**An Outdated Philosophy of Distributed Systems**

Before we begin our analysis of Conflict-free replicable data types, let’s start with a discussion of distributed systems. Formally, “A distributed system is a composition of a set of processes/participants invoking methods on shared objects (registers, queues, etc.). An object implements a programming interface (API) defined by a set of methods, M, with input and output from a data domain D.” [[1]](#footnote-1) IBM’s 1979 *Note on Distributed Databases*, puts it more simply – a distributed database is a database with “multiple sites each of which stores data. These sites communicate over a slow, unreliable communication network. Such a network can lose messages, duplicate messages, and deliver messages out of order.”[[2]](#footnote-2) Fundamentally, replicating data in multiple locations across the network is done to maximize data availability. Consider the following example presented in the IBM paper, if each datastore is available with a probability , then if each piece of data exists in only one location, each piece of data is accessible with probability If, on the other hand, each piece of data is replicated times, then each piece of data is available with probability . If we assume that and then the result of replicating the data changes the probability of availability from .95 to 0.00000625. Although .95 is still a relatively high probability of uptime, in large scale distributed systems (think AWS) that handle trillions of transactions, improbable events like server downtime become almost guaranteed.[[3]](#footnote-3) As a result, systems are designed with replication of datastores to guarantee consistent availability.[[4]](#footnote-4) This solution helps systems maintain their availability in exchange for creating complexity in making sure that data is consistent across the replicas.[[5]](#footnote-5)

Ideally, the consistency model would say that when one update is made to one replica, that update is automatically reflected in real time on all other replicas. Of course, it is not possible for an update to automatically update every replica without communication between replicas. However, there is a consistency model that mimics this desired behavior, the unanimous agreement update strategy.[[6]](#footnote-6) This strategy dictates that unless every replica accepts the update, the update is rejected. Thus, with a replica availability probability of and replicas, each update is only accepted with probability Again, if and , each update is only accepted about 81% of the time.[[7]](#footnote-7) Note, that as the number of replicas grows large, the probability of a successful write operation goes to 0. So, unless the database is used dramatically more for reading than writing and data consistency is of the absolute most importance, the unanimous agreement update strategy prevents write transactions too frequently to be a suitable solution.[[8]](#footnote-8)

There are many other data consistency strategies that offer a higher probability that write requests will succeed….

The consistency models discussed above and written about by IBM in 1979, attempt to achieve distribution transparency – the idea that, to the user, the distributed system appears like it is one singular system instead of a network of databases working together.[[9]](#footnote-9) They took the philosophy that it was better to fail transactions than break the façade of distribution transparency.[[10]](#footnote-10)

**The CAP Theorem – formalizing tradeoffs**

In 2002, researchers from MIT formalized the CAP theorem, which states, “it is impossible for a web service to provide the following three guarantees:” “consistency”, “availability”, and “partition-tolerance.”[[11]](#footnote-11) Consistency is the guarantee that there exists some ordering of all operations such that it appears as if each operation occurred at one singular instant. You can think of this as making the execution in a distributed environment look as if it were on a singular node.[[12]](#footnote-12) Availability says that every request received by a non-failing node must eventually terminate with some response.[[13]](#footnote-13) A partition is the a division of the nodes in a network such that there are no successful communications between nodes in different partitions. Thus, partition-tolerance states that consistency and availability still occur even if the network is partitioned. [[14]](#footnote-14)

Let us discuss the high-level impossibility proof that distributed databases cannot have consistency, availability and partition-tolerance. We will break this proof up into two claims.

First, in a distributed system it is impossible for a read/write data object to have availability and consistency in an environment in which messages may be lost. The basis of the proof follows: assume a network contains two nodes . Create a partition of the network such that and can no longer communicate with each other. Let function write data to . Later, let read from . The value returned from and will be the same. Thus, this system is not consistent. [[15]](#footnote-15)

Second, in a distributed system it is impossible for a read/write data object to be available in all executions and consentient in all executions in which no data is lost. Let us again discuss the high-level ideas of the proof. First, note that the algorithm cannot determine if a message is lost or if its transmission through the network is facing some arbitrary delay. Thus, if the algorithm guarantees atomic consistency for all transactions in which no messages are lost, it must also guarantee atomic consistency in all executions. However, our first proof showed that a network cannot guarantee availability and atomic consistency in all fair executions. Thus, the network is unable to guarantee availability in all fair executions and atomic consistency in only fair executions with no message loss. [[16]](#footnote-16)

**Shifting Philosophies**

Let us remember that up until the mid-1990’s the standard belief was that distributed systems should aim for distribution transparency, that is, it is better to fail than break consistency. [[17]](#footnote-17)

However, as the internet grew and distributed systems became an increasingly popular and important tool to everyday life – sites like UseNet, a messaging board to exchange information on threaded topics [[18]](#footnote-18) – , the idea of systems being unavailable became increasingly less tolerable. [[19]](#footnote-19) MAYBE A GRAPH OF USE OF INTERNET AND SIZE OF DISTRIBUTED SYSTEMS Thus, the industry’s mindset began to shift from one prioritizing consistency to one prioritizing availability.

The CAP Theorem helped researchers understand the tradeoffs that could be made to maintain availability. It proved that only two of the following three properties could be achieved in a distributed system: data consistency, system availability, and tolerance to network partitions. [[20]](#footnote-20)[[21]](#footnote-21)

Importantly, in large-scale distributed systems, network partitions are inevitable [[22]](#footnote-22) and therefore, it is impossible for large-scale distributed systems to maintain data consistency and system availability according to the CAP Theorem. [[23]](#footnote-23)[[24]](#footnote-24) If we want to prioritize availability, then it must be at the expense of consistency.

**Eventual Consistency**

Because we know that partitions will occur, the CAP theorem dictates that because we have chosen to prioritize availability, we must settle for weak consistency. [[25]](#footnote-25)

Let’s informally define weak data consistency as the following: “The system does not guarantee that subsequent accesses will return the updated value. A number of conditions need to be met before the value with be returned.” [[26]](#footnote-26) Those conditions are determined by the specific implementation of weak consistency. [[27]](#footnote-27)

One such form of weak consistency is eventual consistency, which guarantees that if no new updates are made to a data object, then eventually all accesses to that data object will return its most recently updated value. [[28]](#footnote-28)

Consider the following example inspired by Hackernoon.

As I am writing this paper, I want to take precautions to make sure that even if my laptop breaks, I will not lose my paper. To do this, I have bought a backup external hard drive and am syncing my paper to Dropbox. With this hardware, I can back up my work in a few ways.

Option 1. Dropbox automatically syncs my paper to the Dropbox server every time I am connected to the internet and I manually back up my paper to my external hard drive every 20 days. If I want a friend to edit my paper in the middle of one of my twenty-day cycles, I hand them my hard drive even though it might contain a version of my paper that is not the most up to date. This allows my friend to get immediate access to my paper at the expense of having a slightly stale version. This is an eventually consistent model because I know that by the end of the next 20 day cycle, my data will once again be consistent across all three replicas.

Option 2. I use the same cadence for backing up my paper. On the twentieth day of my back-up cycle, I am editing my paper in a park and bring my hard drive with me. As I am uploading my newest version of the paper to my hard drive, I run into a friend, Jake. Jake is interested in what I reading my paper, so I share with him a link to my Dropbox paper. But because I have been making edits to my paper while in the park and not connected to wifi, I tell Jake to only access the link in an hour after I am able to return home, reconnect to wifi, and update the version of the paper stored on Dropbox. This strong consistent model allows Jake to have the most up to date version of my paper at the expense of immediate access to it. [[29]](#footnote-29)

We can summarize the above example in the following sentences. In a weak consistent model, data is easily accessed, but it may be stale. In a strong consistency model, data access may be delayed, but it will always be up to date.

Perhaps the most

**What is a CRDT?**

A conflict-free replicated data type is a data structure, able to be replicated across multiple nodes in a network such that transactions can be processed independently by nodes and shared across the network such that regardless of the order in which each node receives each transaction, each node will result in th (Gilbert and Nancy 2002)e same final state [[30]](#footnote-30)[[31]](#footnote-31)

1. Gaoang Liu and Xiuying Liu, “The Complexity of Weak Consitency,” *Zenodo*, January 29, 2018, https://doi.org/10.5281/zenodo.1161960 page 3. [↑](#footnote-ref-1)
2. Bruce Lindsay et al., “Note on Distributed Databases” (IBM Research Laboratory San Jose, California 95193: IBM, July 14, 1979), https://domino.research.ibm.com/library/cyberdig.nsf/papers/A776EC17FC2FCE73852579F100578964/$File/RJ2571.pdf. page 1 [↑](#footnote-ref-2)
3. Werner Vogels, “Eventually Consistent,” *ACM Queue* 6, no. 6 (December 4, 2008), https://queue.acm.org/detail.cfm?id=1466448. [↑](#footnote-ref-3)
4. Vogels. [↑](#footnote-ref-4)
5. Vogels. [↑](#footnote-ref-5)
6. Lindsay et al., “Note on Distributed Databases.” page 2 [↑](#footnote-ref-6)
7. Lindsay et al. page 2 [↑](#footnote-ref-7)
8. Lindsay et al. page 2 [↑](#footnote-ref-8)
9. Vogels, “Eventually Consistent.” [↑](#footnote-ref-9)
10. Vogels. [↑](#footnote-ref-10)
11. Seth Gilbert and Nancy Lynch, “Brewer’s Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services” (Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, 2002), https://users.ece.cmu.edu/~adrian/731-sp04/readings/GL-cap.pdf. page 1. [↑](#footnote-ref-11)
12. Gilbert and Lynch page 3. [↑](#footnote-ref-12)
13. Gilbert and Lynch page 3. [↑](#footnote-ref-13)
14. Gilbert and Lynch pages 3-4. [↑](#footnote-ref-14)
15. Gilbert and Lynch page 5. [↑](#footnote-ref-15)
16. Gilbert and Lynch page 5. [↑](#footnote-ref-16)
17. Lindsay et al., “Note on Distributed Databases." page 2 [↑](#footnote-ref-17)
18. “Usenet.Com,” n.d., https://www.usenet.com/what-is-usenet/. [↑](#footnote-ref-18)
19. Vogels, “Eventually Consistent.” [↑](#footnote-ref-19)
20. Vogels. [↑](#footnote-ref-20)
21. Gilbert and Lynch, “Brewer’s Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services." page 3 [↑](#footnote-ref-21)
22. Ariel Tseitlin, “The Antifragile Organization,” *ACM Queue* 56, no. 8 (August 2013), https://doi.org/doi:10.1145/2492007.2492022 page 1. [↑](#footnote-ref-22)
23. Vogels, “Eventually Consistent.” [↑](#footnote-ref-23)
24. Gilbert and Lynch, “Brewer’s Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services." page 3 [↑](#footnote-ref-24)
25. Gilbert and Lynch page 3. [↑](#footnote-ref-25)
26. Vogels, “Eventually Consistent.” [↑](#footnote-ref-26)
27. Liu and Liu, “The Complexity of Weak Consitency. page 3” [↑](#footnote-ref-27)
28. Vogels, “Eventually Consistent.” [↑](#footnote-ref-28)
29. Saurabh V, “Eventual vs Strong Consitency in Distributed Databases,” Hackernoon, July 16, 2017, https://hackernoon.com/eventual-vs-strong-consistency-in-distributed-databases-282fdad37cf7. [↑](#footnote-ref-29)
30. Marc Shapiro et al., “A Comprehensive Study of Convergent and Commutative Replicated Data Types,” *Hal-Inria*, January 13, 2011, https://doi.org/inria-00555588. [↑](#footnote-ref-30)
31. Nitin Savant, Elise Olivares, and Sun-Li Beatteay, “Conclave - A Private and Secure Real-Time Collaborative Text Editor,” Conclave, accessed May 12, 2020, https://conclave-team.github.io/conclave-site/. [↑](#footnote-ref-31)