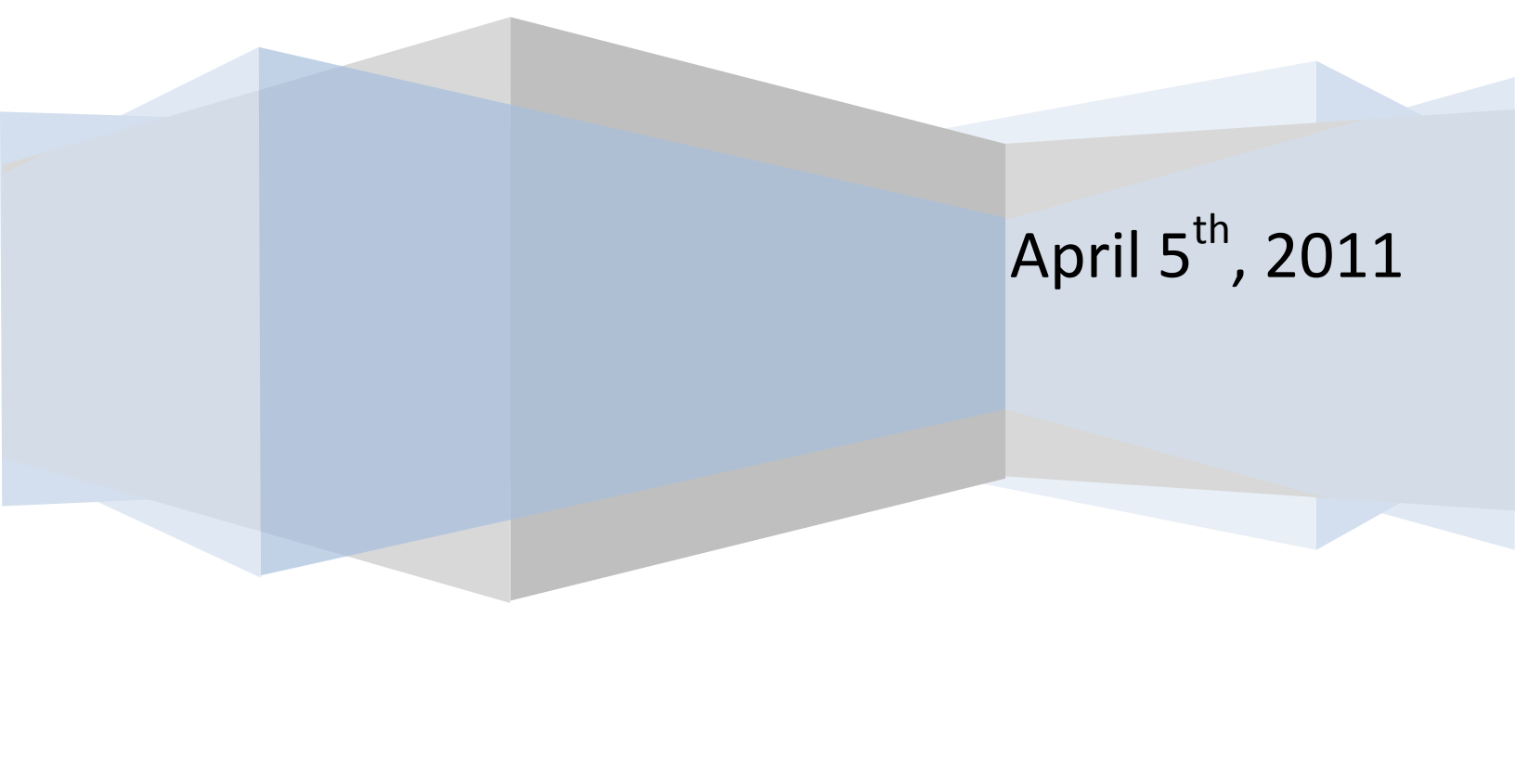


The CAP Theorem as it Applies to Contemporary NoSQL Storage Systems

Andrew Carter



April 5th, 2011

Abstract

NoSQL storage systems are becoming increasingly popular and important as the demand for web-scale performance in today's internet based applications grows. NoSQL storage systems are typically distributed systems and as such fall victim to the restrictions placed on this class of system. Dr. Eric Brewer attempted to classify these restrictions by introducing the notion of the CAP theorem in 2000. This paper explores the CAP theorem and its implications on distributed systems. Additionally, some popular NoSQL systems are presented and analyzed based on the trade-offs they make due to CAP principles.

List of Acronyms

2PC – Two Phase Commit

ACID – Atomicity, Consistency, Isolation, Durability

ASF – Apache Software Foundation

BASE – Basically Available, Soft-state, Eventually-consistent

CAP – Consistency, Availability, Partition-tolerance

DFS – Distributed File System

DNS – Domain Name Service

NoSQL – Not Only SQL (Structured Query Language)

OSDI – Operating Systems Design and Implementation

RDBMS – Relational Database Management System

SOSP – Symposium on Operating Systems Principles

Table of Figures

Figure 1 - Bigtable instance with dependencies shown.....	5
Figure 2 - Tablet location hierarchy	6
Figure 3 - Object versioning via vector clocks example	11

Table of Contents

Abstract	
List of Acronyms	ii
1. Introduction	1
2. Background Information	1
2.1 The CAP Theorem Explained	1
2.1.1 Consistency	1
2.1.2 Availability	2
2.1.3 Partition Tolerance	2
2.2 CAP Classifications Applied to Known Systems	2
2.3 Implications of the CAP Theorem	3
2.3.1 ACID vs. BASE	3
2.3.2 CAP theorem represents a spectrum	3
2.4 NoSQL Systems vs. RDBMS	4
2.4.1 Typical NoSQL System Data Models	4
3. Discussion	5
3.1 CP Systems	5
3.1.1 Bigtable	5
3.1.2 HBase	7
3.1.3 Hypertable	8
3.2 AP Systems	8
3.2.1 Dynamo	8
3.2.2 Other AP Systems	12
3.3 Adjustable Systems	12
3.3.1 Cassandra	12
3.4 Application Areas for NoSQL Systems	13
3.4.1 Case Study: Amazon Shopping Cart Service	13
3.5 Is the CAP Theorem Adequate?	13
3.5.1 Imperfections with CAP classifications	14
4. Conclusions	14
References	15

1. Introduction

The CAP conjecture was described in 2000 by Eric Brewer of Berkley, and was later formally proven by Seth Gilbert and Nancy Lynch of MIT making it a full-blown theorem [1]. CAP is an acronym with the letters standing for Consistency, Availability and Partition-tolerance. The theorem states that a distributed system can satisfy two of these properties at the same time but that no distributed system can simultaneously satisfy all three.

Seeing how many popular very high load storage systems of the present era are distributed systems, they fall victim to the implications of this theorem. This paper will explain the CAP theorem and explore the theorem as it relates to several popular “Not Only SQL” (NoSQL) storage systems.

2. Background Information

2.1 The CAP Theorem Explained

The following sections describe the three attributes associated with the CAP theorem.

2.1.1 Consistency

Consistency implies that any given node of a database system will appear to contain the same information as any other node at all times. For example, if I have a single instance of a MySQL database, it is trivially consistent as that instance is the only store for the data. However, if I have two MySQL nodes in multi-master mode, consistency cannot be taken for granted. In order for there to be complete consistency each write to the database must be replicated to the other node before the write operation is considered complete. It is easy to see in this scenario that as the number of replicas goes up, there is a larger impact on performance for write operations as more node communication and replications must occur.

There are many definitions (with varying degrees of granularity) defining the consistency properties of a given system. We will consider two types of consistency:

- Strong consistency – when a write is performed the system guarantees subsequent reads from any node will produce the written value.
- Eventual consistency – performing a read operation may give stale data to the client, but all writes will eventually propagate through the entire system.

2.1.2 Availability

Availability is a fairly self-explanatory trait. It refers to how available the data stored in a database is for consumption. This is usually considered in the context of a failure scenario. For example, let's say that I have a single MySQL database instance. If this instance were to fail my availability would go from 100% to 0%. However, if I had two instances with the data split between them, if one failed my data availability would go from 100% to 50%. Therefore we can see that increasing the number of nodes in a system, which have replicas of the data, directly increases the availability of the system. Additionally, there are other aspects of a system encapsulated within the concept of availability, such as latency time from request to response for a given query. In general, availability means just that; how available is your data in any given situation?

2.1.3 Partition Tolerance

Partition Tolerance describes a systems ability to continue normal operation in the event of a disconnection between nodes (or node clusters) which are part of the system. Let us say for example that we have two data warehouses, one in Waterloo and one in St. John's. If the network connection which links these two data centers were to go down we would no longer be able to have all nodes of the system communicate. Therefore, the system would be in effect partitioned. Partition tolerance describes a systems ability to continue functioning correctly in this scenario. In their paper proving the CAP conjecture, Gilbert and Lynch defined partition tolerance as so: "No set of failures less than total network failure is allowed to cause the system to respond incorrectly" [1].

2.2 CAP Classifications Applied to Known Systems

Before continuing it is beneficial to get an idea where some systems which are ubiquitous within the internet fit within CAP classifications. In the current internet landscape we have a mixture of different distributed systems which make various trade-off's based on the CAP theorem [2].

- Some CA systems include: single site databases, cluster databases and LDAP. Characteristics commonly shared by these systems are: 2-phase commits (2PC) and the use of cache validation protocols.
- Some CP Systems include: distributed databases, distributed locking and majority protocols. Traits shared by these systems include: pessimistic locking and the behaviour that minority partitions are made unavailable.
- Some AP Systems include: Web caching systems and the Domain Name System (DNS). Common traits are: expirations/leases of cached data and the fact that conflict resolution operations are required on conflicting data versions.

2.3 Implications of the CAP Theorem

The big revelation made by Brewer in 2000 was that people need to stop worrying about data consistency in all internet scale applications [2]. If we want tolerance to network partitions and high levels of availability in these new distributed applications, then guaranteed consistency of data is something we cannot have. This paradigm shift in the way we think about applications interacting with data helped usher in a new way of thinking.

2.3.1 ACID vs. BASE

Database research is generally focused on creating systems which adhere to the ACID properties. However, when considering availability in favour of consistency it is useful to change our way of thinking about the desirable properties of a storage system. We trade the 'C' and 'I' in ACID for availability, performance and graceful degradation. Thus, Brewer states that when we are thinking about data stores useful for solving internet scale problems we should consider it's adherence to BASE. BASE stands for:

- Basically Available
- Soft-state
- Eventual consistency

With a system which adheres to BASE we are not guaranteed to always have fresh data at all nodes. Rather, we accept that given a sufficiently long period of time over which no updates are sent, we can expect that updates will propagate through the system to eventually achieve consistency. Brewer believes that these are the desirable properties to strive and design for if web-scale performance is required from your distributed storage system.

2.3.2 CAP theorem represents a spectrum

One important thing to note about the CAP theorem is that the relationship between the three attributes represent a spectrum, it is not discrete relationship. For example, in many systems it is possible to give up a little consistency to gain a bit more availability. Fundamentally however, when a system is designed the designer chooses two of the three attributes to focus on. As we will see, systems such as Dynamo were fundamentally designed to provide high availability and partition-tolerance with less emphasis put on strong-consistency of the data across nodes. Clearly there is some consistency eventually in Dynamo; there is just no *guaranteed strong* consistency. So, when creating a system which incorporates Dynamo as a data store one must be aware of this issue.

2.4 NoSQL Systems vs. RDBMS

NoSQL has emerged recently as a term for describing any database system that does not use a relational model. Due to the explosion of data intensive web applications, which must handle data flow at an enormous scale, interest in this type of system has been exponentially increasing in the last few years. The problem with RDBMS's for handling this type of application environment is that their methods for scaling are typically vertical (i.e. add more expensive hardware to a single node), while their ability to scale horizontally (i.e. add more nodes with cheap hardware), is typically clumsy.

NoSQL systems on the other hand typically have the ability to scale horizontally as a primary design objective. They sacrifice sophistication and control at the database level for the ability to scale easily and efficiently based on the needs of the application. The control logic and data organization features that are classically found in the database level are now being relocated to the application layer when needed. Additionally, applications which typically use these NoSQL systems have fairly simple requirements and do not need to do much manipulation on the data they consume. Thus, they have no need for fully featured relational algebra support amongst other RDBMS niceties.

2.4.1 Typical NoSQL System Data Models

NoSQL systems do not store their data in a typical relational format. Instead, a taxonomy of data models has emerged within the rapidly growing landscape of NoSQL systems. The two data models most pertinent to this paper are key-value data models and tabular data models. Each of the systems discussed in this report use a data model which falls into one of these two categories.

In a key-value approach, data is stored in a structure where a given key is associated with a given value (or set of values). The acceptable data types for the key and value vary. There are several other terms which more or less describe the same concept such as associative array, map, dictionary, etc. With reference to relational systems, this is analogous to a system which has a giant table with three columns, entity-attribute-value, where the key is defined by the entity-attribute pair [3].

With tabular data structures, you are able to store what are essentially rows, but you do not need to worry whether or not each row has values for the same set of columns. Instead, NULL values are used liberally throughout the large table to fill in the gaps. This approach is similar to that found in relational systems, except that there is no schema defined at the database level.

3. Discussion

3.1 CP Systems

CP systems give up high availability levels in favour of consistency and partition-tolerance. This is generally achieved by ensuring that all replicas of a data object within a system must be updated upon a write before subsequent reads or writes are allowed to occur. The subsections below provide analysis on how the specific database systems mentioned provide consistency and availability. Generally, CP systems are best used when consistency is an absolute requirement (i.e. checkout services, or backend systems which are not required to instantly respond to an end user).

3.1.1 Bigtable

The concept for Bigtable was described by Google in the paper “Bigtable: A Distributed Storage System for Structured Data”, which was presented at the Operating Systems Design and Implementation (OSDI) conference in 2006 [4]. The data model used within Bigtable falls into the tabular category; it is described in the paper as a “sparse, distributed, persistent multi-dimensional sorted map”. For a detailed description of the data model used in Bigtable please see this article [5], as an explanation is outside the scope of this paper. Bigtable is designed as a high performance storage solution for many of Google’s products and services.

Bigtable is built upon some underlying technologies: Chubby (which implements Paxos to provide distributed locking) and Google File System (GFS) a distributed file system (stores log and data files). The logical layout of Bigtable and its dependencies can be seen in Figure 1.

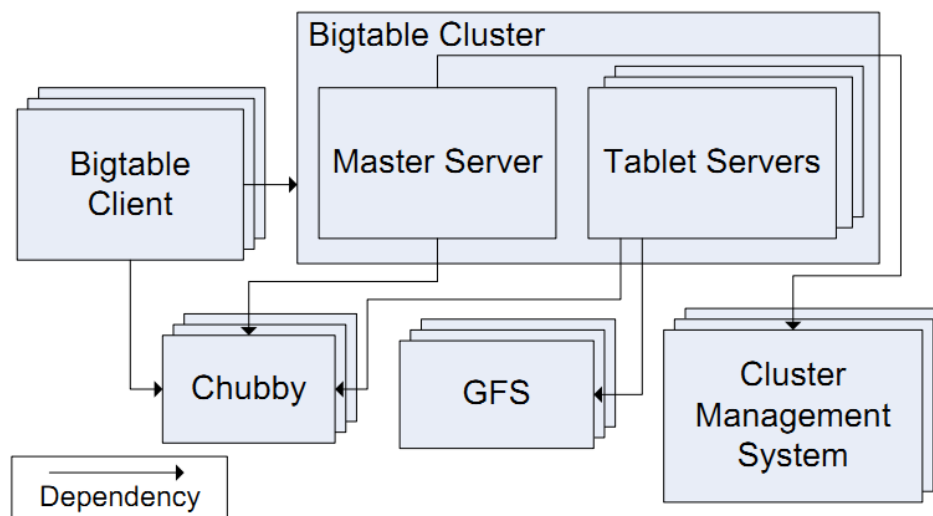


Figure 1 - Bigtable instance with dependencies shown

Each table within a Bigtable cluster consists of a set of tablets, and each tablet is responsible for all data associated with a specific row range. When a table is first created, it consists of just one tablet but as it grows, it is split into multiple tablets with each tablet being roughly 100-200MB in size. The logical tablet hierarchy can be seen in Figure 2. Chubby holds onto the location of the first tablet in the tablet hierarchy. This root tablet contains links to all tablets in the next layer, the metadata layer. Tablets in the metadata layer are responsible for maintaining links to tablets in the base layer which are in turn responsible for holding actual user data in their rows.

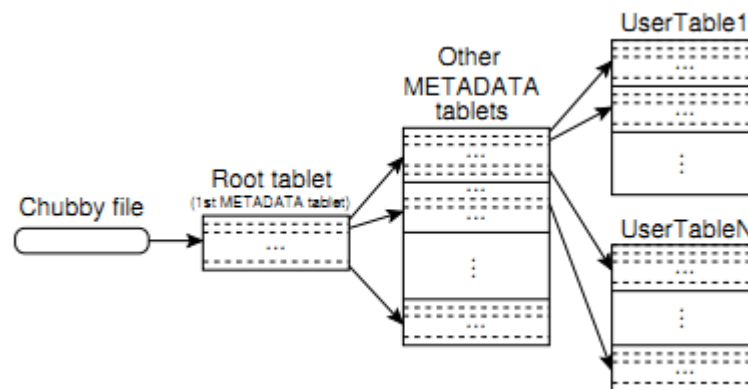


Figure 2 - Tablet location hierarchy

The key aspects of the Bigtable system that cause availability to be traded for consistency are the dependency upon the Chubby locking service to perform many key functions of the system and the replication of data which occurs upon writes in the underlying GFS.

3.1.1.1 Chubby usage in Bigtable

There are several tasks within Bigtable which Chubby helps facilitate. These include: ensuring there is at most one active master server, storing the bootstrap location of Bigtable data, discovering new tablet servers, confirming tablet server deaths, storing Bigtable schema information and storing access control lists. Therefore, the Bigtable system relies on the Chubby service to maintain consistency within its subsystems. As a result, if the Chubby service becomes unavailable due to network issues or outages, the Bigtable system becomes unavailable. In tests run by Google this has not been much of an issue for them, but nevertheless it does occur.

3.1.1.2 GFS usage in Bigtable

Another issue which is not described in much detail in the referenced paper is the underlying data replication which occurs at the GFS level. Availability of the system is ultimately impacted by the consistency requirements in place at this level. GFS is a highly distributed system and replicates data chunks as part of its normal mode of operation [6]. When writes are processed through the system and ready to be written to disk, commit logs are written to ensure recoverability and data is also written and replicated by the underlying GFS. Thus operations which access the GFS layer are slower due to an affinity for strong consistency, negatively impacting availability.

3.1.1.3 Bigtable distribution across multiple data centers

In addition to these issues, in the paper the system is described as it functions within a single data center. They mention that as of the writing, they are currently working on implementing support for cross-data-center replicated Bigtables with multiple master replicas. Once they allow a Bigtable installation to span multiple data centers, CAP trade-offs will undoubtedly come into play again. If they chose to have their system maintain strong consistency in a multi-site configuration, availability will inevitably be impacted. This will be particularly acute during network partitions, due to a communication breakdown among nodes slowing or blocking the use of replication mechanisms.

3.1.2 HBase

HBase is a distributed storage system which closely resembles Google's Bigtable. Its design and implementation were directly inspired by the Bigtable paper discussed previously. It was originally created at Powerset in 2007, with a lot of subsequent work being done by several other companies, notably StumbleUpon [7]. It is currently a project under the Apache Software Foundation (ASF).

Due to its resemblance to Bigtable it shares a lot of commonality. For instance, HBase is built on top of ZooKeeper (Locking service, Paxos algorithm) and Hadoop DFS (Distributed File System), these are open source analogs of the Chubby and GFS systems Google uses. HBase and Bigtable both use master to describe their master servers, but in the HBase parlance Bigtable tablets are referred to as region servers.

Like Bigtable, HBase is designed to provide strong consistency. Therefore, it also runs into problems providing high availability in the event of network partitions for similar reasons as those presented in the Bigtable section. Due to the similarity between the two systems further discussion of CAP with regards to HBase will be forgone.

3.1.3 Hypertable

Hypertable is yet another storage system which is modeled after Google's Bigtable; it was originally developed by Zvents. It is written almost entirely in C++ in order to provide a high level of performance [8].

Hypertable is also built on top of a DFS and a distributed locking service. In Hypertables case the locking service is called Hyperspace. As for the DFS portion, Hypertable is interesting in this regard; it allows the user to select which DFS to use and manages its use through another layer, the DFS broker. As with HBase the similarity between Hypertable and Bigtable means that the CAP trade-offs involved are similar, thus they will not be discussed in detail.

3.2 AP Systems

AP systems give up strong consistency in favour of high availability and partition-tolerance. This is generally achieved by allowing stale data to be read from a system, but eventually propagating and reconciling all divergent versions of data objects which have been written. The subsections below provide analysis on how the specific database systems mentioned provide high levels of availability even in the event of a network partition. Generally, AP systems are best used when consistency can be sacrificed for quick response of the system even under failure (i.e. services whose availability directly affects customer experience).

3.2.1 Dynamo

Dynamo is a highly-available key-value storage system used by some of Amazon's core services, such as the shopping cart service. It is not an open-source or publically available solution. The paper describing Dynamo was first presented at the 2007 Symposium on Operating Systems Principles (SOSP) [9]. Additionally, Amazon claimed they had been using the system in production for some time before publishing the paper. In contrast to the systems discussed in the CP Systems section, Dynamo is a system which trades strong consistency in order to achieve high availability even in the event of a network partition.

Dynamo was created by Amazon to help fulfill a business need for a system which could manage the state of services which have extremely high reliability requirements. Additionally, these services need a high degree of control over the trade-offs between availability, consistency, cost-effectiveness and performance. Dynamo acts as a proof of concept that with some clever design, an eventually-consistent storage solution can be used in a production system within an extremely demanding, performance-oriented environment.

3.2.1.1 Dynamo Design Decisions

Given the eventually-consistent nature of Dynamo, it is clear that the engineering design team had to address the issue of uncertainty in query results. Traditionally, in strongly-consistent systems as those discussed above, the systems chose not to make the data available to the client until there is absolute certainty in the result. This in turn affects availability of the data as latency and possible network partitions delay the serving of the data. The Dynamo philosophy on the other hand is to allow uncertainty in the answer if resolving the uncertainty would negatively affect the availability of the data. Thus, consistency is sacrificed in the name of availability.

Another design choice which had to be made by Dynamo engineers was when to handle the resolution of conflicts within the system. Based on the eventual-consistency approach being employed, data version conflicts between nodes are inevitable. Based on an analysis of the system use cases, Dynamo engineers decided it would be best to handle conflict resolution upon reads from the system. This approach was deemed to provide a better level of customer experience. The reason being latency impact upon a write request is more noticeable to the user, thus it negatively impacts perceived system availability.

3.2.1.2 Dynamo System Interface

The programming model provided to application developers by Dynamo is harder to design for than the model provided by strongly-consistent relational systems. However, relational systems are limited in their abilities to provide availability and scalability to the levels required by web-scale applications.

The system interface for Dynamo consists of two basic operations, `get()` and `put()`. When performing a `get(key)` operation, Dynamo locates the object replicas associated with the key argument and returns the singular object if there are no conflicting versions, or in the case of conflicts, and object list with conflicting versions and a context. The `put(key, context, object)` operation is handled by finding the appropriate nodes to store the replicas of the object based on the key, and then placing the replicas in these nodes. The context object is used to store metadata about the object being stored such as its version history (discussed later).

3.2.1.3 Partitioning

Dynamo is partitioned via the use of consistent hashing. In consistent hashing the output results of a hash function represent a fixed circular space (such that the largest hash value wraps around to the smallest hash value). In Dynamo, a variant of consistent hashing is used to assign each of the active

nodes in the system to multiple points in the ring. These multiple points are referred to as “virtual nodes”, each physical node can have multiple virtual nodes.

When a key is passed to Dynamo, it is hashed and the hash result represents a position on the circular ring. Each virtual node is responsible for the range of possible hash results between itself and its predecessor node on the ring. The use of virtual nodes within the Dynamo has several advantages. For one, it allows for heterogeneous hardware to be used for nodes; a machine which is less capable will be assigned fewer virtual nodes. Additionally, it is easy for the load handled by a node to be evenly dispersed amongst other nodes if it becomes unavailable.

3.2.1.4 Consistency Model

Dynamo provides an eventual-consistency guarantee. When a put() call is made the system does not ensure that all nodes replicating the data are consistent before serving subsequent get() calls for the same data. Thus there are scenarios where a get() call may return stale data. In the absence of system failures Dynamo does propagate updates thorough the system in a bounded time, however during certain failure scenarios updates may not get propagated within the bounded timeframe [9].

As was mentioned previously, Dynamo stores versions of the object being modified by put() commands. This allows divergent versions of an object to be reconciled once the updates propagate through the system. Vector clocks are used to determine whether object versions need to be reconciled by the client. Vector clocks work by assigning a (node, counter) pair with every version of a given object. It can be determined if an object is an ancestor of another by comparing the counters of the first objects clock with that of the second. If two nodes write the same object before the update is able to propagate between them, the node counter pairs will indicate this. Upon the next read of the object the client will have to reconcile the object difference. The node counter pairs are stored in the context object mentioned previously.

An example of an update path which results in a conflicted object is presented in Figure 3. D1-5 represent different versions of the same object and Sx, Sy, Sz represent different nodes in the system. As can be seen, in the third step nodes Sy and Sz both update D2 before the update by Sx was able to propagate, the fourth step shows the resulting reconciled object after it has been read and written by node Sx again.

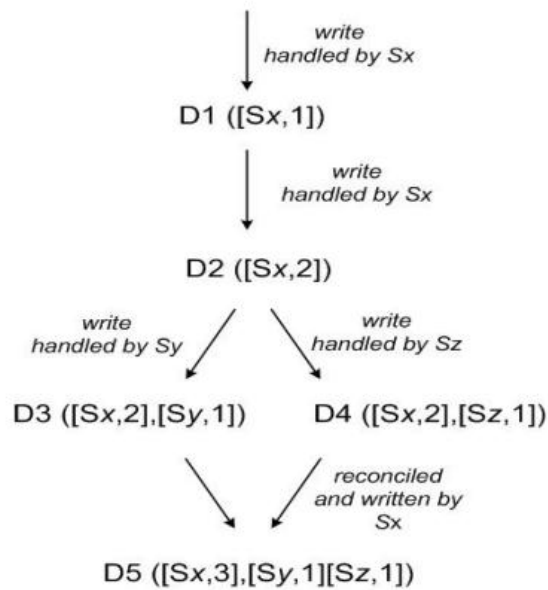


Figure 3 - Object versioning via vector clocks example

3.2.1.5 Handling Failures

Dynamo does not use a traditional quorum approach for handling failures as if it did, it would be unavailable during network partitions and server failures. Instead, Dynamo uses a “sloppy quorum” and hinted handoff approach. In the event of a node failure, a surrogate node will receive a replica of the data object the failed node is responsible for upon a write request. This node will then contain a hint in its metadata indicating the node the data was initially intended for. The hinted replica is kept in a separate local database of the surrogate node which is scanned periodically. Once the failed node has recovered, the surrogate node will attempt to transfer the replica to the failed node. If this is successful the surrogate node will delete the replica data, yet the number of replicas of the object in the system will remain the same. The number of replicas for each object in the system is configurable and its configured value is based on a trade-off between availability and durability.

3.2.1.6 CAP trade-offs in Dynamo

It is clear that given the eventual-consistency model, and the lack of a traditional quorum approach for failure handling, Dynamo engineers choose to heavily favour availability over consistency. Additionally,

Dynamos design allows it to remain highly available in the event of node failures or network partitions via hinted handoff and a modified consistent hashing technique.

3.2.2 Other AP Systems

There are several other NoSQL systems which fall under the branch of AP Systems. Two worth noting are Voldemort and Riak. A discussion of the CAP trade-offs present in their design will not be presented.

3.3 Adjustable Systems

Adjustable systems are systems which, through configuration, can provide varying levels of consistency or availability, while still maintaining a tolerance to network partitions. These “jack of all trades” systems provide flexibility to the application designer.

3.3.1 Cassandra

Cassandra is an Apache Software Foundation (ASF) project originally created by Facebook to address their needs to store massive amounts of data in a performance-critical environment [10]. It was conceived and designed by some of the engineers who created Dynamo. They were recruited from Amazon by Facebook to work on the development of Cassandra.

Cassandra is an interesting anomaly in the current world of NoSQL systems in that it has an aspect of its design which allows it to be “tuned” between offering strong consistency or high availability. The way in which this is accomplished is by providing the ability to define a “ConsistencyLevel” when performing a read or write operation. ConsistencyLevel is an enumeration provided by the Cassandra API which defines how many nodes must take part in a read or write call before the call returns to the sender.

This is best illustrated with an example. Let W represent the number of nodes to block for on a write call, and R represent the number of nodes to block for on reads. If, $W + R > \text{ReplicationFactor}$ of the system, then you will have strongly consistent behaviour, i.e. readers will always see the most recent write [11]. This situation is represented by the ConsistencyLevel value of ALL. If a medium level of consistency is desired, the ConsistencyLevel of QUORUM can be used. With QUORUM, a call will block until it has agreement amongst $N/2 + 1$ of the participating nodes on an objects value where N is the ReplicationFactor. For the lowest level of consistency we have the ConsistencyLevel of ONE, where we require only one node to perform the operation. In the ONE case we have the highest likelihood of introducing stale data into the results, but we achieve the lowest latency response times. It is worth noting that the increased availability you are getting from this system is in the form of reduced latency time for the read and write calls to complete.

A discussion on the architectural features of Cassandra will not be presented as they will do little to aid in the analysis of the CAP trade-offs present in Cassandra. Additionally, it would be redundant as Cassandra, from an architectural point of view, borrows heavily from Dynamo and lightly from Bigtable.

3.4 Application Areas for NoSQL Systems

As was discussed earlier NoSQL systems are not intended to replace the RDBMS entirely. Instead they were created to address problem spaces which RDBMSs are not designed to solve. Following is a case study of how a NoSQL system is being used in a production environment.

3.4.1 Case Study: Amazon Shopping Cart Service

Amazon has designed their shopping cart application to work in the eventually-consistent environment provided by their Dynamo storage system. To provide a pleasant user experience the shopping cart application must never forget or reject an “add to cart” operation. Additionally, it must respond to user actions quickly.

To accomplish this, the application never rejects an “add to cart” or “delete item from cart” request, even if the most current version of the shopping cart is unavailable to the application. Instead it adds or deletes from an older version of the cart (both add and delete are sent as put requests to the Dynamo storage system) and reconciles the difference between cart versions at a later point in time. Often, graceful merging is able to occur between the two divergent versions of the users shopping cart. However, in some rare cases, when failures occur in parallel with concurrent operations the system must rely on the user to merge differing object versions. This manual merging is handled in the following way, “add to cart” operations are never forgotten by the system, but it is possible for deleted items to reappear in a user’s shopping cart, at which point they would recognize them and delete them again. This is seen as an acceptable trade-off for the high level of availability seen throughout the rest of the user experience.

3.5 Is the CAP Theorem Adequate?

We have seen examples throughout this paper of the CAP theorem being used to explain the engineering trade-offs inevitable in the design of distributed systems. The CAP theorem is certainly a useful tool for analysing these trade-offs but it is not without its shortcomings.

3.5.1 Imperfections with CAP classifications

Several database researchers have argued that the CAP theorem is not a sufficient way to characterize all of the engineering compromises behind building scalable, distributed systems. Some of the issues raised about the CAP theorem are presented now.

The first is the observation of an asymmetry between the C and A in CAP. Consider CP systems, these systems provide consistency and partition-tolerance and sacrifice availability in the event of network partitions. AP Systems on the other hand, provide availability and partition-tolerance at the expense of consistency. From these definitions we can see that systems which sacrifice availability (i.e. CP) do so only in the presence of network partitions, whereas systems which sacrifice consistency do so all the time. Thus, there is an asymmetry between C and A in CAP [12].

Additionally, it has been argued that there is little difference between CP and CA systems in practice. For instance, CP systems by definition give up availability when there is a network partition. However, denoting a system as CA implies that it is not tolerant to network partitions. One has to ask what “not tolerant to network partitions” means in this context. In practice, it means that in the event of a network partition the CA system will lose availability. Therefore, we can see that there is practically no difference between CA and CP systems in many cases.

However, because partition tolerance is a must in virtually all web-scale production systems (because hardware and networks fail all the time), realistically partition-tolerance is always required. As a result, systems must choose whether they require strong consistency or high availability. Therefore, in this paper systems were either classified as CP or AP.

4. Conclusions

This paper has described the CAP theorem and its implications on distributed systems. Additionally, it has presented an analysis of the CAP trade-offs made in the designs of some popular NoSQL storage systems. The CAP theorem is a useful tool for assessing and classifying the relative strengths and weaknesses of NoSQL storage systems, and distributed systems in general. Furthermore, it can provide a useful way of thinking about how individual sub systems will fit together when attempting to design an application which makes use of distributed components.

References

- [1] Seth Gilbert and Nancy Lynch, "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services," 2002.
- [2] Eric A. Brewer, "Towards Robust Distributed Systems," in *PODC Keynote*, 2000.
- [3] Curt Monash. (2010, October) InformationWeek. [Online].
http://www.informationweek.com/news/software/info_management/227701021?pgno=1
- [4] Fay Chang et al., "Bigtable: A Distributed Storage System for Structured Data," in *OSDI*, 2006, pp. 1-14.
- [5] Jim R. Wilson. (2008, May) Jimbojw. [Online].
http://jimbojw.com/wiki/index.php?title=Understanding_Hbase_and_BigTable
- [6] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung, "The Google File System," in *SOSP*, Bolton Landing, New York, USA, 2003.
- [7] Ryan Rawson, "HBase," in *NoSQL debrief*, San Francisco, 2009.
- [8] Hypertable. [Online]. <http://code.google.com/p/hypertable/wiki/ArchitecturalOverview>
- [9] Giuseppe DeCandia et al., "Dynamo: Amazon's Highly Available Key-value Store," in *SOSP*, Stevenson, Washington, USA., 2007, pp. 205-220.
- [10] Avinash Lakshman and Prashant Malik, "Cassandra - A Decentralized Structured Storage System," 2009.
- [11] Cassandra API. [Online]. <http://wiki.apache.org/cassandra/API>
- [12] Daniel Abadi. (2010, April) DBMS Musings. [Online].
<http://dbmsmusings.blogspot.com/2010/04/problems-with-cap-and-yahoos-little.html>