others because of the asynchronous replication mechanism, a client reading from a distant replica may read stale data.

In [24], two examples that illustrate the typical use case and show the necessity for this consistency model were presented. The worldwide domain name system (DNS) is a perfect example of system for which eventual consistency is the best fit. The DNS namespace is partitioned into domains where each domain is assigned to a naming authority. This is an entity that will be responsible for this domain and is the only one that can update it. This scheme eliminates the update-update conflict. Therefore, only the read-update conflict needs to be handled. As updates are less frequent, in order to maintain system availability and fast accesses for users read operations, lazy replication is the best fit solution. Another example is the World Wide Web. In general, each web page is updated by a single authority, the webmaster. This also avoids any update-update conflict. However, in order to improve performance and lower read access latency, browsers and Web proxies are often configured to keep a fetched page in a local cache for future requests. As a result, a stale out-of-date page maybe read. However, many users find this inconsistency acceptable (to a certain degree) [24].

Table 1.1: Consistency Models

Consistency Model		**Brief Description**
Strong Consistency	serializability	Serial order of concurrent executions of a set of serialization units (set of operations).
	linearizability	Global total order of operations (single operations), every operation is perceived instantaneously.
Weak Consistency	Read-your-writes	A process always see his last update with read operations.
	Session consistency	Read-your-writes consistency is guaranteed only within a session.
	Monotonic reads	Successive reads must always return the same or more recent value than a previous read.
	Monotonic writes	A write operation must always complete before any successive writes.
Causal Consistency		Total ordering between operations that have causal relation.
Eventual Consistency		In the absence of updates, all replicas will gradually and eventually become consistent.
Timeline Consistency		All replicas perform operations on one record in the same "correct order".

1.3.5 Timeline Consistency

The timeline consistency model was proposed specifically for the design of Yahoo! PNUTS [26], the storage system designed for Yahoo web applications. This consistency model was proposed to overcome the inefficiency of serializability of transactions at massive scales and geo-replication. Moreover, it aims to limit the weakness of eventual consistency. The authors abandoned transaction serializability as a design choice after they observed that web applications typically manipulate one record at a time. Therefore, they proposed a per-record timeline consistency. Unlike eventual consistency, where operations order can vary from one replica to another, All replicas of a record perform the operations in the same "correct" order. For instance, if two concurrent updates are performed, all replicas will execute them in the same order and thereby avoid inconsistencies. Nevertheless, data propagation to replicas is done lazily, which makes the consistency of all replicas eventual. This may allow clients that read data from local replicas to access a stale version of data. In order to preserve the order of operations for a given record, one replica is designated dynamically as a master replica for the record that handles all the updates.

1.4 Cloud Storage Systems

In this section we describe some state-of-art cloud storage systems which are adopted by the big cloud vendors, such as Amazon Dynamo, Apache Cassandra, Yahoo! PNUTS and Google Spanner. We then give an overview of their real-life applications and use-cases (summarized in Table 1.2).

1.4.1 Amazon Dynamo

Amazon Dynamo [27], is a storage system designed by Amazon engineers to fit the requirements of their web services. Dynamo provides the storage backend for the highly available worldwide Amazon.com e-commerce platform and overcomes the inefficiency of RDBMs for this type of applications. Reliability and scaling requirements within this platform services are high. Moreover, availability is very important, as the increase of latencies by only minimal fractions can cause financial losses. Dynamo provides a flexible design where services may control availability, consistency, cost-effectiveness and performance tradeoffs.

Since Amazon has been tailored specifically for Amazon services, the SLA (Service Level Agreement) of those services were carefully considered. A simple SLA constraint is for instance to provide a response within 300ms for 99.9% of client requests under a peak load of 500 requests per second. Dynamo has been designed in a completely decentralized, massively scalable way, with flexible usage in terms of availability, consistency and performance (latency and throughput). Aggregated services within Amazon can access data in a manner that does not violate SLA constraints, with 99.9 percentile guaranteed quality of service, whereas other services settle for median (average) percentile quality.

Dynamo's partitioning scheme relies on a variation of consistent hashing [28]. In their scheme, the resulting range or space of a hash function is considered as a ring. Every member of the ring is a virtual node (host) where a physical node may be

responsible for one or more virtual nodes. The introduction of virtual nodes, instead of using fixed physical nodes on the ring, is a choice that provides better availability and load balancing under failures. Each data item can be assigned to a node on the ring based on its key. The hashed value of the key determines its position on the ring. Data then, is assigned to the closest node on the ring clockwise. Moreover, data is replicated on the successive $K-1$ nodes for a given replication factor K, avoiding virtual nodes that belong to the same physical nodes. All the nodes on Dynamo are considered equal and are able to compute the *reference list* for any given key. The reference list is the list of nodes that store a copy of data referenced by the key.

Dynamo is an eventually-consistent system. Updates are asynchronously propagated to replicas. As data is usually available while updates are being propagated, clients may perform updates on older versions of data for which the last updates had not been committed yet. As a result, the system may suffer from update conflicts. To deal with these situations, Dynamo relies on data versioning. Every updated replica is assigned a new immutable version. The conflicting versions of data resulting from concurrent updates may be solved at a latter time. This allows the system to be always available and fast to respond to client requests. Versions that share a causal relation are easy to solve by the system based on syntactic reconciliation. However, a difficulty arises with versions branching. This often happens in the presence of failures combined with concurrent updates and results in conflicting versions of data. The reconciliation in this case is left to the client rather than the system because the latter lacks the semantic context. The reconciliation is performed by collapsing the multiple data versions into one (semantic reconciliation). A simple example is the case of the *shopping cart* application. This application chooses to merge the diverging versions as a reconciliation strategy. Dynamo uses vector clocks to implement this mechanism. Vector clocks allow the system to capture causality between different versions if any or detect version conflicts otherwise. A vector clock for a given version of a given key consists of (node, counter) pairs. Moreover, timestamps are used to indicate the last time the node updated the data item. In order to detect inconsistencies between replicas and repair them in the event of failures and other threats to data durability, Dynamo implements an anti-entropy replicas synchronization protocol.

Replica consistency is handled by a quorum-like system. In a system that maintains N replicas, R is the minimum number of nodes (replicas) that must participate in the read operation, and W is the minimum number of nodes that must participate in a write operation are configured on a per operation basis and are of high importance. By setting these two parameters, one can define the tradeoff between consistency and latency. A configuration that provides R + $W > N$ is a quorum-like setting. This configuration insures that the last up-to-date replica is included in the quorum and thus the response. However, the operation latencies are as small as the longest replica response time. In a configuration where R + $W < N$, clients may be exposed to stale versions of data.

1.4.2 Cassandra

The Facebook social networking platform is the largest networking platform serving hundred millions of users at peak times and having no less than 900 million active users [7]. Therefore, and in order to keep users satisfied, an efficient Big Data management that guarantees high availability, performance, and reliability is required. Moreover, a storage system that fulfills these needs must be able to elastically scale out to meet the continuous growth of the data-intensive platform. Cassandra [29] is a highly available, highly scalable distributed storage system that was first built within Facebook. It was designed for managing large objects of structured data spread over a large amount of commodity hardware located in different datacenters worldwide.

The design of Cassandra was highly inspired by that of two other distributed storage systems. Implementation choices and consistency management are very similar to the ones of Amazon Dynamo (except for in-memory management) while its data model is derived from the Google BigTable [4] model. The Cassandra partitioning scheme is based on consistent hashing. Unlike Dynamo, which uses virtual nodes to overcome the non-uniformity of load distribution, every node on the ring is a physical host. Therefore, and in order to guarantee uniform load distribution, Cassandra uses the same technique as in [30], where lightly loaded nodes move on the ring.

Cassandra stores data in tables. A table is a distributed multidimensional map where every row is indexed by a key. Every operation on a single row key is atomic per a replica without considering which columns are accessed. In this model, as in Google BigTable, columns are dynamic and grouped into column families. Such a data model provides great abilities for structured large data, as it offers a more flexible yet efficient data access. Moreover, it enables a better dynamic memory management. Like BigTable, Cassandra keeps data in-memory in small tables called *memtables*. When a memtable size grows over a given threshold, it is considered as full and data is flushed into an *sstable* that will be dumped to the disk.

Replication in Cassandra is performed in the same manner as in Dynamo. However, Cassandra implements few replication strategies that consider the system topology. Therefore, strategies that are *Rack UnAware*, *Rack Aware*, and *Datacenter Aware* are provided. For the two latter strategies, Cassandra implements algorithms in Zookeeper [31] in order to compute *the reference list* for a given key. This list is cached locally at the level of every node as to preserve the zero-hop property of the system.

In the Cassandra storage system, several consistency levels [32] are proposed per operation. A write of consistency level *One* implies that data has to be written to the commit log and memory table of at least one replica before returning a success. Moreover, as shown in Figure 1.3, a read operation with consistency level *All* (strong consistency) implies that the read operation must wait for all the replicas to reply and insures that all replicas are consistent in order to return the data to the client. In contrast, in a read consistency of level of *Quorum*, 2 of the 3 replicas are contacted to fulfill the read request and the replica with the most recent version would return the requested data. In the background, a read repair will be issued to the third replica and will check for consistency with the first two. If inconsistency occurs, an asynchronous process will be launched to repair the stale nodes at a latter time.

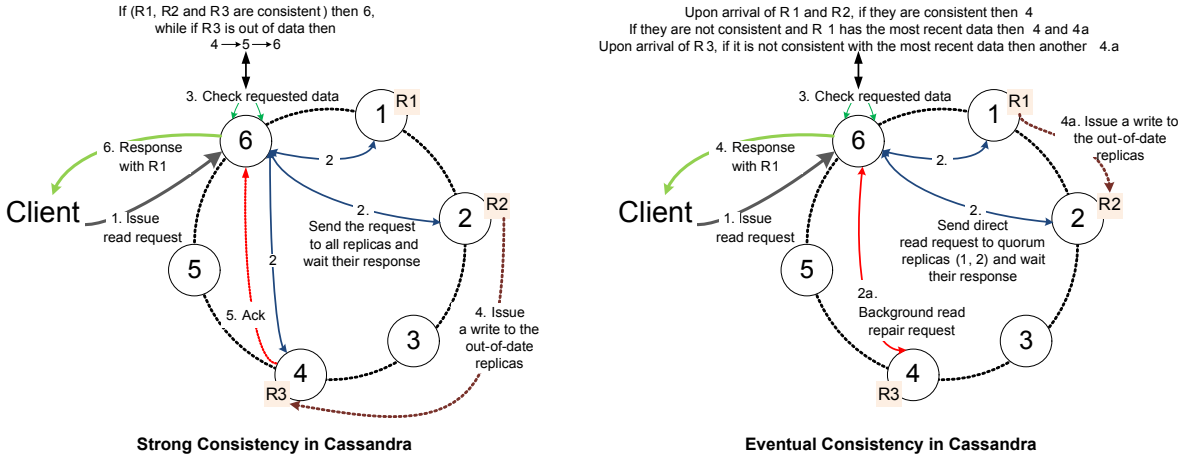

Figure 1.3: **Synchronous replication *vs* Quorum replication in Cassandra [33]**

1.4.3 Yahoo! PNUTS

Yahoo!'s requirements for a data management platform that provides scalability, fast response, reliability, and high availability in different geographical areas, led them to the design and implementation of *PNUTS* [26]. PNUTS is a massively parallel geographically distributed storage system. Its main purpose is to host and serve data for Yahoo! web applications. PNUTS relies on a novel relaxed consistency model to cope with availability and fault-tolerance requirements at large scale.

PNUTS provides the user with a simplified relational model. Data is stored in a set of tables of records with multiple attributes. An additional data type provided to the users is the "blob" type. A blob encapsulates arbitrary data structures (not necessarily large objects) inside records. PNUTS divides the systems into a set of regions. Regions are typically, but not necessarily, geographically distributed. Every region consists of a set of *storage units*, a *tablet controller* and a set of *routers*. Data tables are decomposed horizontally into smaller data structures called *tablets* that are stored across storage units (servers) within multiple regions. On the other hand, the routers functionality is to locate data within tablets and storage units based on a *mapping* computed and provided by the tablet controller. PNUTS introduces the novel consistency model of *per-record timeline consistency* described in section 1.3. Therefore, it uses an asynchronous replication scheme. In order to provide reliability

and replication, PNUTS relies on a pub/sub mechanism, called *Yahoo! Message Broker* (YMB). With YMB, PNUTS avoids other asynchronous replication protocols such as Gossip, and optimizes geographical replication. Moreover, a replica does not need to acquire the location of other replicas. Instead, it needs just to subscribe to the data updates within YMB.

In order for applications and users to deal with timeline consistency, API calls which provide varying consistency guarantees were proposed. The *read-any* call may return stale data to the users favoring performance and fast response to consistency. In common cases, a class of applications require the read data to be more recent than a given version. The API call *read-critical (required_version)* is proposed to deal with these requirements. In contrast, the *read-latest* call always the most recent version of data. This call however may be costly in terms of latency. Moreover, the API provides two calls for writing data. The *write* call gives ACID guarantees for the write (a write is a transaction with a single operation). In contrast, *test-and-set-write(required_version)* checks the version of the actual data in the system. If, and only if, the version matches *required_version*, the write is performed. This flexible API calls give a degree of freedom to applications and users to choose their consistency guarantees and control their availability, consistency, and performance tradeoffs.

1.4.4 Google Spanner

Spanner [42] is a scalable, globally distributed database that provides synchronous replication and ensures strong consistency. While many applications within Google require geo-replication for global availability and geographical locality reasons, a large class of these applications still needs a strong consistency and an SQL-like query model. Google BigTable[4] still serves and manages data efficiently for many applications, but it only guarantees eventual consistency at global scale and provides a noSQL API. Therefore, Spanner is designed to overcome BigTable insufficiencies for a class of applications and provides globe scale external consistency (linearizability) and SQL-like query language similar to Google Megastore query language [38].

Table 1.2: Cloud Storage Systems

Storage System	Consistency Model	License	Cloud Applications/Services
Amazon Dynamo	Eventual Consistency	Internal	Amazon.com e-commerce platform, Few AWS (Amazon Web Services) (eg. DynamoDB)
Cassandra	Eventual Consistency	Open Source	Facebook inbox search, Twitter, Netflix, eBay, SOUNDCLOUD, RackSpace Cloud
Riak [34]	Eventual Consistency	Open Source	Yammer private social network, Clipboard, GitHub, enStratus Cloud
Voldemort [35]	Eventual Consistency	Open Source	LinkedIn, eHarmony, GiltGroup, Nokia
CouchDB [36]	Eventual Consistency	Open Source	Ubuntu One cloud, BBC (Dynamic Content Platform), Credit Suisse (Market Place Framework)
MongoDB [37]	Eventual Consistency	Open Source	SAP AG Software Entreprise, MTV, and Sourceforge
Yahoo PNUTS!	Timeline Consistency	Internal	Yahoo web applicastions
Google BigTable [4]	Strong Consistency	Internal	Google analytics, Google earth, Google personalized search
Google Megastore [38]	Strong Consistency	Internal	Google applications: Gmail, Picasa, Google Calendar, Android Market, and AppEngine
Google Spanner	Strong Consistency	Internal	Google F1
Redis [39]	Strong Consistency	Open Source	Instagram, Flickr, The guardian news paper
Microsoft Azure Storage [40]	Strong Consistency	Internal	Microsoft internal applications: networking search, serving video, music and game content, Blob storage cloud service
Apache HBase [41]	Strong Consistency	Open Source	Facebook messaging system, traditionally used with Hadoop for large set of applications

The Spanner architecture consists of a *universe* that may contain several *zones* where zones are the unit of administrative deployment. A zone additionally presents a location where data may be replicated. Each zone encapsulates a set of *spanservers* that host data tables split into data structures called *tablet*. Spanner timestamps data in order to provide multi-versioning features. A *zonemaster* is responsible for assigning data to spanservers whereas, the *location proxies* components provide clients with information to locate the spanserver responsible for its data. Moreover, Spanner introduces an additional data abstraction called *directories*, which are a kind of buckets to gather data that have the same access properties. The directory abstraction is the unit used to perform and optimize data movement and location. Data is stored into semi-relational tables to support an SQL-like query language and general-purpose transactions.

Replication is supported by implementing a Paxos protocol. Each spanserver associates a Paxos state machine with a tablet. The set of replicas for a given tablet is called a *Paxos group*. For each tablet and its replicas, a long-lived Paxos leader is designated with a time-based leader lease. The Paxos state machines are used to keep a consistent state of replicas. Therefore, writes must all initiate the Paxos protocol at

the level of the Paxos leader while reads can access Paxos states at any replica that is sufficiently up-to-date. At the level of the leader replica, a lock table is used to manage concurrency control based on a two-phase locking (2PL) protocol. Consequently, all operations that require synchronization should acquire locks at the lock table. In order to manage the global ordering and an external consistency, Spanner relies on a time API called *TrueTime*. This API exposes clock uncertainty and allows Spanner to assign globally meaningful commit timestamps. The clock uncertainty is kept small within the TrueTime API relying on atomic clocks and GPS based clocks at the level of every data center. Moreover, when uncertainty grows to a large value, Spanner slows down to wait out that uncertainty. The TrueTime API is then used to guarantee spanner desired correctness properties for concurrent executions. Therefore, providing external consistency (linearizability) while enabling lock-free for read-only transactions and non-blocking reads in the past.

Spanner presents a novel globally-distributed architecture that implements the first globally ordered system with external consistency guarantees. While such guarantees were estimated to be fundamental for many applications within Google, it is unclear how such an implementation affects latency, performance, and availability. In particular, the write throughput might suffer from the two-phase locking mechanism, which is known to be very expensive at wide scale. Moreover, it is not obvious how Spanner deals with availability during network failures.

1.4.5 Discussion

As cloud computing technology emerges, more and more cloud storage systems have been developed. Table 1.2 gives an overview of the four aforementioned cloud storage systems along with several other storage system examples. They all serve as back-end storage system for several services. These services cover many fields in addition to social applications and business platforms. We can observe that for all these systems the designers had to opt for a unique consistency model, typically either strong consistency or eventual consistency.

1.5 Adaptive Consistency

To cope with the cloud dynamicity and scale as well as the ever-growing users' requirements, many adaptive and dynamic consistency approaches were introduced. Their goal is to use strong consistency only when it is necessary. Hereafter, we discuss three adaptive approaches differ in the way they define the consistency requirements as summarized in Table 1.3.

1.5.1 RedBlue Consistency

RedBlue consistency [6] is introduced in order to provide as fast responses as possible and consistency when necessary. It provides two types of operations: Red and Blue. Blue operations are executed locally and replicated lazily. Therefore, their ordering can vary from site to site. In contrast, Red operations require a stronger consistency. They must satisfy serializable ordering with each other and as a result generate communication across sites for coordination. Subsequently, the RedBlue order is defined as a partial ordering for which all Red operations are totally ordered. Moreover, every site has a local causal serialization that provides a total ordering of operations which are applied locally. This definition of the RedBlue consistency does not guarantee the replicas state convergence. Convergence is reached if all causal serializations of operations at the level of each site reach the same state. However, with the RedBlue consistency, blue operations might have different orders in different sites. Therefore, non-commutative operations executed in a different order won't allow the replicas convergence. As a result, non-commutative operations should not be tagged as blue if the convergence is to be insured. An extension of the RedBlue consistency consists in splitting original application operations into two components. A *generator operation* that has no side-effect and is executed only at the primary site and *shadow operation*, which is executed at every site. *Shadow operations* that are non-commutative or violate the application variant (e.g. negative values for a positive variable) are labeled Red while all other *shadow operations* are labeled blue.

The RedBlue consistency is implemented in a system called *Gemini* storage system. Gemini uses MySQL as its storage backend. Its deployment consists of several sites where each site is composed of four components: a *storage engine*, a *proxy server*, *concurrency coordinator*, and *data writer*. The proxy server is the component that processes client requests for data hosted on the storage engine (a relational database). *Generator operation*s are performed on a temporary private scratchpad, resulting in a virtual private copy of the service state. Upon the completion of a *generator operation*, the proxy server sends the *shadow operation* to the concurrency coordinator. The latter notifies the proxy server whether the operation is accepted or rejected according to the RedBlue consistency. If accepted, the operation is then delegated to the local data writer in order to be executed in the storage engine.

1.5.2 Consistency Rationing

The *consistency rationing* model [43] allows designers to define consistency requirements on data instead of transactions. It divides data into three categories: A, B, and C. Category A data requires strong consistency guarantees. Therefore, all transactions on this data are serializable. However, serializability requires protocols and implementation techniques as well as coordination, which are expensive in terms of monetary cost and performance. Data within C category is data for which temporary inconsistency is acceptable. Subsequently, only weaker consistency guarantees, in the form of session consistency, are implemented for this category. This comes at a cheaper cost per transaction and allows better availability. The B category on the other hand presents data for which consistency requirements change in time as in the case for many applications. These data endure adaptive consistency that switch between serializability and session consistency at runtime whenever necessary. The goal of the adaptive consistency strategies is to minimize the overall cost of the provided service in the cloud. The general policy is an adaptive consistency model that relies on an updates conflict probability. It observes the data access frequency to data items in order to compute the probability of access conflicts. When this probability grows over an adaptive threshold, serializability is selected. The computation of the

adaptive threshold is based on the monetary cost of weak and strong consistency, and the expected cost of violating consistency.

Consistency rationing is implemented in a system that provides storage on top of *Amazon Simple Storage Service* (S3) [44], which provides only eventual consistency. Clients Requests are directed to application servers. These servers are hosts on *Amazon EC2* [2]. Therefore, application servers interact with the persistent storage on *Amazon S3*. In order to provide consistency guarantees, the update requests are buffered in queues called pending updates queues that are implemented on the *Amazon Simple Queue Service* (SQS) [45]. Session consistency is provided by always routing requests from the same client to the same server within a session. In contrast, and in order to provide serializability, a two-phase locking protocol is used.

1.5.3 Harmony: Automated Self-Adaptive Consistency

While most of the existing adaptive consistency approaches require a global knowledge of the application access pattern (e.g., consistency rationing requires the data to be categorized in advance). However, with the tremendous increase in data size along with the significant variation in services load, this task is hard to accomplish and will add an extra overhead to the system. Moreover, these approaches cover a small set of applications where operation orderings is strictly required. In this context, we introduce our approach Harmony [33], an *automated self-adaptive* approach that considers applications tolerance rate for stale reads (i.e., Harmony complements other adaptive approaches as it targets applications with stale reads consideration rather than operation orderings). Harmony tunes the consistency level at run-time according to the application requirements. To cope with the ever-growing diversity in the access patterns of cloud applications, Harmony embraces an intelligent estimation model of stale reads, allowing to elastically scale up or down the number of replicas involved in read operations to maintain a low tolerable fraction of stale reads.

Why use the stale reads rate to define the consistency requirements of an

Table 1.3: Adaptive consistency approaches

	The level at which the consistency is specified	**Cloud Storage system: implemented within**	**Testbed for evaluating the solution**
RedBlue Consistency	Operations	Gemini	Amazon EC2 in different availability zones
Consistency Rationing	Data	Amazon S3	Amazon S3
Harmony	Operations	Apache Cassandra	Grid'5000 and Amazon EC2

application? We consider two applications that may at some point have the same access pattern. One is a web-shop application that can have heavy reads and writes during the busy holiday periods, and a social network application that can also have heavy access during important events or in the evening of a working day. These two applications may have the same behavior at some point and are the same from the point of view of the system when monitoring data accesses and network state, thus they may be given the same consistency level. However, the cost for stale reads is not the same for both applications. A social network application can tolerate a higher number of stale reads than a web-shop application: a stale read has no effects on the former, whereas it could result in anomalies for the latter. Consequently, defining the consistency level in accordance to the stale reads rate can precisely reflect the application requirement.

Harmony Implementation Harmony can be applied to different cloud storage systems that are featured with flexible consistency rules. The current implementation of Harmony operates on top of *Apache Cassandra* storage [46] and consists of two modules. The monitoring module collects relevant metrics about data access in the storage system: read rates and write rates, as well as network latencies. These data are further fed to the adaptive consistency module. This module is the heart of the Harmony implementation where the estimation and the resulting consistency level computations are performed: the Harmony estimation model, which is based on probabilistic computations, predicts the stale read rate in accordance to the statics fed by the monitoring module. Accordingly, it chooses whether to select the basic consistency level ONE (involving only one replica) or else, computes the number of involved replicas necessary to maintain an acceptable stale reads rate while allowing a better performance. More details about Harmony can be found in [33].

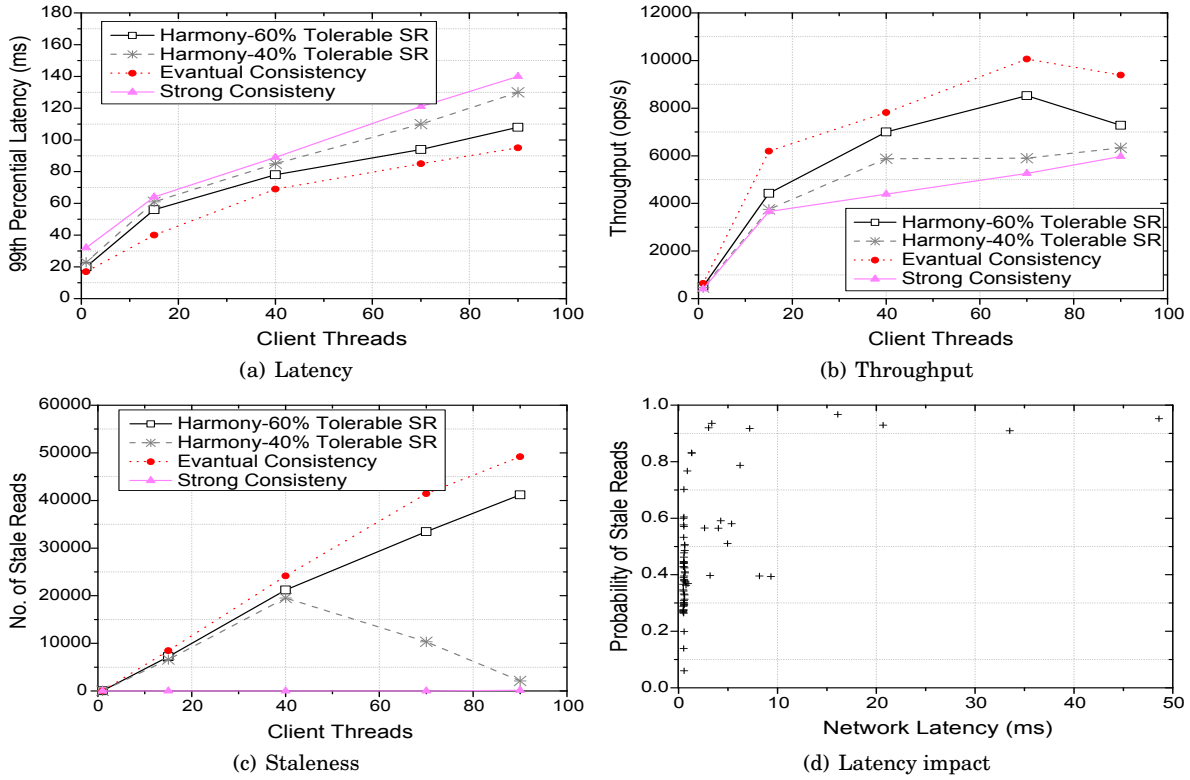

Figure 1.4: Harmony with S% tolerable stale reads (Harmony-S% Tolerable SR) against strong and eventual consistency on Amazon EC2.

Harmony Evaluation

In order to validate Harmony, a set of experimental evaluations was conducted on Amazon Elastic Cloud Compute (EC2) [2]. For all experiments we ran Apache Cassandra-1.0.2 as an underlying storage system, and used the Yahoo Cloud Serving Benchmark! (YCSB) [47]. We deployed Cassandra on 20 VMs on Amazon EC2. The goal of these experiments is to evaluate system performance and measure the staleness rate. We used Workload-A: a heavy read-update workload from YCSB! with two data sets with the size of 23.85 GB for EC2 and a total of 5 million operations.

Throughput & Latency: We compare Harmony with two settings (two different tolerable stale read rates of an application) with strong and eventual consistency. The first tolerable stale read rate is 60% for Amazon EC2 (this rate tolerates more staleness in the system implying lower consistency levels and thus less waiting time), and the second tolerable stale read rate is 40% for Amazon EC2 (this rate is more

restrictive than the first one, meaning that the number of read operations performed with a higher level of consistency is larger). Network latency is high and varies in time in Amazon EC2 (we observe it is 5 times higher than in a local cluster). We run workload-A while varying the number of client threads.

Figure 1.4(a) presents the 99th percentile latency of read operations when the number of client threads increases on EC2. The strong consistency approach provides the highest latency having all reads to wait for responses from all replicas that are spread over different racks and clusters. Eventual consistency is the approach that provides the smallest latencies since all read operations are performed on one local replica (possibly at the cost of consistency violation). We can clearly see that Harmony with both settings provides almost the same latency as a basic static eventual consistency. Moreover, the latency increases by decreasing the tolerable stale reads rate of an application as the probability of stale read can easily get higher than these rates, which requires higher consistency levels and, as a result, a higher latency.

In Figure 1.4(b), we show the overall throughput for read and write operations with different numbers of client threads. The throughput increases as the number of threads increases. However, the throughput decreases with more than 90 threads. This is because the number of client threads is higher than the number of storage hosts and threads are served concurrently. We can observe that the throughput is smaller with strong consistency. This is because of the extra network traffic generated by the synchronization process as well as the high operation latencies. We can notice that our approach with a stale reads rate of 60%, provides very good throughput that can be compared to the one of static eventual consistency approach. While exhibiting high throughputs, our adaptive policies provide fewer stale reads as higher consistency levels are chosen only when it matters.

Staleness: In Figure 1.4(c) , we show that Harmony, with both policies with different application tolerated stale reads rates, provides less stale reads than the eventual consistency approach. Moreover, we can see that, with a more restrictive tolerated stale reads rate, we get a smaller number of stale reads. We observe that with rates

Table 1.4: Consistency in cloud storage system: Taxonomy

	Datacenter level		**Wide-area level**	
	Lock	**Lock-free**	**Lock**	**Lock-free**
Eventual Consistency	MongoDB	Dynamo, Cassandra, Riak, Voldemort, CouchDB, S3	BigTable, HBase	Dynamo, Cassandra, Riak, Voldemort, CouchDB, S3
Timeline Consistency	PNUTS		PNUTS	
Strong Consistency	Bigtable, HBase	VoltDB	Scatter, Spanner, Megastore, Microsoft Azure Storage	
Adaptive Consistency	Gemini, Consistency Rationing, Harmony			

of 40%, the number of stale reads decreases when the number of threads grows over 40 threads. This is explained by the fact that with more than 40 threads the estimated rate grows higher than 40%, for most of the run time due to concurrent accesses, and higher consistency levels are chosen, thus decreasing the number of stale reads.

In order to see the impact of network latency on the stale reads estimation we ran workload-A –varying the number of threads starting with 90 threads, then, 70, 40, 15 and finally, one thread– on Amazon EC2 and measure the network latency during the run-time. Figure 1.4(d) shows that high network latency causes higher stale reads regardless of the number of the threads (higher latency dominates the probability of stale reads), while when the latency is small, the access pattern has more influence on the probability.

1.6 Conclusion

This chapter addresses a major open issue in cloud storage systems: the management of consistency for replicated data. Despite a plethora of cloud storage systems available today, data consistency schemes are still far from satisfactory. We take this opportunity to ponder the ***CAP*** theorem 13 years after it's formulation and discuss its implications in the modern context of cloud computing. The tension between Consistency, Availability and Partition Tolerance has been handled in various ways in existing distributed storage systems (e.g., by relaxing consistency at wide-area level).

We provide an overview of the major consistency models and approaches used for providing scalable yet highly available services on clouds. Cloud storage is foundational to cloud computing because it provides a backend for hosting not only user data but also the system-level data needed by cloud services. While discussing state-of-art cloud storage systems used by the main cloud vendors, we advocate self-adaptivity as a key means to approach the tradeoffs that must be handled by the user applications. We review several different approaches to adaptive consistency that provide a flexible consistency management for users to reduce performance overhead and monetary cost when data are distributed across geographically distributed sites. Then, we discuss in details Harmony: an automated approach to adaptive consistency evaluated with the Cassandra cloud storage on Amazon EC2. The detailed review of the aforementioned approaches led us to the definition of a taxonomy of different cloud storage systems and the consistency model they have adopted according to their target environment (intra- and inter-sites).

References

[1] H. Jin, S. Ibrahim, T. Bell, L. Qi, H. Cao, S. Wu, and X. Shi, "Tools and technologies for building the clouds," *Cloud computing: Principles Systems and Applications*, pp. 3–20, Aug. 2010.

[2] "Amazon Elastic Compute Cloud (Amazon EC2)," February 2013. [Online]. Available: http://aws.amazon.com/ec2/

[3] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," *SIGOPS - Operating Systems Review*, vol. 37, no. 5, pp. 29–43, 2003.

[4] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," in *Proceedings of the 7th conference on usenix symposium on operating systems design and implementation*, 2006, pp. 205–218.

[5] S. Ibrahim, H. Jin, L. Lu, B. He, G. Antoniu, and S. Wu, "Maestro: Replica-aware map scheduling for mapreduce," in *Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2012)*, Ottawa, Canada, 2012, pp. 59–72.

[6] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues, "Making geo-replicated systems fast as possible, consistent when necessary," in *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, ser. OSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 265–278.

[7] "Facebook Statistcs," February 2013. [Online]. Available: http://newsroom.fb.com/content/default.aspx?NewsAreaId=22

[8] H. Wada, A. Fekete, L. Zhao, K. Lee, and A. Liu, "Data consistency properties and the trade-offs in commercial cloud storage: the consumers' perspective," in *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*, 2011, pp. 134–143.

[9] E. A. Brewer, "Towards robust distributed systems (abstract)," in *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, ser. PODC '00. New York, NY, USA: ACM, 2000, pp. 7–.

[10] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent available partition-tolerant web services," in *ACM SIGACT News*, 2002, p. 2002.

[11] E. Brewer, "Cap twelve years later: How the "rules" have changed," *Computer*, vol. 45, no. 2, pp. 23 –29, feb. 2012.

[12] D. J. Abadi, "Consistency tradeoffs in modern distributed database system design: Cap is only part of the story," *Computer*, vol. 45, pp. 37–42, 2012.

[13] S. Gilbert and N. Lynch, "Perspectives on the cap theorem," *Computer*, vol. 45, no. 2, pp. 30 –36, feb. 2012.

[14] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency control and recovery in database systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1987.

[15] M. P. Herlihy and J. M. Wing, "Linearizability: a correctness condition for concurrent objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, Jul. 1990.

[16] M. Dubois, C. Scheurich, and F. Briggs, "Memory access buffering in multiprocessors," in *25 years of the international symposia on Computer architecture (selected papers)*, ser. ISCA '98. New York, NY, USA: ACM, 1998, pp. 320–328.

[17] C. Scheurich and M. Dubois, "Concurrent miss resolution in multiprocessor caches," in *ICPP (1)*, 1988, pp. 118–125.

[18] S. V. Adve and M. D. Hill, "Weak ordering - a new definition," *SIGARCH Comput. Archit. News*, vol. 18, no. 3a, pp. 2–14, May 1990.

[19] R. Bisiani, A. Nowatzyk, and M. Ravishankar, "Coherent shared memory on a distributed memory machine," in *ICPP (1)*, 1989, pp. 133–141.

[20] W. Vogels, "Eventually consistent," *Commun. ACM*, pp. 40–44, 2009.

[21] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto, "Causal memory: Definitions, implementation, and programming," *Distributed Computing*, vol. 9, no. 1, pp. 37–49, 1995.

[22] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Don't settle for eventual: scalable causal consistency for wide-area storage with cops," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11. New York, NY, USA: ACM, 2011, pp. 401–416.

[23] M. Shapiro and B. Kemme, "Eventual Consistency," in *Encyclopedia of Database Systems (online and print)*, M. T. Ózsu and L. Liu, Eds. springer, 2009.

[24] A. S. Tanenbaum and M. v. Steen, *Distributed Systems: Principles and Paradigms (2nd Edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2006.

[25] Y. Saito and M. Shapiro, "Optimistic replication," *ACM Comput. Surv.*, vol. 37, no. 1, pp. 42–81, Mar. 2005.

[26] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver,

and R. Yerneni, "Pnuts: Yahoo!'s hosted data serving platform," *Proc. VLDB Endow.*, vol. 1, no. 2, pp. 1277–1288, Aug. 2008.

[27] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, ser. SOSP '07. New York, NY, USA: ACM, 2007, pp. 205–220.

[28] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web," in *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, ser. STOC '97. New York, NY, USA: ACM, 1997, pp. 654–663.

[29] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, pp. 35–40, April 2010.

[30] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, ser. SIGCOMM '01. New York, NY, USA: ACM, 2001, pp. 149–160.

[31] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: wait-free coordination for internet-scale systems," in *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, ser. USENIXATC'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 11–11.

[32] "About Data Consistency in Cassandra," February 2012. [Online]. Available: http://www.datastax.com/docs/1.0/dml/data_consistency

[33] H.-E. Chihoub, S. Ibrahim, G. Antoniu, and M. S. Pérez-Hernández, "Harmony: Towards automated self-adaptive consistency in cloud storage," in *2012 IEEE International Conference on Cluster Computing (CLUSTER'12)*, Beijing, China, 2012, pp. 293–301.

[34] "Riak," February 2013. [Online]. Available: http://basho.com/riak/

[35] "Voldemort," February 2013. [Online]. Available: http://www.project-voldemort.com/voldemort/

[36] "Apache CouchDB," February 2013. [Online]. Available: http://couchdb.apache.org/

[37] "mongoDB," February 2013. [Online]. Available: http://www.mongodb.org/

[38] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh, "Megastore: Providing scalable, highly available storage for interactive services," in *Proceedings of the Conference on Innovative Data system Research (CIDR)*, 2011, pp. 223–234.

[39] "Redis," February 2013. [Online]. Available: http://redis.io/

[40] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. u. Haq, M. I. u. Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas, "Windows azure storage: a highly available cloud storage service with strong consistency," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11. New York, NY, USA: ACM, 2011, pp. 143–157.

[41] "Apache HBase," February 2013. [Online]. Available: http://hbase.apache.org/

[42] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, "Spanner: Google's globally-distributed database," in *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, ser. OSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 251–264.

[43] T. Kraska, M. Hentschel, G. Alonso, and D. Kossmann, "Consistency rationing in the cloud: pay only when it matters," *Proc. VLDB Endow.*, vol. 2, pp. 253–264, August 2009.

[44] "Amazon Simple Storage Service (Amazon S3)," February 2013. [Online]. Available: http://aws.amazon.com/s3/

[45] "Amazon Simple Queue Service (Amazon SQS)," February 2013. [Online]. Available: http://aws.amazon.com/sqs/

[46] "Apache Cassandra," February 2012. [Online]. Available: http://cassandra.apache.org/

[47] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*, ser. SoCC '10. New York, NY, USA: ACM, 2010, pp. 143–154.