CAP Twelve Years Later: How the “Rules” Have Changed
Review by Abhishek Dalvi (50320110)

In the 1960s-70s, traditional database maintenance used to occur when people were not using them(availability wasn’t important); e.g., in bank databases, batch processing, and auditing happened during nighttime. With the rise of the internet and live distributed systems, there was a shift in the focus from consistency to availability as updates were necessary at any time. Therefore, traditional databases that followed ACID transactions were frequently unavailable in such a setting. Eric Brewer found it necessary to migrate from the traditional notion of ACID by introducing the CAP theorem. C-consistency, A- availability, and P-partition tolerance. CAP theorem says that only 2 of these properties can be achieved. This paper proposes that instead of having only 2 of these, one can achieve tradeoffs of all the three properties if partitions are managed.

The paper nicely explains ACID, BASE, CAP, and clears the confusion between these concepts. It explains that while designing a system, choosing between the consistency-availability spectrum is equivalent to thinking between 2 extremes of this spectrum, i.e., ACID (consistency end) and BASE (availability end). The paper also explains that the relationship between ACID and CAP is more complex and often misunderstood by people primarily because availability affects only some ACID properties. It further presents the CAP confusion, i.e., C and A both can be achieved as partitions are very rare and that the choice between C & A can occur many times within this system. Finally, all three properties are more continuous than binary.

The paper further illustrates the connection between CAP and latency. It explains it using timeouts, where the system has to choose between availability(ignore ack) and consistency(wait till ack received, risking availability). The paper says that one has to develop a smart system design to handle such scenarios by setting appropriate thresholds for response times. It further shows how Yahoo and Facebook handle such scenarios and what properties they trade-off.

The paper continues to introduce the proposed approach to achieve all the three tradeoffs: 1)detecting partition, 2) limiting operations, or maintaining an extra log of what happens in the partitions for partition recovery and 3)partition recovery. In the first step, when a partition is detected, the detecting side or both sides go into partition mode. Second, in the partition mode, one should decide whether the partition should limit operation to maintain consistency or should keep extra logs to help during recovery mode, which involves a choice between violating and not following the invariant. Finally, when connection resumes, partition recovery takes place, which can be done by rolling back to the most recent consistent state or by compensating for the mistakes.

This being a very generalized approach, one can implement their own strategies to handle these subtasks depending on the use case; for example, in recovery, one can roll back to the most recent consistent state between partitions. The paper also shows a really easy method for partition recovery using monotonic systems. If the value of a variable goes in one direction, then it is easy to reconcile it. An excellent example of this is explained in the class: amazon's shopping cart example. If there are two versions of the shopping cart due to a partition, you just take the union, hence the customer sees 2 items in their shopping car. The paper mentions another example which was also mentioned in class about 2 nodes trying to communicate with each other when a partition occurs; where a node could wait indefinitely for an acknowledgment (give up availability) or it could continue operation without receiving the acknowledgment (give up consistency). The paper also proposes an ingenious and easy way for compensating for the mistakes like version vectors and simple strategies like "last writer wins".

Another strength of this paper is that it nicely explains concepts through examples which makes understanding the paper a lot easier. An example of git version control(any version control, I have considered git) is mentioned where it is not guaranteed that all merges will work, i.e., conflicts might occur. In this case, one has to fix conflicts by hand. This is a perfect example where availability prevents eventual consistency. The paper also uses the same example to explain one might not care about the conflicts(eg. unused variable causing a conflict), which is an example where availability and consistency(almost full consistency as a programmer won't care about a conflict caused by an unused variable) is achieved.

The paper fails to mention why people are focusing more on availability and in turn leads to giving emphasis to latency. It should have shown surveys where there is a clear correlation between latency and revenue, and why DB companies focusing on AP are successful businesses. It should have also focused on a very controversial question i.e is a timeout a partition?. Although many people believe that it is not possible to reliably detect a partition and the only thing that matters is the expiration of the timeout without thinking about the underlying reason for the cause of the timeout, I think that clear reasoning for the cause could provide a base of further study to investigate this topic. The paper also did not mention the advantages of a consistent system. If a system is designed such that it is consistent, then the system architect doesn't need to worry about any underlying invariants. Perhaps, in such a setting, one could then maybe achieve different levels of availability.

The future work on this topic is deciding on the invariants of the system. Global invariants are the ones breaking the system in the presence of partitions. In the presence of disconnected components, a system can achieve a moderate amount of consistency by only maintaining local invariants. A good example of this is are how MAC addresses of different systems are unique and never have the same addresses, even when they are manufactured by different vendors(disconnected/partition). This is because ethernet card manufacturers or vendors are given a range of MAC addresses, hence they have a local invariant, i.e, generate unique numbers in the given range.

A and P are the major focus but to achieve different levels of C, one could design a system with a significant focus on local invariants. Global invariants are the ones that most adversely affect the system. Deciding between local and global invariants and to what degree one should use local and global invariants is highly dependent on the use case. I think the further extension of this topic could be done by investigating these tradeoffs.