You Can't Sacrifice Partition Tolerance

In which there are limits to the CAP conjecture.

07 Oct 2010

I've seen a number of distributed databases recently describe themselves as being "CA" –that is, providing both consistency and availability while not providing partition-tolerance. To me, this indicates that the developers of these systems do not understand the CAP theorem and its implications.

A Quick Refresher

In 2000, Dr. Eric Brewer gave a keynote at the *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing*[1] in which he laid out his famous CAP Theorem: *a shared-data system can have at most two of the three following properties: **C**onsistency, **A**vailability, and tolerance to network **P**artitions*. In 2002, Gilbert and Lynch[2] converted "Brewer's conjecture" into a more formal definition with an informal proof. As far as I can tell, it's been misunderstood ever since.

So let's be clear on the terms we're using.

On Consistency

From Gilbert and Lynch[2]:

> Atomic, or linearizable, consistency is the condition expected by most web services today. Under this consistency guarantee, 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.

Most people seem to understand this, but it bears repetition: a system is consistent if an update is applied to all relevant nodes at the same logical time. Among other things, this means that standard database replication is *not strongly consistent*. As anyone whose read replicas have drifted from the master knows, special logic must be introduced to handle replication lag.

That said, consistency which is both instantaneous and global is impossible. The universe simply does not permit it. So the goal here is to push the time

resolutions at which the consistency breaks down to a point where we no longer notice it. Just don't try to act outside your own light cone…

On Availability

Again from Gilbert and Lynch[2]:

> For a distributed system to be continuously available, 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.

Despite the notion of "100% uptime as much as possible," there are limits to availability. If you have a single piece of data on five nodes and all five nodes die, that data is gone and any request which required it in order to be processed cannot be handled.

(N.B.: A `500 The Bees They're In My Eyes` response does not count as an actual response any more than a network timeout does. A response contains the results of the requested work.)

On Partition Tolerance

Once more, Gilbert and Lynch[2]:

> In order to model partition tolerance, the network will be allowed to lose arbitrarily many messages sent from one node to another. When a network is partitioned, all messages sent from nodes in one component of the partition to nodes in another component are lost. (And any pattern of message loss can be modeled as a temporary partition separating the communicating nodes at the exact instant the message is lost.)

This seems to be the part that most people misunderstand.

Some systems cannot be partitioned. Single-node systems (e.g., a monolithic Oracle server with no replication) are incapable of experiencing a network partition. But practically speaking these are rare; add remote clients to the monolithic Oracle server and you get a distributed system which *can* experience a network partition (e.g., the Oracle server becomes unavailable).

Network partitions aren't limited to dropped packets: a crashed server can be thought of as a network partition. The failed node is effectively the only

member of its partition component, and thus all messages to it are "lost" (i.e., they are not processed by the node due to its failure). Handling a crashed machine counts as partition-tolerance. (**Update: I was wrong about this part. See this for more.**) (N.B.: A node which has gone offline is actually the easiest sort of failure to deal with—you're assured that the dead node is not giving incorrect responses to another component of your system.)

For a distributed (i.e., multi-node) 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*. You and I do not work with these types of systems because **they don't exist.**

Given A System In Which Failure Is An Option

Michael Stonebraker's assertions aside, partitions (read: failures) do happen, and the chances that any one of your nodes will fail jumps exponentially as the number of nodes increases:

$$P(\text{any failure}) = 1 - P(\text{individual node not failing})^{\text{number of nodes}}$$

If a single node has a 99.9% chance of not failing in a particular time period, a cluster of 40 has a 96.1% chance not failing. In other words, you've got around a 4% chance that *something* will go wrong. (And this is assuming that your failures are unrelated; in reality, they tend to cascade.)

Therefore, the question you should be asking yourself is:

> In the event of failures, which will this system sacrifice?
> Consistency or availability?

Choosing Consistency Over Availability

If a system chooses to provide Consistency over Availability in the presence of partitions (again, read: failures), it 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).

This is perfectly reasonable. 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.

Choosing Availability Over Consistency

If a system chooses to provide Availability over Consistency in the presence of partitions (all together now: failures), it 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. (Dynamo systems usually offer both of these; Cassandra's hard-coded Last-Writer-Wins conflict resolution being the main exception.)

Again, *this is perfectly reasonable*. There are plenty of data models which are amenable to conflict resolution and for which stale reads are acceptable (ironically, many of these data models are in the financial industry) and for which unavailability results in massive bottom-line losses. (Amazon's shopping cart system is the canonical example of a Dynamo model[3]).

But Never Both

> You cannot, however, choose both consistency and availability in a distributed system.

As a thought experiment, imagine a distributed system which keeps track of a single piece of data using three nodes—A, B, and C—and which claims to be both consistent and available in the face of network partitions. Misfortune strikes, and that system is partitioned 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 either choose availability (Door #1) or you choose consistency (Door #2). 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***.

This is, as you might imagine, rarely true.

A Readjustment In Focus

I think part of the problem with practical interpretations of the CAP theorem, especially with Gilbert and Lynch's formulation, is the fact that most real distributed systems do not require atomic consistency or perfect availability and will never be called upon to perform on a network suffering from arbitrary message loss. Consistency, Availability, and Partition Tolerance are the Platonic ideals of a distributed system—we can partake of them enough to meet business requirements, but the nature of reality is such that there will always be compromises.

When it comes to designing or evaluating distributed systems, then, I think we should focus less on which two of the three Virtues we like most and more on what compromises a system makes as things go bad. (Because they will go bad.)

This brings us to an earlier bit of Brewer wisdom: **yield** and **harvest**, which come from Fox and Brewer's "Harvest, Yield, and Scalable Tolerant Systems"[4].

> We assume that clients make queries to servers, in which case there are at least two metrics for correct behavior: **yield**, which is the probability of completing a request, and **harvest**, which measures the fraction of the data reflected in the response, i.e. the completeness of the answer to the query.

On Yield

In a later article[5], Brewer expands on **yield** and its uses:

> 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.

On Harvest

Harvest is a far more overlooked metric, especially in the age of the relational database. If we imagine working on a search engine, however, we can imagine there being separate indexes for each word. The index of web pages which use the word "cute" is on node A, the index of web pages which use the word

"baby" is on node B, and the index for the word "animals" is on machine C. A search, then, for "cute baby animals" which combined results from nodes $\{A, B, C\}$ would have a 100% harvest. If node B was unavailable, however, we might return a result for just "cute animals," which would be a harvest of 66%. In other words:

$$\text{harvest} = \frac{\text{data available}}{\text{total data}}$$

Another example would be a system which stores versions of documents. In the event that some nodes are down, the system could choose to present the most recent version of a document that it could find, even if it knew there was a probability that it was not the most recent version it had stored.

A Better Heuristic

Whether a system favors yield or harvest (or is even *capable* of reducing harvest) tends to be an outcome of its design.

As Brewer puts it[5]:

> 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.

In terms of general advice to people building distributed systems (and really, who isn't these days?), I think the following is far more effective:

> Despite your best efforts, 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.

Well Now What

It's incredibly unlikely this will change the way people will build distributed systems—far more of our motivation comes from business concerns ("our shopping carts *cannot* go down" or "there would be no way for us to reconcile this later"). What I'd like to see, though, is far fewer people unknowingly describing their systems as logical impossibilities.

tl;dr

Of the CAP theorem's Consistency, Availability, and Partition Tolerance, Partition Tolerance is mandatory in distributed systems. You cannot **not** choose it. 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.

Updated October 8, 2010

Dr. Brewer approves (somewhat):

> I really need to write an updated CAP theorem paper. But until then, this is pretty good: http://bit.ly/btkdJ5 (from @coda)
>
> — Abstract Cloud (@eric_brewer) October 9, 2010

Updated October 21, 2010

Dr. Stonebraker does not approve (somewhat).

Updated October 22, 2010

In his response, Dr. Stonebraker says:

> In Coda's view, the dead node is in one partition and the remaining N-1 nodes are in the other one. The guidance from the CAP theorem is that you must choose either A or C, when a network partition is present. As is obvious in the real world, it is possible to achieve both C and A in this failure mode. You simply failover to a replica in a transactionally consistent way. Notably, at least Tandem and Vertica have been doing exactly this for years. Therefore, considering a node failure as a partition results in an obviously inappropriate CAP theorem conclusion.

He is correct: a dead (or wholly partitioned) node can receive no requests, and so the other nodes can easily compensate without compromising consistency or availability here. My error lies in forgetting that Gilbert and Lynch's formulation of availability requires only non-failing nodes to respond, and that without ≥1 live nodes in a partition there isn't the option for split-brain syndrome. I regret the error and thank both him and Dr. Brewer for pointing this out.

That said, Dr. Stonebraker's assertion that "surviving [partitions] will not 'move the needle' on availability because higher frequency events will cause global outages" is wrong. Multi-node failures may be rarer than single-node

failures, but they are still common enough to have serious effects on business. In my limited experience I've dealt with long-lived network partitions in a single data center (DC), PDU failures, switch failures, accidental power cycles of whole racks, whole-DC backbone failures, whole-DC power failures, and a hypoglycemic driver smashing his Ford pickup truck into a DC's HVAC system. And I'm not even an ops guy.

The fact of the matter is that most real-world systems require substantially less in the way of consistency guarantees than they do in the way of availability guarantees. Amazon's shopping carts, for example, do not require the full ACID treatment, nor do Twitter's timelines or Facebook's news feeds or Google's indexes. Even the canonical example of an isolated transaction–a transfer of funds between bank accounts–happens with a 24-hour window of indeterminacy. In fact, one of the few financial transactions which actually resembles a database transaction is the physical exchange of cash for goods–a totally analog experience.

But whereas failures of consistency are tolerated or even expected, just about every failure of availability means lost money. Every failed Google search means fewer ads served and advertisers charged; every item a user can't add to their shopping cart means fewer items sold; every unprocessed credit charge risks a regulatory fine. The choice of availability over consistency is a business choice, not a technical one.

Given this economic context, it becomes clear why most practitioners at any interesting scale meet their business needs using highly-available, eventually consistent systems.

References (i.e., Things You Should Read)

1. Brewer. Towards robust distributed systems. Proceedings of the Annual ACM Symposium on Principles of Distributed Computing (2000) vol. 19 pp. 7—10 ↩

2. Gilbert and Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. ACM SIGACT News (2002) vol. 33 (2) pp. 59 ↩ ↩2 ↩3 ↩4

3. DeCandia et al. Dynamo: Amazon's highly available key-value store. SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles (2007) ↩

4. Fox and Brewer. Harvest, yield, and scalable tolerant systems. Hot Topics in Operating Systems, 1999. Proceedings of the Seventh Workshop on (1999) pp. 174—178 ↩

5. Brewer. Lessons from giant-scale services. Internet Computing, IEEE (2001) vol. 5 (4) pp. 46-55 ↩ ↩2