CAP Theorem between Claims and Misunderstandings: What is to be Sacrificed?

Balla Wade Diack[1], Samba Ndiaye[1] and Yahya Slimani[2]

1 Department of Mathematics and Informatics
Cheikh Anta Diop University of Dakar
2 Department of Informatics, El Manar University of Tunis
balla.diack@ucad.edu.sn, samba.ndiaye@ucad.edu.sn
yayha.slimani@fst.rnu.tn

Abstract

Modern large distributed Systems at wide scale have adopted new types of databases that are not subject to the features which ensure strong Consistency. In this paper, we discuss the CAP theorem, its evolution and its influence on these systems. After, we talk about the misunderstandings and problems aroused by this theorem. Finally, we give the updates on CAP brought by some researchers.

Keywords: *CAP; PACELC; liveness; safety; replication; latency*

1. Introduction

Commercial relational database management systems from vendors such as Oracle, IBM, Sybase and Microsoft have been the default home for computational data. However, with the phenomenal growth of web-generated data, this conventional way of storing data has encountered a formidable challenge. Because the traditional way of handling petabytes of data with a relational database in the back-end does not scale well, managing this phenomenon referred to as the Bigdata challenge has become problematic. [6].

To achieve these goals, many people are convinced that they must harm - temporarily - the consistency of the Data on a single-copy to ensure high Availability of Data. They are all based on the CAP theorem to justify their conviction. What is it about? In this work, we first talk about the CAP theorem and his proof. After we expose the criticisms on CAP, the misunderstandings and also some claims not always verified. After we point on Brewers update on his theorem and various opinions and thoughts about CAP contributed by several researchers.

2. The CAP Theorem

In 1997, Fox and al. released a paper in which they show the need to reconsider the ACID constraints (Atomicity, Consistency, Isolation, and Durability) for building cluster-based scalable network services that encapsulates the following requirements: incremental scalability and overflow growth provisioning 24x7 availability through fault masking, and cost-effectiveness. Thus, they suggested BASE properties: Basically Available, Soft State and Eventual Consistency. BASE semantics allow us to handle partial failure in clusters with less complexity and cost. For example, where ACID requires durable and consistent state across partial failures, BASE semantics often allows us to avoid communication and disk activity or to postpone it until a more convenient time. BASE semantics greatly simplify the implementation of fault tolerance and availability and permit performance optimizations within our framework that would be precluded by ACID [8].

- Basically Available: Replication premise to reduce the likelihood of data unavailability and sharding, or partitioning allows the data among many different storage servers, to make any remaining failures partial. The result is a system that is always available, even if subsets of the data become unavailable for short periods of time.
- Soft state: While ACID systems assume that data consistency is a hard requirement, here systems allow data to be inconsistent and relegate designing around such inconsistencies to application developers.
- Eventually consistent: Although applications must deal with instantaneous consistency, these systems ensure that at some future point in time the data assumes a consistent state. In contrast to ACID systems that enforce consistency at transaction commit, consistency is guaranteed only at some undefined future time.

The strong CAP Principle (strong Consistency, high Availability and Partition resilience) first appeared in a paper written by researchers of Berkeley University that established cluster-based wide area systems. The CAP formulation makes explicit the trade-offs in designing distributed infrastructure applications.

- CA without P: Databases that provide distributed transactional semantics can only do so in the absence of a network partition separating server peers.
- CP without A: In the event of a partition, further transactions to an ACID database may be blocked until the partition heals, to avoid the risk of introducing merge conflicts (and thus inconsistency).
- AP without C: HTTP Web caching provides client/server partition resilience by replicating documents, but a client-server partition prevents verification of the freshness of an expired replica [9].

2.1. Yield and Harvest [2, 9]

There's two metrics for improving availability: yield and harvest. Yield is the probability of completing a request. It is typically measured in 'nines' (*i.e.* 'four nines availability' means a completion probability of 0.9999). Harvest measures the fraction of the data reflected in the response, *i.e.* the completeness of the answer to the query. In the presence of faults there is a tradeoff between providing no answer (reducing yield) and providing an imperfect answer (maintaining yield, but reducing harvest). Some applications do not tolerate harvest degradation because any deviation from the single well defined correct behavior renders the result useless. Other applications tolerate graceful degradation of harvest [2, 9].

2.1.1. Yield: The traditional metric for availability is uptime, which is the fraction of time a site is handling traffic, uptime is typically measured in nines.

$$uptime = (MTBF - MTTR)/MTBF \qquad [2]$$

MTBF: meantime-between-failure and MTTR: meantime-to repair

$$yield = queries\ completed/queries\ offered \qquad [2]$$

Numerically, this is typically very close to uptime, but it is more useful in practice because it directly maps to user experience and because it correctly reflects that not all seconds have equal value. Being down for a second when there are no queries has no impact on users or yield, but reduces uptime. Similarly, being down for one second at peak and off-peak times generates the same uptime, but vastly different yields because there might be an order-of-magnitude difference in load between the peak second and the minimum-load second. Thus we focus on yield rather than uptime.

2.1.2. Harvest: Query completeness can be measured: how much of the database is reflected in the answer. Following fraction is defined as the harvest of the query:

harvest = data available/complete data *[2]*

A perfect system would have 100 percent yield and 100 percent harvest. That is, every query would complete and would reflect the entire database. The key insight is that we can influence whether faults impact yield, harvest, or both.

Replicated systems tend to map faults to reduced capacity (and to yield at high utilizations), while partitioned systems tend to map faults to reduced harvest, as parts of the database temporarily disappear, but the capacity in queries per second remains the same [2].

2.2. Brewer's Conjecture

2.2.1. PODC keynote: One year later; when presenting a keynote at PODC Symposium, E. Brewer, one of researchers mentioned above, made the following conjecture: "*It is impossible for a web service to provide the three following guarantees : Consistency, Availability and Partition-tolerance*".[1] Two years later, Gilbert and Lynch gave a more formal definition to Brewer's Conjecture in [4].

Consistency: With atomic or linearizable consistency there must exist a total order on all operations such that each operation looks as if it were completed at a single instant. This is equivalent to requiring requests of the distributed shared memory to act as if they were executing on a single node, responding to operations one at a time.

Availability: Every request received by a non-failing node in the system must result in a response. That is, any algorithm used by the service must eventually terminate. When qualified by the need for partition tolerance, this can be seen as a strong definition of availability: even when severe network failures occur, every request must terminate.

Partition tolerance: The network will be allowed to lose arbitrarily many messages sent from one node to another. No set of failures less than total network failure is allowed to cause the system to respond incorrectly [4].

2.2.2. Proof: CAP was proved four years after his enunciation by Gilbert and Lynch. Their proof was illustrated by J. Browne as in Figures 1 to 3.

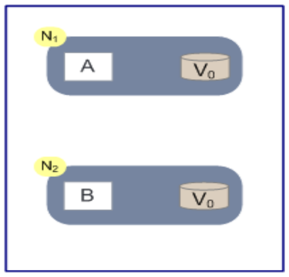

Figure 1. Two Nodes Sharing a Piece of Data V [24]

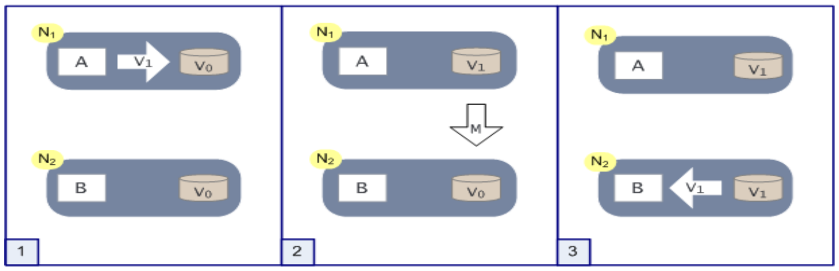

Figure 2. Consistency and Availability in System in which there's No Partitions [24]

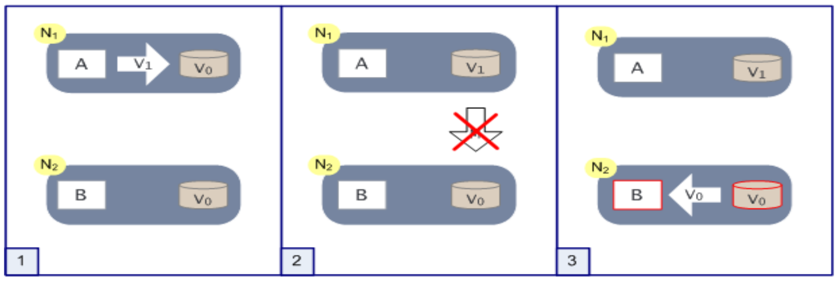

Figure 3. Lack of Consistency in System in which there's Partitions [24]

The diagram above shows two nodes in a network, N_1 and N_2. They both share a piece of data *V*, which has a value V_0. Running on N_1 is an algorithm called A which we can consider to be safe, bug free, predictable and reliable. Running on N_2 is a similar algorithm called B. A writes new values of V and B reads values of V (Figure 1).

Whether there's no partition: first A writes a new value of V, which we'll call V1. (2) Then a message (M) is passed from N1 to N2 which updates the copy of V there. (3) Now any read by B of V will return V1 (Figure 2).

If the network partitions (that is messages from N_1 to N_2 are not delivered) then N_2 contains an inconsistent value of V when step (3) occurs (Figure 3).

Scale this is up to even a few hundred transactions and it becomes a major issue. If M is an asynchronous message then N_1 has no way of knowing whether N_2 gets the message. Even with guaranteed delivery of M, N_1 has no way of knowing if a message is delayed by a partition event or something failing in N_2. Making M synchronous doesn't help because that treats the write by A on N_1 and the update event from N_1 to N_2 as an atomic operation, which gives us the same latency issues we have already talked about (or worse). Gilbert and Lynch also prove that even in a partially-synchronous model (with ordered clocks on each node) atomicity cannot be guaranteed.

So CAP tells us that if we want A and B to be highly available and we want our nodes N_1 to N_k (where k could be hundreds or even thousands) to remain tolerant of network partitions (lost messages, undeliverable messages, hardware outages, process failures) then sometimes we are going to get cases where some nodes think that V is V_0 and other nodes will think that V is V_1 [4, 24].

3. Problems, Misleadings and Claims on the CAP Theorem

The CAP theorem was not subject of discussion for over a decade, but since four years, he started arouse controversy. This is due to the fact that the NoSQL Community has used it as justification for giving up consistency. In early April 2010, some aware researchers drew attention to this simplistic conception of the CAP theorem. Thus, Abadi D., Stonebreaker M. and Hale C. have reported confusion about the CAP theorem [17, 18, 19, 20]. And they were preceded by Julian Browne in 2009, in a blog post; he was the first to report what is behind all the fuss about CAP [24].

3.1. Errors on the Conception of Eventual Consistency

The tradeoff in a Distributed System on the web depends on what caused a break in the network. In fact, a typical hardware model is a collection of local processing and storage nodes assembled into a cluster using LAN networking. The clusters, in turn, are wired together using WAN networking. CAP theorem does not apply at all in the following cases:

- The application performed one or more incorrect updates. The database must be backed up to a point before the offending transaction(s), and subsequent activity redone.
- The DBMS crashed at a processing node. Executing the same transaction on a processing node with a replica will cause the backup to crash (Bohr bugs).
- There’s a disaster. The local cluster is wiped out by a flood, earthquake, etc. The cluster no longer exists.

In the first two scenarios, availability is impossible to achieve. Also, replica consistency is meaningless; the current DBMS state is simply wrong. For the latter scenario error will be recoverable if a local transaction is only committed after the assurance that the transaction has been received by another WAN-connected cluster. Hence, eventual consistency cannot be guaranteed, because a transaction may be completely lost if a disaster occurs at a local cluster before the transaction has been successfully forwarded elsewhere.

The fall of a node within a cluster (OS failure, mechanical failure local network outage - very rare) is easily manageable now with failover.

A network failure in the WAN connecting clusters together. The WAN failed and clusters can no longer all communicate with each other. There is enough redundancy engineered into today’s WANs that a partition is quite rare. Moreover, the most likely WAN failure is to separate a small portion of the network from the majority. In this case, the majority can continue with straightforward algorithms, and only the small portion must block. Hence, it seems unwise to give up consistency all the time in exchange for availability of a small subset of the nodes in a fairly rare scenario. In summary, one should not throw out the C so quickly, since there are real error scenarios where CAP does not apply and it seems like a bad tradeoff in many of the other situations [18].

3.2. PACELC instead of CAP [10]

D. Abadi drew attention to the error associated with trilogy implied by the CAP theorem. CAP implies that there are three types of distributed systems: CA (consistent and available, but not tolerant of partitions), CP (consistent and tolerant of network partitions, but not available), and AP (available and tolerant of network partitions, but not consistent).

The definition of CP looks incoherent: "*consistent and tolerant of network partitions, but not available*"; the way that this is written makes it look like such as system is never available - a clearly useless system. Obviously this is not really the case; rather, availability is only sacrificed when there is a network partition. In practice, this means that the roles of the A and C in CAP are asymmetric. Systems that sacrifice consistency (AP systems) tend to do so all the time, not just when there is a network partition. So, the asymmetry of A and C make problem.

The second problem in CAP is that, there is no practical difference between CA systems and CP systems. CP systems give up availability only when there is a network partition whether CA systems are "not tolerant of network partitions" in other words they lose availability if there is a partition. Hence CP and CA are essentially identical. So in reality, there are only two types of systems: CP/CA and AP. Having three letters in CAP and saying you can pick any two does nothing but confuse this point.

Even more serious than the above, the main problem with CAP is that it focuses everyone on a consistency/availability tradeoff, resulting in a perception that the reason why NoSQL systems give up consistency is to get availability. But this is far from the case (see Yahoo's NoSQL system called PNUTS) [16, 20].

PNUTS give up both consistency and availability while CAP says you only have to give up just one! It relaxes consistency by only guaranteeing "timeline consistency" where replicas may not be consistent with each other but updates are guaranteed to be applied in the same order at all replicas. However, they also give up availability - if the master replica for a particular data item is unreachable, that item becomes unavailable for updates (when focusing on the default system described in the original PNUTS paper).

The reason is that CAP is missing latency (L). Yahoo's PNUTS gives up consistency not for the goal of improving availability, but it is to lower latency. Keeping replicas consistent over a wide area network requires at least one message to be sent over the WAN in the critical path to perform the write. Unfortunately, a message over a WAN significantly increases the latency of a transaction (on the order of hundreds of milliseconds), a cost too large for many Web applications that businesses like Amazon and Yahoo need to implement. Consequently, in order to reduce latency, replication must be performed asynchronously. This reduces consistency (by definition). In Yahoo's case, their method of reducing consistency (timeline consistency) enables an application developer to rely on some guarantees when reasoning about how this consistency is reduced, but consistency is nonetheless reduced.

So, CAP should really be PACELC. Thus, if there is a partition (P) how does the system tradeoff between availability and consistency (A and C); else (E) when the system is running as normal in the absence of partitions, how does the system tradeoff between latency (L) and consistency (C)?

Systems that tend to give up consistency for availability when there is a partition give up also consistency for latency when there is no partition. This is the source of the asymmetry of the C and A in CAP and this confusion is not present in PACELC. For example, Amazon's Dynamo (and related systems like Cassandra and SimpleDB) are

PA/EL in PACELC - upon a partition, they give up consistency for availability; and under normal operation they give up consistency for lower latency. Giving up C in both parts of PACELC makes the design simpler - once the application is configured to be able to handle inconsistencies, it makes sense to give up consistency for both availability and lower latency. Fully ACID systems (VoltDB/H-Store and Megastore) are PC/EC in PACELC. They refuse to give up consistency, and will pay the availability and latency costs to achieve it.

However, there are some interesting counterexamples where the C's of PACELC are not correlated. One such example is PNUTS, which is PC/EL in PACELC. In normal operation they give up consistency for latency; however, upon a partition they don't give up any additional consistency (rather they give up availability) [13, 16].

In conclusion, rewriting CAP as PACELC removes some confusing asymmetry in CAP, and comes closer to explaining the design of NoSQL systems [10, 16, 20].

3.3. You Can't Sacrifice Partition Tolerance [19]

Developers of systems which claim to be CA do not understand the CAP theorem and its implications. Partition tolerance seems to be the part that most people misunderstand.

For a distributed system to not require partition-tolerance it would have to run on a network which is guaranteed to never drop messages (or even deliver them late) and whose nodes are guaranteed to never die (what doesn't exist).

Partitions (failures) do happen, and the chance that any one of your nodes will fail jumps exponentially as the number of nodes increases:

$$P(\textit{any fail.}) = 1 - P(\textit{individual node not failing})^{\textit{number of nodes}}$$

If a single node has a 99.9% chance of not failing in a particular time period, a cluster of 100 has a 90.5% chance not failing. In other words, you've got around a 10% chance that something will go wrong. Therefore, the question you should be asking yourself is: "In the event of failures, which will this system sacrifice, consistency or availability? "

3.3.1. Choosing Consistency over Availability: Here system will preserve the guarantees of its atomic reads and writes by refusing to respond to some requests. It may decide to shut down entirely (like the clients of a single-node data store), refuse writes (like Two-Phase Commit), or only respond to reads and writes for pieces of data whose "master" node is inside the partition component (like Membase). There are plenty of things (atomic counters, for one) which are made much easier (or even possible) by strongly consistent systems. They are a perfectly valid type of tool for satisfying a particular set of business requirements.

3.3.2. Choosing Availability over Consistency: The system will respond to all requests, potentially returning stale reads and accepting conflicting writes. These inconsistencies are often resolved via causal ordering mechanisms like vector clocks and application-specific conflict resolution procedures. There are plenty of data models which are amenable to conflict resolution and for which stale reads are acceptable and for which unavailability results in massive bottom-line losses. (Amazon's shopping cart system is the canonical example of a Dynamo model).

3.3.3. Proof that you cannot have Consistency and Availability simultaneously: Given a distributed system using three nodes: A, B, and C, and which claims to be both consistent and available in the face of network partitions; a misfortune partition the system into two components: {A,B} and {C}. In this state, a write request arrives at node C to update the single piece of data. That node only has two options: (1) Accept the write, knowing that neither A nor B will know about this new data until the partition heals. (2) Refuse the write, knowing that the client might not be able to contact A or B until the partition heals. You cannot choose both. To claim to do so is claiming either that the system operates on a single node (and is therefore not distributed) or that an update applied to a node in one component of a network partition will also be applied to another node in a different partition component magically.

3.3.4. Conclusion: Unavoidably, your system will experience enough faults that it will have to make a choice between reducing yield (*i.e.*, stop answering requests) and reducing harvest (*i.e.*, giving answers based on incomplete data). This decision should be based on business requirements. Instead of CAP, you should think about your availability in terms of yield (percent of requests answered successfully) and harvest (percent of required data actually included in the responses) and which of these two your system will sacrifice when failures happen [19].

4. The CAP Theorem Fifty Years Later

IEEE Computer magazine came out with an issue largely devoted to a 12-year retrospective (or exactly 15 years) of the CAP theorem and contains several articles (about six) from distributed systems researchers that contribute about CAP [20]. The most important item of this issue is that of the author, after his theorem has begun to raise the controversy.

4.1. How Rules have changed [3]

Designers and researchers have used (and sometimes abused) the CAP theorem as a reason to explore a wide variety of novel distributed systems. Brewer noticed particularly that NoSQL movement has applied it as an argument against traditional databases and explained how the CAP theorem was misunderstood. The modern CAP goal should be to maximize combinations of consistency and availability that make sense for the specific application, according to him.

The “2 of 3” view is misleading on several fronts:

- First, because partitions are rare, there is little reason to forfeit C or A when the system is not partitioned.
- Second, the choice between C and A can occur many times within the same system at very fine granularity; not only can subsystems make different choices, but the choice can change according to the operation or even the specific data or user involved.
- Third, all three properties are more continuous than binary. Availability is obviously continuous from 0 to 100 percent, but there are also many levels of consistency, and even partitions have nuances, including disagreement within the system about whether a partition exists.

Because partitions are rare, CAP should allow perfect C and A most of the time, but when partitions are present or perceived, the strategy which have three steps should be used: detect partitions, enter an explicit partition mode that can limit some operations, and initiate a recovery process to restore consistency and compensate for mistakes made during a partition.

It's to manage partitions very explicitly, including not only detection, but also a specific recovery process and a plan for all of the invariants that might be violated during a partition. This management approach has three steps:

- detect the start of a partition,
- enter an explicit partition mode that may limit some operations,
- initiate partition recovery when communication is restored which aims to restore consistency and compensate for mistakes the program made while the system was partitioned.

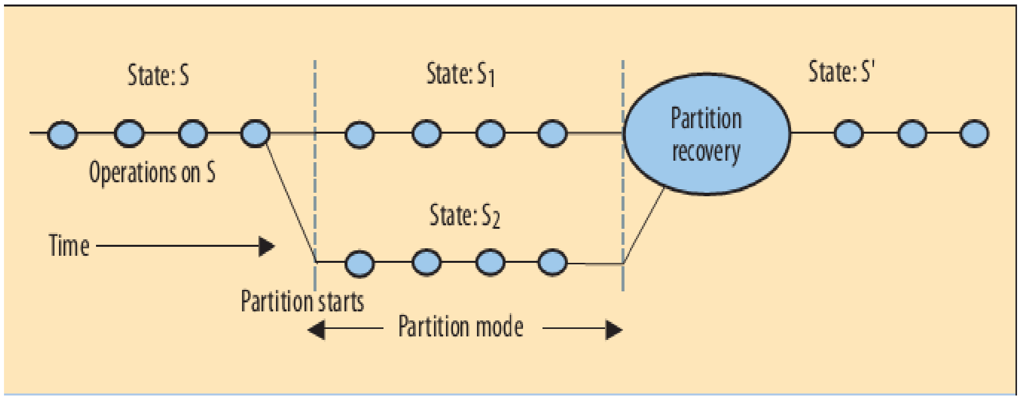

Figure 4. A Partition's Evolution [3]

Normal operation is a sequence of atomic operations, thus partitions always start between operations. Once the system times out, it detects a partition, and the detecting side enters partition mode. Thus, the other side communicates as needed and either this side responds correctly or no communication was required; either way, operations remain consistent. However, because the detecting side could have inconsistent operations, it must enter partition mode. Once the system enters partition mode, two strategies are possible. The first is to limit some operations, thereby reducing availability. The second is to record extra information about the operations that will be helpful during partition recovery.

We should decide what operations we are going to limit while maintaining the invariants of the system. The best way to track the history of operations on both sides is to use version vectors, which capture the causal dependencies among operations. The vector's elements are a pair (node, logical time), with one entry for every node that has updated the object and the time of its last update. Given the version vector history of both sides, the system can easily tell which operations are already in a known order and which executed concurrently. [21] proved that this kind of causal consistency is the best possible outcome in general if the designer chooses to focus on availability.

The designer must solve two hard problems during recovery: (1) the state on both sides must become consistent, and (2) there must be compensation for the mistakes made during partition mode. Although the general problem of conflict resolution is not solvable, in practice, designers can choose to constrain the use of certain operations during partitioning so that the system can automatically merge state during recovery. Delaying risky operations is one relatively easy implementation of this strategy. Using commutative operations is the closest approach to a general framework for automatic state convergence, Marc Shapiro and colleagues at INRIA has improved the use of commutative operations for state convergence by developing commutative replicated data types (CRDTs), a class of data structures that provably converge after a partition, and describe how to use these structures to: (1) ensure that all operations during a partition are com-mutative, or (2) represent values on a lattice and ensure that all operations during a partition are monotonically increasing with respect to that lattice [22, 23].

Conclusion: System designers should not blindly sacrifice consistency or availability when partitions exist. They can optimize both properties through careful management of invariants during partitions. As newer techniques, such as version vectors and CRDTs, move into frameworks that simplify their use, this kind of optimization should become more wide-spread. However, unlike ACID transactions, this approach requires more thoughtful deployment relative to past strategies, and the best solutions will depend heavily on details about the service's invariants and operations [3, 7].

4.2. Other New Issues

S. Gilbert and N. Lynch reviewed the Cap theorem and stated it as follows: "*In a network subject to communication failures, it is impossible for any web service to implement an atomic read/write shared memory that guarantees a response to every request*". They conceptualized CAP tradeoff between safety (nothing bad ever happens) and liveness eventually (something good happens) *i.e.* it is impossible of guaranteeing the safety and liveness consensus in an unreliable distributed system. They explained how practically systems tradeoff between Availability and Consistency maximizing the goal for each of the two features. They concluded by discussing on CAP influence on Future Systems, and explained that we need new theoretical insights to address new challenges as Scalability, Tolerating attacks, Mobile wireless Network [5].

In "CAP and Cloud Data Management," R. Ramakrishnan points out that coordinating all updates through a master may have obvious performance and availability implications. PNUTS alleviates these issues by automatically migrating the master to be close to the writers. As this makes the practical impact on performance and availability insignificant for Yahoo's applications because of localized user access patterns.

Finally, in "Overcoming CAP with Consistent Soft-State Replication," Kenneth P. Birman and his coauthors advocate for even stronger consistency inside the datacenter, where partitions are rare. They show that in this setting, it is possible to achieve low latency and scalability without sacrificing consistency [6, 14].

5. Conclusion

The Brewer conjecture better known under the designation of the CAP theorem has experienced a resurgence of interest with the affluence of new types of databases that use it to justify the relaxation of the strong consistency. Awakened mind notified showed that release consistency in a widely distributed environment does not necessarily guarantee a high availability. In this paper we shown that there's not a binary about CAP and that people want all these properties together, but it is the reality that prevents them from getting it.

The networked world has changed significantly in the last ten years, creating new challenges for system designers, and new areas in which these same inherent tradeoffs can be explored. So, new techniques for coping with the problem in real-world systems are needed.

• Scalability: A system is scalable if it can grow efficiently, using new resources efficiently to handle more load. There appear to be inherent trade-offs between scalability and consistency. For example, in order to efficiently use new resources, there must be coordination among those resources; the consistency required for this coordination appears subject to the CAP theorem trade-offs. Studying this question may help to explain why even within a datacenter, where there are rarely partitions, it seems difficult to efficiently scale strongly consistent protocols?

• Tolerating attacks: The CAP Theorem focuses on network partitions: sometimes, some servers cannot communicate reliably. Increasingly, however, we are seeing more severe attacks on networks. For example, denial-of-service attacks are becoming a near continuous threat to everyday network operations. A denial-of-service attack, however, cannot simply be modeled as a network partition. Similarly, we are seeing problems with malicious users hacking servers and otherwise disrupting major internet services. Tolerating these more problematic forms of disruption requires a somewhat different understanding of the fundamental consistency/availability trade-offs.

• Mobile wireless networks: The CAP Theorem initially focused on wide-area internet services. Today, however, a significant percentage of internet traffic is initiated by mobile devices. Many of the same trade-offs explored in the context of the CAP Theorem also hold in mobile networks – and many of the problems are even harder to resolve. Notably, wireless communication is notoriously unreliable. The key problem that motivated the CAP Theorem was the frequency of semi-stable partitions that change every few minutes. In a wireless networks, partitions are less common. However, unpredictable message loss is very common, and message latencies can vary significantly. In addition, the types of applications being deployed in wireless networks may be somewhat different. The CAP Theorem was motivated by internet search engines and e-commerce web sites. There is a new generation of wireless applications, however, that tend to focus on different priorities: geography and proximity are critical; social interactions are primary; and privacy has a somewhat more immediate meaning. For example, consider foursquare, an application in which users check-in to locations, and initiate comments and discussion based on where they are. By re-examining the CAP Theorem in the context of wireless networks, we may hope to better understand the unique trade-offs that occur in these types of scenarios [5].

Acknowledgements

We thank the Database and Datamining group members of the Doctoral School of Mathematics and Computer Science at UCAD for their contribution to our research.

References

[1] E. Brewer, “Towards Robust Distributed Systems”, Portland, Oregon, Keynote at the ACM Symposium on Principles of Distributed Computing (PODC) on **(2000)** July 19.
[2] E. Brewer, “Lessons from Giant-Scale Services”, IEEE Internet Computing, **(2001)** July-August, pp.46-55.
[3] E. Brewer, “Pushing the CAP: Strategies for Consistency and Availability”, Computer, **(2012)** February, pp. 23-29.
[4] S. Gilbert and N. Lynch, “Brewer’s Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services”, ACM SIGACT News, **(2002)** June, pp. 51-59.
[5] S. Gilbert and N. Lynch, “Perspectives on the CAP theorem”, IEEE Computer Society, vol. 45, no. 2, **(2012)** February, pp. 30-36.
[6] S. S. Y. Shim, “CAP theorem’s growing impact”, IEEE Computer Society, vol. 45, no. 2, pp. 20-21, **(2012)** February.
[7] K. Birman, Q. Huang and D. Freedman, “Overcoming the ‘D’ in CAP: Using Isis2 to Build Locally Responsive Cloud Services,” Computer, **(2011)** February, pp. 50-58.
[8] A. Fox, “Cluster-Based Scalable Network Services”, Proc. 16th ACM Symposium. Operating Systems Principles (SOSP 97), ACM, **(1997)**, pp. 78-91.

[9] A. Fox and E. A. Brewer, “Harvest, Yield and Scalable Tolerant Systems”, Proc. 7th Workshop Hot Topics in Operating Systems (HotOS99), IEEE CS, **(1999)**, pp. 174-178.
[10] D. J. Abadi, “Consistency Tradeoffs in Modern Distributed Database System Design”, IEEE Computer Society, vol. 45, no. 2, **(2012)** February, pp. 37-42.
[11] W. Vogels, “Eventually Consistent”, ACM Queue, vol. 6, no.6, **(2008)**, pp. 14-19.
[12] D. Pritchett, “BASE: An Acid Alternative,” ACM Queue, May/June 2008, pp. 48-55.
[13] R. Ramakrishnan, “CAP and Cloud Data Management”, IEEE Computer Society, vol. 45, Issue: 2 pp. 43-49, Feb. 2012.
[14] K. Birman and al., “Overcoming CAP with Consistent Soft-State Replication”, IEEE Computer Society, vol. 45, Issue: 2 pp. 50- 58, Feb. 2012.
[15] Oracle Corporation, “Oracle NoSQL Database”, Oracle Corporation World Headquarters, **(2011)**.
[16] B. F. Cooper, “PNUTS: Yahoo!’s Hosted Data Serving Plat form”, Proc. VLDB Endowment (VLDB 08), ACM, 2008, pp. 1277-1288.
[17] M. Stonebraker, “Errors in Database Systems, Eventual Consistency, and the CAP Theorem”, blog, Comm. ACM, **(2010)** April 5, http://cacm.acm.org/blogs/blog-cacm/83396-errors-in-database-systems-eventualconsistency-and-the-cap-theorem.
[18] M. Stonebraker, “Clarifications on the CAP Theorem and Data-Related Errors”, VoltDB blog, **(2010)** October 21, http://blog.voltdb.com/clarifications-cap-theorem-and-data-related-errors/.
[19] C. Hale, “You Can’t Sacrifice Partition Tolerance”, **(2010)** October 7, http://codahale.com/you-cant-sacrifice-partition-tolerance.
[20] D. Abadi, “IEEE Computer issue on the CAP Theorem”, DBMS Musings blog, **(2012)** October 29, http://dbmsmusings.blogspot.com/2012_10_01_archive.html.
[21] P. Mahajan, L. Alvisi and M. Dahlin, “Consistency, Availability, and Convergence”, tech. report UTCS TR-11-22, Univ. of Texas at Austin, **(2011)**.
[22] M. Shapiro, “Conflict-Free Replicated Data Types”, Proc. 13th Int’l Conf. Stabilization, Safety, and Security of Distributed Systems (SSS 11), ACM, **(2011)**, pp. 386-400.
[23] M. Shapiro, “Convergent and Commutative Replicated Data Types”, Bulletin of the EATCS, no. 104, **(2011)** June, pp. 67-88.
[24] J. Browne, “Brewer’s CAP theorem”, J. Browne blog, **(2009)** January 11, http://www.julianbrowne.com/article/viewer/brewers-cap-theorem.

Authors

Balla Wade DIACK7, after his Phil. M. at Cheikh Anta Diop University (UCAD) in 2010, he is preparing a PhD on Distributed Databases in the Cloud Computing. Presently he is a member of Databases and Datamining Group at Graduate School of Mathematic and informatics in UCAD.

Samba NDIAYE is an Assistant Professor at the Faculty of Science and Technology of UCAD. He’s the Coordinator of the group of Database

Yahya SLIMANI, a Professor at Faculty of Sciences of Tunis and Chief of the Department of Computer Science. He’s also a Professor Visitor at University USTO of Oran, Algeria.