Brewer's CAP Theorem

Report to Brewer's original presentation of his CAP Theorem at the Symposium on Principles of Distributed Computing (PODC) 2000

Written by Salomé Simon

Table of Contents

Introduction

At the Symposium on Principles of Distributed Computing in the year 2000, Eric Brewer held a keynote talk about his experience with the recent changes in the development of distributed databases. (Brewer, Towards Robust Distributed System, 2000)

In the years before his talk, the size of data grew immensely, making it necessary to find more scalable solutions than the so far existing ACID-databases. As a result new principles were developed, summed up under the BASE-paradigm (basically available, soft-state, eventual consistency).

Brewer analyzed the consequences of this paradigm change and its implications, resulting in the CAP-Theorem which he presented in his talk – at this point more a personal intuition than an actual proven fact. However, the theorem had such a huge impact that many researchers picked up the subject, and two years later the theorem had been proven formally.

Over the years, the CAP theorem and has been constantly developed and slight adjustments have been made, most prominently by Brewer himself who amended in a later paper that some of the conclusions, while not wrong, could be misleading (Brewer, CAP twelve years later: How the "rules" have changed, 2012). However, the CAP-theorem still is one of the most important findings for distributed databases.

The CAP-Theorem

(Brewer, Towards Robust Distributed System, 2000)

A distributed database has three very desirable properties:

1. Tolerance towards Network Partition
2. Consistency
3. Availability

The CAP theorem states: **You can have at most two of these properties for any shared-data system**

Theoretically there are three options:

1. **Forfeit Partition Tolerance**
   The system does not have a defined behavior in case of a network partition. Brewer names 2-Phase-Commit as a trait of this option, although 2PC supports temporarily partitions (node crashes, lost messages) by waiting until all messages are received.
2. **Forfeit Consistency**
   In case of partition data can still be used, but since the nodes cannot communicate with each other there is no guarantee that the data is consistent. It implies optimistic locking and inconsistency resolving protocols.
3. **Forfeit Availability**
   Data can only be used if its consistency is guaranteed. This implies pessimistic locking, since we need to lock any updated object until the update has been propagated to all nodes. In case of a network partition it might take quite long until the database is in a consistent state again, thus we cannot guarantee high availability anymore.

The option of forfeiting Partition Tolerance is not feasible in realistic environments, since we will always have network partitions. Thus it follows that we need to decide between Availability and Consistency, which can be represented by ACID (Consistency) and BASE (Availability).

However, Brewer already recognized that the decision is not binary. The whole spectrum in between is useful; mixing different levels of Availability and Consistency usually yields a better result.

Comments on the CAP Theorem

While the theorem is flawless in its correctness, the formulation can be misleading about the implications:

1. The theorem presents the three properties as equal. But while Consistency and Availability can be measured in a spectrum, Partition Tolerance is rather binary. One can vary the definition of Partition Tolerance, but in the end one can only say the system supports Partition Tolerance or it does not.
2. If we vary the definition of Partition Tolerance it starts to merge with the Availability property. A temporary Partition Tolerance might as well be called a temporary Unavailability.
3. Partition Tolerance can only be forfeited in a hypothetical environment where no partition can happen. But any real system that would forfeit partition tolerance would not be working correctly and thus the option Availability and Consistency should not be considered.

In my opinion the theorem would have made its implications clearer, if it would mention Partition Tolerance as a given property, and say that under these conditions only Consistency or Availability can be guaranteed. Better even it should reflect the spectrum of possibilities, i.e. it should speak of a Consistency-Availability Tradeoff rather than of a choice between the two.

In a later paper (Brewer, CAP twelve years later: How the "rules" have changed, 2012), Brewer suggested another improvement which portraits the factor of Partition Tolerance clearer: This tradeoff between Consistency and Availability only has to be considered when the network is partitioned. At any time where the network is not partitioned, we can have both Consistency and Availability. One can interpret this fact that the system should forfeit Partition Tolerance as long as there is no partition, and as soon as a network partition occurs it needs to switch its strategy and choose a tradeoff between Consistency and Availability.

Formal Proof

In 2002 Gilbert and Lynch provided a formal proof (Gilbert & Lynch, 2002) of the cap theorem for the following three network types:

1. Asynchronous network with message loss
2. Asynchronous network without message loss
3. Partially synchronous network with local clocks

We will however only discuss the proof for asynchronous networks with message loss.

First Gilbert and Lynch defined the three properties:

1. **Consistency** (atomic data objects)
   A total order must exist on all operations such that each operation looks as if it were completed at a single instance. For distributed shared memory this means (among other things) that all read operations that occur after a write operation completes must return the value of this (or a later) write operation.
2. **Availability**
   Every request received by a non-failing node must result in a response. This means, any algorithm used by the service must eventually terminate.
3. **Partition Tolerance**
   The network is allowed to lose arbitrarily many messages sent from one node to another.

With this definition, the theorem was proven by contradiction:

Assume all three criteria (atomicity, availability and partition tolerance) are fulfilled. Since any network with at least two nodes can be divided into two disjoint, non-empty sets $\{G_1, G_2\}$, we define our network as such. An atomic object o has the initial value v_0. We define α_1 as part of an execution consisting of a single write on the atomic object to a value $v_1 \neq v_0$ in G_1 . Assume α_1 is the only client request during that time. Further, assume that no messages from G_1 are received in G_2, and vice versa. Because of the availability requirement we know that α_1 will complete, meaning that the object o now has value v_1 in G_1.

Similarly α_2 is part of an execution consisting of a single read of o in G_2. During α_2 again no messages from G_2 are received in G_1 and vice versa. Due to the availability requirement we know that α_2 will complete.

If we start an execution consisting of α_1 and α_2, G_2 only sees α_2 (since it does not receive any messages or requests concerning α_1. Therefore the read request from α_2 still must return the value v_0. But since the read request starts only after the write request ended, the atomicity requirement is violated, which proves that we cannot guarantee all three requirements at the same time. *q.e.d.*

Examples for the spectrum of the C-A Tradeoff

PNUTS from Yahoo

(Cooper & al, 2008)

PNUTS is a Data Serving Platform which forfeits serializability for transactions in favor of high availability. They argue that in most cases, serializable transactions are not necessary, thus they are not worth the impracticability and the loss of availability, on which Yahoo's web applications depend.

But they also think that the pure BASE-driven approach of eventual consistency is too weak a guarantee. As example they describe a photo share application on which you can upload pictures and choose who has permission to see your photos. Now imagine a user wants to execute two updates in this exact order: exclude his mother from the users eligible to see his pictures, and then post photos

of the recent spring-break. Obviously this user would be very unhappy about eventual consistency, if during the time in which the database was not consistent yet the mother logged into her account and saw the spring-break pictures.

As a result from the considerations above PNUTS provides a consistency model which is in the middle of the C-A-tradeoff: "**per-record timeline consistency:** all replicas of a given record apply all updates to the record in the same order". This means, the database will not immediately be consistent, but it is guaranteed that all updates made on an object are done in the same order they occurred in the timeline for all replicas of the object.

Dynamo

(DeCandia & al, 2007)

Dynamo is a completely decentralized Key-value Store developed by Amazon which strives for high availability. It incorporates the "eventual consistency" principle from BASE: a decentralized replica synchronization protocol maintains consistency during update with a quorum-like approach and object versioning. Through gossip failures can be detected.

Discussion

If we imagine the tradeoff between Consistency and Availability as a scale with one extreme meaning sacrificing all consistency for availability and the other extreme meaning sacrificing all availability for consistency, there is no accurate measure to tell us where exactly on that scale a certain database implementation is. We can however compare two databases with each other and say which one is nearer at the consistency or availability extreme. It is also important to mention that no implementation of either extreme is feasible; e.g. while BASE aims for availability it is still eventually consistent, so it is not on the availability extreme.

The decision which tradeoff is the best for a product has to be considered carefully. There is no right answer. We need to weight carefully how long a user is willing to wait for an answer of a system and how tolerant he is of inconsistencies. Of course, money also plays a considerable role for enterprise applications, and it has shown that with this constraint, systems tend more towards availability: a user does not want to wait too long on a web application; he rather reads temporarily inconsistent data, since for many applications the data is not so critical that it must be consistent.

We also need to consider that different transactions inside the same program maybe have different consistency or availability requirements: while an Amazon user doesn't mind too much if the article he put into the shopping cart is not available anymore when he proceeds to the checkout half an hour later, he will not be happy if he receives an email after the successfully buy of the product that it is not available and his order has to be cancelled (after all, he received a confirmation that his order has been placed successfully). On the other hand he would mind if he needs to wait 2 minutes every time he puts something in the shopping cart, while waiting 2 minutes for the final checkout should not be too tragic.

References

Brewer, E. (2012, February). CAP twelve years later: How the "rules" have changed. *Computer, vol. 45, no. 2* , pp. 23-29.

Brewer, E. (2000). Towards Robust Distributed System. *Symposium on Principles of Distributed Computing (PODC).*

Browne, J. (2009, January 11). *julianbrowne.com.* Retrieved January 04, 2013, from http://www.julianbrowne.com/article/viewer/brewers-cap-theorem

Cooper, B., & al. (2008). PNUTS: Yahoo's hosted data serving platform. *VLDB* .

DeCandia, G., & al. (2007). Dynamo: Amazon's Higly Available Key-value Store. *ACM Press New York* , pp. 205-220.

Gilbert, S., & Lynch, N. (2002, June). Brewers Conjunction and the Feasability of Consistent, Available, Partition-Tolerant Web Services. *ACM SIGACT News* , p. 33(2).

Shim, S. S. (2012, February). Guest Editor's Introduction: The CAP Theorem's Growing Impact. *Computer, vol. 45, no. 2* , pp. 21-22.