BLOG@CACM

Errors in Database Systems, Eventual Consistency, and the CAP Theorem

By Michael Stonebraker
April 5, 2010
Comments (12)

Recently, there has been considerable renewed interest in the CAP theorem [1] for database management system (DBMS) applications that span multiple processing sites. In brief, this theorem states that there are three interesting properties that could be desired by DBMS applications:

C: Consistency. The goal is to allow multisite transactions to have the familiar all-or-nothing semantics, commonly supported by commercial DBMSs. In addition, when replicas are supported, one would want the replicas to always have consistent states.

A: Availability. The goal is to support a DBMS that is always up. In other words, when a failure occurs, the system should keep going, switching over to a replica, if required. This feature was popularized by Tandem Computers more than 20 years ago.

P: Partition-tolerance. If there is a network failure that splits the processing nodes into two groups that cannot talk to each other, then the goal would be to allow processing to continue in both subgroups.

The CAP theorem is a negative result that says you cannot simultaneously achieve all three goals in the presence of errors. Hence, you must pick one objective to give up.

In the NoSQL community, this theorem has been used as the justification for giving up consistency. Since most NoSQL systems typically disallow transactions that cross a node boundary, then consistency applies only to replicas. Therefore, the CAP theorem is used to justify giving up consistent replicas, replacing this goal with “eventual consistency.” With this relaxed notion, one only guarantees that all replicas will converge to the same state eventually, i.e., when network connectivity has been re-established and enough subsequent time has elapsed for replica cleanup. The justification for giving up C is so that the A and P can be preserved.

The purpose of this blog post is to assert that the above analysis is suspect, and that recovery from errors has more dimensions to consider. We assume a typical hardware model of 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.

Let’s start with a discussion of what causes errors in databases. The following is at least a partial list:

1) Application errors. The application performed one or more incorrect updates. Generally, this is not discovered for minutes to hours thereafter. The database must be backed up to a point before the offending transaction(s), and subsequent activity redone.

2) Repeatable DBMS errors. The DBMS crashed at a processing node. Executing the same transaction on a processing node with a replica will cause the backup to crash. These errors have been termed Bohr bugs. [2]

3) Unrepeatable DBMS errors. The database crashed, but a replica is likely to be ok. These are often caused by weird corner cases dealing with asynchronous operations, and have been termed Heisenbugs [2]

4) Operating system errors. The OS crashed at a node, generating the “blue screen of death.”

5) A hardware failure in a local cluster. These include memory failures, disk failures, etc. Generally, these cause a “panic stop” by the OS or the DBMS. However, sometimes these failures appear as Heisenbugs.

6) A network partition in a local cluster. The LAN failed and the nodes can no longer all communicate with each other.

7) A disaster. The local cluster is wiped out by a flood, earthquake, etc. The cluster no longer exists.

8) A network failure in the WAN connecting clusters together. The WAN failed and clusters can no longer all communicate with each other.

First, note that errors 1 and 2 will cause problems with any high availability scheme. In these two scenarios, there is no way to keep going; i.e., availability is impossible to achieve. Also, replica consistency is meaningless; the current DBMS state is simply wrong. Error 7 will only be recoverable if a local transaction is only committed after the assurance that the transaction has been received by another WAN-connected cluster. Few application builders are willing to accept this kind of latency. 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. Put differently, the application designer chooses to suffer data loss when a rare event (such as a disaster) occurs, because the performance penalty for avoiding it is too high.

As such, errors 1, 2, and 7 are examples of cases for which the CAP theorem simply does not apply. Any real system must be prepared to deal with recovery in these cases. The CAP theorem cannot be appealed to for guidance.

Let us now turn to cases where the CAP theorem might apply. Consider error 6 where a LAN partitions. In my experience, this is exceedingly

rare, especially if one replicates the LAN (as Tandem did). Considering local failures (3, 4, 5, and 6), the overwhelming majority cause a single node to fail, which is a degenerate case of a network partition that is easily survived by lots of algorithms. Hence, in my opinion, one is much better off giving up P rather than sacrificing C. (In a LAN environment, I think one should choose CA rather than AP). Newer SQL OLTP systems (e.g., VoltDB and NimbusDB) appear to do exactly this.

Next, consider error 8, a partition in a WAN network. There is enough redundancy engineered into today's WANs that a partition is quite rare. My experience is that local failures and application errors are way more likely. 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.

Lastly, consider a slowdown either in the OS, the DBMS, or the network manager. This may be caused by skew in load, buffer pool issues, or innumerable other reasons. The only decision one can make in these scenarios is to "fail" the offending component; i.e., turn the slow response time into a failure of one of the cases mentioned earlier. In my opinion, this is almost always a bad thing to do. One simply pushes the problem somewhere else and adds a noticeable processing load to deal with the subsequent recovery. Also, such problems invariably occur under a heavy load–dealing with this by subtracting hardware is going in the wrong direction.

Obviously, one should write software that can deal with load spikes without failing; for example, by shedding load or operating in a degraded mode. Also, good monitoring software will help identify such problems early, since the real solution is to add more capacity. Lastly, self-reconfiguring software that can absorb additional resources quickly is obviously a good idea.

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.

References

[1] Eric Brewer, "Towards Robust Distributed Systems," http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf

[2] Jim Gray, "Why Do Computers Stop and What Can be Done About It," Tandem Computers Technical Report 85.7, Cupertino, Ca., 1985. http://www.hpl.hp.com/techreports/tandem/TR-85.7.pdf

Disclosure: In addition to being an adjunct professor at the Massachusetts Institute of Technology, Michael Stonebraker is associated with four startups that are either producers or consumers of database technology.

Comments

Dwight Merriman

April 07, 2010 12:49

"Degenerate network partitions" is a very good point - in practice i have found that most network partitions in the real world are of this class.

I like to term certain classes of network partitions "trivial". By trivial, if there are no clients in the partitioned region, or if there are servers in the partitioned region, it is then trivial. So it could involve more than one machine, but it then readily handled.

Alaric Snell-Pym

April 07, 2010 01:28

I think a lot of the discussion about distributed database semantics, much like a lot of the discussion about SQL vs. NoSQL, has been somewhat clouded by a shortage of pragmatism. So an analysis of the CAP theorem in terms of actual practical situations is a welcome change :-)

My company (GenieDB) has developed a replicated database engine that provides "AP" semantics, then developed a "consistency buffer" that provides a consistent view of the database - as long as there are no server or network failures; then providing a degraded service, with some fraction of the records in the database becoming "eventually consistent" while the rest remain "immediately consistent". Providing a *degraded* service rather than *no* service is a good thing, as it reduces the cost of developing applications that use a distributed database compared to existing solutions - but not something that somebody too blinded by the CAP theorem might consider!

In a similar vein, we've provided both NoSQL and SQL interfaces to our database, with different tradeoffs available in both, and both can be used at once on the same data... people need to stop fighting over X vs. Y, and think about how to combine the best of both in practical ways!

Michael Malone

April 07, 2010 01:47

Hey Michael,

Interesting read. I'm with you on most points. I do have a couple comments though...

Failure mode 1 is an application error.

Failure mode 2 should result in a failed write. It shouldn't be too hard to trap errors programmatically and handle them intelligently / not

propagate them. Of course the devil's in the details and hardware / interpreter / other problems in components that are outside of the DBs control can make things more difficult. These are the sorts of issues that tend to stick around until a system is "battle tested" and run in a couple of large / high volume operations.

Failure modes 3, 4, 5, 6 (partition in a local cluster) - this seems to be where the meat of your argument is... but you totally gloss over your solution. I'm not sure I believe that network partitions (even single node failures) are easily survived by lots of algorithms... Or, more specifically, I don't believe that they can be survived while maintaining consistency (in the CAP sense, not the ACID sense). I threw together a super simplified "proof" of why consistency is essentially impossible in this situation in a recent talk. See http://www.slideshare.net/mmalone/scaling-gis-data-in-nonrelational-data-stores - slides 16 through 20. What algorithms are there to get around this? If a replica is partitioned you either can't replicate to it and have to fail (unavailable) or you can't replicate to it and succeed anyways (replica is inconsistent).

I also don't buy the argument that partitions (LAN or WAN) are rare and therefore we shouldn't worry about them. For a small operation this may be true, but when you're doing a million operations a second then a one-in-a-million failure scenario will happen every second.

Failure mode 7 will probably result in some data loss unless (as you mention) you're willing to live with the latency of waiting for durable multi-datacenter writes to occur. But having that option is definitely nice, and that's a trade off that I'd like to be able to make on a per-write basis. I may choose to accept that latency when I'm recording a large financial transaction, for example. Another thought related to this issue - in a lot of ways writing something to memory on multiple nodes is more "durable" than writing it to disk on one. So you may be able to do multi-DC replicated writes in memory with tolerable latency assuming your DCs are close enough that the speed of light isn't limiting. That should get you durability up to the point where the entire eastern seaboard disappears, at least.

Failure mode 8 is another core issue that I think you're glossing over. WAN failures (particularly short-lived ones) can and do happen on a regular basis. It's true that routing issues are typically resolved quickly, but it's another law-of-large-numbers thing. Amazon AWS had an issue that took an entire data center offline a while back, for example. Shit happens. In CAP terms this is really the same thing as a failure modes 3, 4, 5, 6, and 7 though. So the same arguments apply. Re: your argument that only a small segment splits - what happens when a read comes into the small split segment (maybe from a client in the same datacenter)? If data has been updated on the larger segment it couldn't have been replicated, so again you're either serving stale data or your data store is unavailable.

Thanks for putting this together, it was an interesting read. Looking forward to hearing more about some of these issues!

Paul Prescod

April 07, 2010 04:42

I find this blog post quite confusing. The key to my confusion is this paragraph: "Error 7 will only be recoverable if a local transaction is only committed after the assurance that the transaction has been received by another WAN-connected cluster. Few application builders are willing to accept this kind of latency."

If you are correct that few applications are willing to accept the latency of a synchronous write to another WAN-connected cluster, then the only alternatives are asynchronous write or no write at all. Asynchronous write is Eventual Consistency. Eventually the remote cluster is consistent with the local one.

So you claim that you are arguing against eventual consistency, but I interpret your words as: "Eventual Consistency is the only option when you are working at WAN scale." That is the same conclusion that Amazon, Yahoo, Google, Facebook, Twitter and Rackspace have made. Are you agreeing with them or disagreeing with them?

You claim that LAN failures are rare. Facebook has 60,000 servers at last count. Some people estimate that Google has upwards of 250,000 servers. Yahoo has around 150,000 servers. Rackspace has more than 50,000. Amazon must be in the ballpark somewhere there. It seems to me that one cannot operate at that scale without frequent LAN and server failures. Eventual Consistency is just the mechanism by which transient outages are seamlessly handled at massive scale.

I certainly do not need to deal with that volume of data. But I do look forward to seamless outsourced management of the data that I do control (I'm not in a highly secretive industry). What those big vendors sell as their cloud database products are variants of what they use in-house, and naturally handle all of the LAN, WAN and node outages that would otherwise make my application unavailable.

If they are willing to go through the engineering effort to make downtime nearly non-existent (as opposed to "rare") then I am happy to pay them a reasonable monthly fee for the use of their technology. I look forward to doing so as these products mature.

Paul Prescod

April 07, 2010 04:48

One more thought: Most people discussing these trends do not know that Cassandra (for example) allows you to decide on a *per-query* basis (whether a read or write) whether you want strong consistency (ConsistencyLevel.ALL) or weak (ConsistencyLevel.ONE). The choice is write there in the API and the developer gets to decide based on the semantics of the query. "Writing an instance message...better use a low consistency level for low latency. Writing out a financial transaction? Better make sure that every replica knows about it so we don't double charge."

Not that I would manage financial data in a datastore versioned 0.7 of course....

Michael Malone

April 08, 2010 12:29

Paul, small correction... and I could be wrong here. But, to be fair, I believe even with Cassandra's ConsistencyLevel.ALL you may only get eventually consistent behavior when node failures occur. I believe ALL means "write to N nodes," where N is your replication factor. If all N of the authoritative nodes for a given key are down, I believe Cassandra will still write to the next N nodes in the cluster. And, if I'm understanding things correctly, I believe the argument Michael is trying to make is that it shouldn't... Instead it should fail the write.

The trouble is, when the N authoritative nodes come back up, reads will go to them. And even with ConsistencyLevel.ALL your reads will be inconsistent until hinted-handoff, anti-entropy, or read-repair kicks in. That should be pretty damn fast though. I suppose one solution to this would be to run hintend-handoff before unavailable nodes rejoin the cluster. Not sure if that's feasible though.

In all other cases (failure and non-failure) as long as one of the N authoritative nodes for an item is up, I believe ConsistencyLevel.ALL will give you strong consistency. Even with ConsistencyLevel.ONE Cassandra will give you strong consistency if there are no partitions.

So Michael's argument seems to be that if you get to the point where lots of failures pile up you should just become unavailable, because it's pretty rare that you actually get to that point (e.g., the point where an authoritative quorum is unavailable for a read/write). I still disagree. I believe that large scale operations are often in a state where more than N nodes are unavailable, which would mean some fraction of data is unavailable. It's also nice operationally to have the ability to restart, decommission, and otherwise futz with individual nodes in a cluster without worrying about breaking stuff. But reasonable minds can differ.

Lou Montulli

April 08, 2010 07:01

I rarely see discussions that include consensus quorum database design. This is probably due to the lack of consensus quorum relational implementations. Consensus quorum solves many of the CAP issues and can be used to make an incredibly robust implementation.

What do you think?

Russell Okamoto

April 09, 2010 10:03

Thanks Mike for an eloquent explanation of CAP considerations.

In recent discourse about the CAP theorem, I wonder if another of Eric Brewer's important contributions, "Harvest, Yield, and Scalable Tolerant Systems", http://citeseer.ist.psu.edu/fox99harvest.html, has been overlooked. Brewer's 1999 paper describes strategies for dealing with CAP based on probabilistic availability and application decomposition, resulting in graceful degradation and partial failure modes. By decomposing applications into independent subsystems, for example, the failure of one subsystem can be isolated from other subsystems enabling the system at large to still operate (but possibly with reduced functionality).

Since individual subsystems have different requirements for what can be relaxed--e.g., one subsystem may prioritize CA while another may prioritize AP--decomposition also hints at the need for "tunable CAP", i.e., the ability for system designers to choose where in their system-wide workflows to use CA or CP or AP. Not all activities (subsystems) in a complex workflow have the same data requirements--a billing subsystem might prioritize consistency whereas a web recommendation subsystem might deprioritize consistency and permit eventually consistent, stale reads. Hence being able to relax C or A or P at different places and times and subsystems can result in an optimal system-wide design. So to evoke another Stonebrakerism: "one size does not fit all". Tunable CAP ideas have in fact already been incorporated (for several years) into state-of-the-art data fabric products currently adopted by all top banking institutions for scalable, continuously available, high-speed trading applications (e.g., http://www.gemstone.com/hardest-problems#whatshard).

As an approach for dealing with CAP issues, application decomposition also enables system designers to identify and exploit "orthogonal mechanisms" (Brewer page 3) for non-invasive subsystem management. As an example, data schema interdependence policies can become orthogonal mechanism that allows users to configure requisite availability semantics for individual subsystems. The value here is that data interdependency policies can be declared at "compile-time" (system configuration time) enabling early detection of problematic/risky configurations that can be quickly corrected by administrators. Second, at "run-time", process survivability decisions can be guided by the rational logic of schema data dependencies rather than randomness of primary network partitioning models (where the losing "side" of a partition gets killed arbitrarily).

And finally, a consequence of CAP and the use of application decomposition is that deconstructing a system into holistic, autonomous business chunks that communicate via message passing and are transactionally independent leads to a service "entity" abstraction ala Pat Helland's "Life Beyond Distributed Transactions", http://www.cidrdb.org/cidr2007/papers/cidr07p15.pdf. This service entity model, in turn, can be thought of as an Actor model for concurrent computation. So to summarize:

CAP ==> Application Decomposition ==> Service Entities ==> Actor Model

CACM Administrator

April 15, 2010 02:22

There has been quite a collection of comments to my posting. Rather than respond individually, I will make two points that addresses several of

the comments. Specifically, I will discuss error frequencies and draw some conclusions as a result, and clarify my comment about application builders not being willing to pay for two phase commit.

The various respondents have posited various error characteristics, and then drawn conclusions as a result. At this point there is very little hard data that I am aware of concerning exactly how frequently various DBMS errors occur. Also, such error frequency is clearly a function of the operating environment, and that obviously varies.

There is considerable data on specific hardware failures; for example, the frequency of disk crashes or RAM failures. However, what is really needed is the relative frequency of all of the failure modes mentioned in my blog. In the 1980's Jim Gray published just such an analysis of errors in Tandem systems. Unfortunately, I can't seem to locate that paper, which was Tandem-specific, in any case. In short, I would really welcome somebody publishing data on the current distribution of errors in DBMS applications. In the absence of hard data, here is my (not scientific) spin on the topic. Remember "your mileage will vary."

1) Application errors. The application performed one or more incorrect updates. Generally this is not discovered for minutes to hours thereafter. The data base must be backed-up to a point before the offending transaction(s), and subsequent activity redone.

My sense is that this happens once every three years per application. However, it depends a lot on how well the application is tested.

2) Repeatable DBMS errors. The DBMS crashed at a processing node. Executing the same transaction on a processing node with a replica will cause the backup to crash. These errors have been termed Bohr bugs [2].

Most Bohr bugs occur during the early days of a release; i.e., most of them are fairly quickly beaten out of a DBMS release. Most serious users decline to use a new release of any system software until this break-in period is over. As such, I believe Bohr bugs are quite rare, happening perhaps once every 12 months, per database. Of course, the corrective action is workarounds and vendor patches.

3) Unrepeatable DBMS errors. The database crashed, but a replica is likely to be ok. These are often caused by weird corner cases dealing with asynchronous operations, and have been termed Heisenbugs [2].

Heisenbugs generally occur under heavy load and are devilishly hard to locate and even harder to test for in advance. My experience is that a Heisenbug also occurs about every 12 months per database. Again, these are rarely seen by lightly loaded systems; hence the rate of Heisenbugs is very sensitive to operating characteristics.

4) Operating system (OS) errors. The OS crashed at a node, generating the "blue screen of death."

The frequency of OS errors depends a great deal on which OS is being run. The gold standard is MVS, which is hyper-reliable, typically crashing less than once a year. Most other OSs crash considerably more frequently. In my experience, a production OS on a given node crashes about every 6 months.

5) A hardware failure in a local cluster. These include memory failures, disk failures, etc. Generally, these cause a "panic stop" by the OS or the DBMS. However, sometimes these appear as Heisenbugs.

In my experience a disk error on a node occurs once a year (though this depends on how many disks are attached), and a hardware error elsewhere in the node also occurs once a year. In aggregate, a node crash occurs every six months. Also, hardware problems are quite "time-skewed," i.e., both "infant mortality" and "old age" are problematic.

6) A network partition in a local cluster. The LAN failed and the nodes can no longer all communicate with each other.

In my experience, a LAN partition failure occurs rarely, perhaps every fourth year. It is way more likely that the LAN connection on the local node fails, causing a single node to be separated from the rest of the LAN. This "single node separation" might occur every six months.

7) A disaster. The local cluster is wiped out, by a flood, earthquake, etc. The cluster no longer exists.

In my experience, a node totally disappears every 20 years or less.

8) A network failures in the WAN connecting clusters together. The WAN failed and clusters can no longer all communicate with each other.

In my experience, a WAN failure causing a partition in a private network (as opposed to the Internet) is quite rare, occurring perhaps once every other year. Again in my experience, the most likely failure is in the interface between a local cluster and the WAN, i.e., a partition that disconnects a single node from the WAN is 20 times more likely than a more general network partition. In other words, a site cannot connect to the WAN once a month.

Now consider an operating environment consisting of 15 nodes in a LAN cluster. The cluster is dedicated to a single DBMS to which three applications are connected. Consider all of the failure modes above and divide them into four groups:

Group A): The application cannot continue (error modes 1 and 2). In our scenario, an error in Group A happens every six months. Regardless of your high availability strategy, blocking is inevitable.

Group B): There is an easy-to-survive error consisting of a single node failure or a single node being isolated from its processing peers in a cluster (errors 3, 4, 5 and some of 6). An error in this group happens about every six days.

Group C): There is a failure that causes a cluster to fail or to be isolated from its peers (error modes 7 and some of 8). This error happens once a month.

Group D): There is a general network partition (some of error modes 6 and 8). This error happens every couple of years.

In round numbers, an easy-to-survive error (B and C above) happens once a week, an error which forces the system to block occurs every six months and a general network partition occurs every couple of years. You can easily do the above analysis assuming your specific hardware environment and failure characteristics. This will move the rate at which the various errors happen up or down; however, I would be quite

surprised the relative frequencies noted above change a great deal.

As such, the point I am trying to make is that forced blocking from application errors and Bohr bugs is way more likely than general network partitions. Put differently, staying up during general network partitions does not "move the blocking needle," because they are dwarfed in frequency by blocking errors.

Hence, the P of the CAP theorem is a rare event, and one that is dominated by other errors that force blocking. Hence, surviving general network partitions without blocking does not improve availability very much. As such, trading consistency (C) for partition tolerance (P) does not appear to be warranted.

The second point I am trying to make concerns WAN replication. Consider a specific series of events (this is obviously not the only way to do the work, but it is the way several systems I am aware of do things):

1) Application sends transaction to a co-ordinator (generally co-located with the "primary" storage node)
2) Co-ordinator does the transaction locally; in parallel sends the transaction to a "backup"
3) Primary responds to the co-ordinator "done"
4) Co-ordinator responds "done" to application
5) Backup site finishes executing the transaction.

This is an active-active system, in which the transaction is executed twice, once at the primary and once at the backup, which is assumed to be at the other end of a WAN. A very possible failure scenario is the co-ordinator finishes the transaction at the primary and responds done (step 4) and then fails (let's say the rare but possible disaster–error 7). In this case, the transaction at the primary is permanently lost. Consider now a network failure in which the message to the backup is lost. In this double failure scenario, the transaction is completely lost, even though the user was sent a "done." Hence, consistency (or eventual consistency for that matter) is not possible. The only way to avoid data loss in this scenario is to perform a two-phase commit, a long latency tactic that few are willing to pay for. The point I am trying to make is that there are rare, corner cases, such as the above, where application designers are not willing to pay for either consistency or eventual consistency. Put differently, consistency and eventual consistency are not, in practice, 100% guarantees, but are engineering tradeoffs between cost, latency, and code complexity.
--Michael Stonebraker, April 15, 2010

Frank Patz-Brockmann

May 07, 2010 02:18

As I understood it, consistency and consistency are not the same. If your database is a model of the real world (financial transactions, say), not being consistent is disastrous for the application's purpose. If your database *is* the virtual world (twitter, say), loosing or delaying a user's update is far less important and can be easily traded for the ability to keep the application available for millions of other users.

View More Comments