

:julianbrowne

[home](#) [archive](#) [about](#) [rss](#)

16

Brewer's CAP Theorem

The kool aid Amazon and Ebay have been drinking

By Julian Browne on January 11, 2009. Filed Under [architecture](#), [business](#), [strategy](#)

On Friday 4th June 1976, in a small upstairs room away from the main concert auditorium, the [Sex Pistols](#) kicked off their first gig at Manchester's [Lesser Free Trade Hall](#). There's some confusion as to who exactly was there in the audience that night, partly because there was another concert just six weeks later, but mostly because it's considered to be a gig that [changed western music culture](#) forever. So iconic and important has that appearance become that David Nolan wrote a book, [I Swear I Was There: The Gig That Changed the World](#), investigating just whose claim to have been present was justified. Because the 4th of June is generally considered to be the genesis of punk rock⁶.

Prior to this (in fact since around 1971) there had been a number of [protopunk](#) bands, such as the [New York Dolls](#) and the [Velvet Underground](#), but it was this one set by the Sex Pistols that in music folklore started the revolution that set in motion the driving guitars of the [Buzzcocks](#), the plaintive wailing of [The Smiths](#), the eclectic syncopations of the [The Fall](#), the rising majesty of [Joy Division](#) and Simply Red (I guess you can't have everything).

Wednesday 19th July 2000, may not go down in popular culture with quite the same magnitude but it's had a similar impact on internet scale business as the Sex Pistols did on music a quarter of a century earlier, for that was the [keynote speech](#) by [Eric Brewer](#) at the ACM Symposium on the [Principles of Distributed Computing](#) (PODC).

The Sex Pistols had shown that barely-constrained fury was more important to their contemporaries than art-school structuralism, giving anyone with three chords and something to say permission to start a band. Eric Brewer, in what became known as Brewer's Conjecture, said that as applications become more web-based we should stop worrying about data consistency, because if we want high availability in these new distributed applications, then guaranteed consistency of data is something we *cannot have*, thus giving anyone with three servers and a keen eye for customer experience permission to start an internet scale business. Disciples of Brewer (present that day or later converts) include the likes of [Amazon](#), [EBay](#), and [Twitter](#).

Two years later, in 2002, [Seth Gilbert](#) and [Nancy Lynch](#) of MIT, [formally proved](#) Brewer to be correct and thus Brewer's Theorem was born.

Brewer's (CAP) Theorem

So what exactly is Brewer's Theorem, and why does it warrant comparison with a 1976 punk gig in Manchester?

Brewer's 2000 talk was based on his theoretical work at UC Berkley and observations from running [Inktomi](#),

though Brewer and others were talking about trade-off decisions that need to be made in highly scalable systems years before that (e.g. "[Cluster-Based Scalable Network Services](#)" from SOSP in 1997 and "[Harvest, yield, and scalable tolerant systems](#)" in 1999) so the contents of the presentation weren't new and, like many of these ideas, they were the work of many smart people (as I am sure Brewer himself would be quick to point out).

What he said was there are three core [systemic requirements](#) that exist in a special relationship when it comes to designing and deploying applications in a distributed environment (he was talking specifically about the web but so many corporate businesses are multi-site/multi-country these days that the effects could equally apply to your data-centre/LAN/WAN arrangement).

The three requirements are: **Consistency**, **Availability** and **Partition Tolerance**, giving Brewer's Theorem its other name - **CAP**.

To give these some real-world meaning let's use a simple example: you want to buy a copy of [Tolstoy's War & Peace](#) to read on a particularly long vacation you're starting tomorrow. Your favourite web bookstore has one copy left in stock. You do your search, check that it can be delivered before you leave and add it to your basket. You remember that you need a few other things so you browse the site for a bit (have you ever bought just *one* thing online? Gotta maximise the parcel dollar). While you're reading the customer reviews of a suntan lotion product, someone, somewhere else in the country, arrives at the site, adds a copy to their basket and goes right to the checkout process (they need an urgent fix for a wobbly table with one leg *much* shorter than the others).

- **Consistency**

A service that is *consistent* operates fully or not at all. Gilbert and Lynch use the word "atomic" instead of consistent in their proof, which makes more sense technically because, strictly speaking, consistent is the C in [ACID](#) as applied to the ideal properties of database transactions and means that data will never be persisted that breaks certain pre-set constraints. But if you consider it a preset constraint of distributed systems that multiple values for the same piece of data are not allowed then I think the leak in the abstraction is plugged (plus, if Brewer had used the word atomic, it would be called the AAP theorem and we'd all be in hospital every time we tried to pronounce it).

In the book buying example you can add the book to your basket, or fail. Purchase it, or not. You can't half-add or half-purchase a book. There's one copy in stock and only one person will get it the next day. If both customers can continue through the order process to the end (i.e. make payment) the lack of consistency between what's in stock and what's in the system will cause an issue. Maybe not a huge issue in this case - someone's either going to be bored on vacation or spilling soup - but scale this up to thousands of inconsistencies and give them a monetary value (e.g. trades on a financial exchange where there's an inconsistency between what you think you've bought or sold and what the exchange record states) and it's a huge issue.

We might solve consistency by utilising a database. At the correct moment in the book order process the number of War and Peace books-in-stock is decremented by one. When the other customer reaches this point, the cupboard is bare and the order process will alert them to this *without* continuing to payment. The first operates fully, the second not at all.

Databases are great at this because they focus on ACID properties and give us Consistency by also giving us Isolation, so that when Customer One is reducing books-in-stock by one, and simultaneously increasing books-in-basket by one, any intermediate states are isolated from Customer Two, who has to wait a few milliseconds while the data store is made consistent.

- **Availability**

Availability means just that - the service is available (to operate fully or not as above). When you buy the book you want to get a response, not some browser message about the web site being uncommunicative. Gilbert & Lynch in their proof of CAP Theorem make the good point that availability most often deserts you when you need it most - sites tend to go down at busy periods precisely because they are busy. A service that's available but not being accessed is of no benefit to anyone.

- **Partition Tolerance**

If your application and database runs on one box then (ignoring scale issues and assuming all your code is perfect) your server acts as a kind of atomic processor in that it either works or doesn't (i.e. if it has crashed it's not available, but it won't cause data inconsistency either).

Once you start to spread data and logic around different nodes then there's a risk of partitions forming. A partition happens when, say, a network cable gets chopped, and Node A can no longer communicate with Node B. With the kind of distribution capabilities the web provides, temporary partitions are a relatively common occurrence and, as I said earlier, they're also not that rare inside global corporations with multiple data centres.

Gilbert & Lynch defined partition tolerance as:

No set of failures less than total network failure is allowed to cause the system to respond incorrectly

and noted Brewer's comment that a one-node partition is equivalent to a server crash, because if nothing can connect to it, it may as well not be there.

The Significance of the Theorem

CAP Theorem comes to life as an application scales. At low transactional volumes, small latencies to allow databases to get consistent has no noticeable affect on either overall performance or the user experience. Any load distribution you do undertake, therefore, is likely to be for systems management reasons.

But as activity increases, these pinch-points in throughput will begin limit growth and create errors. It's one thing having to wait for a web page to come back with a response and another experience altogether to enter your credit card details to be met with "HTTP 500 java.lang.schrodinger.purchasingerror" and wonder whether you've just paid for something you won't get, not paid at all, or maybe the error is immaterial to this transaction. Who knows? You are unlikely to continue, more likely to shop elsewhere, and very likely to phone your bank.

Either way this is not good for business. Amazon [claim](#) that just an extra one tenth of a second on their response times will cost them 1% in sales. Google [said](#) they noticed that just a half a second increase in latency caused traffic to drop by a fifth.

I've written a little about scalability [before](#), so won't repeat all that here except to make two points: the first is that whilst addressing the problems of scale might be an architectural concern, the initial discussions are not. They are business decisions. I get very tired of hearing, from techies, that such-and-such an approach is not warranted because current activity volumes don't justify it. It's not that they're wrong; more often than not they're quite correct, it's that to limit scale from the outset is to implicitly make revenue decisions - a factor that should be made explicit during business analysis.

The second point is that once you embark on discussions around *how* to best scale your application the world falls broadly into two ideological camps: the database crowd and the non-database crowd.

The database crowd, unsurprisingly, like database technology and will tend to address scale by talking of things like [optimistic locking](#) and [sharding](#), keeping the database at the heart of things.

The non-database crowd will tend to address scale by managing data outside of the database environment (avoiding the relational world) for as long as possible.

I think it's fair to say that the former group haven't taken to CAP Theorem with quite the same gusto as the latter (though they are [talking about it](#)). This is because if you have to drop one of consistency, availability, or partition tolerance, many opt to drop consistency which is the *raison d'être* of the database. The logic, no doubt, is that availability and partition-tolerance keep your money-making application alive, whereas inconsistency just feels like one of those things you can work around with clever design.

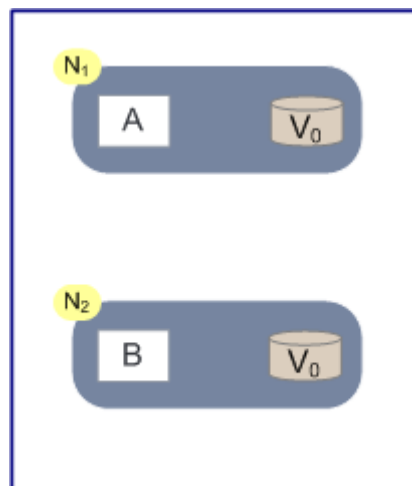
Like so much else in IT, it's not as black and white as this. Eric Brewer, on slide 13 of his PODC talk, when comparing ACID and it's informal counterpart [BASE](#) even says "I think it's a spectrum". And if you're interested in this as a topic (it's slightly outside of what I want to talk about here) you could do worse than start with a paper called "[Design and Evaluation of a Continuous Consistency Model for Replicated Services](#)" by Haifeng Yu and Amin Vahdat. Nobody should interpret CAP as implying the database is dead.

Where both sides agree though is that the answer to scale is distributed parallelisation not, as was once thought, supercomputer grunt. Eric Brewer's influence on the [Network of Workstations](#) projects of the mid-nineties led to the architectures that exposed CAP theorem, because as he says in another presentation on [Inktomi and the Internet Bubble](#) the answer has always been processors working in parallel:

If they're not working in parallel you have no chance to get the problem done in a reasonable amount of time. This is a lot like anything else. If you have a really big job to do you get lots of people to do it. So if you are building a bridge you have lots of construction workers. That's parallel processing also. So a lot of this will end up being "how do we mix parallel processing and the internet?"

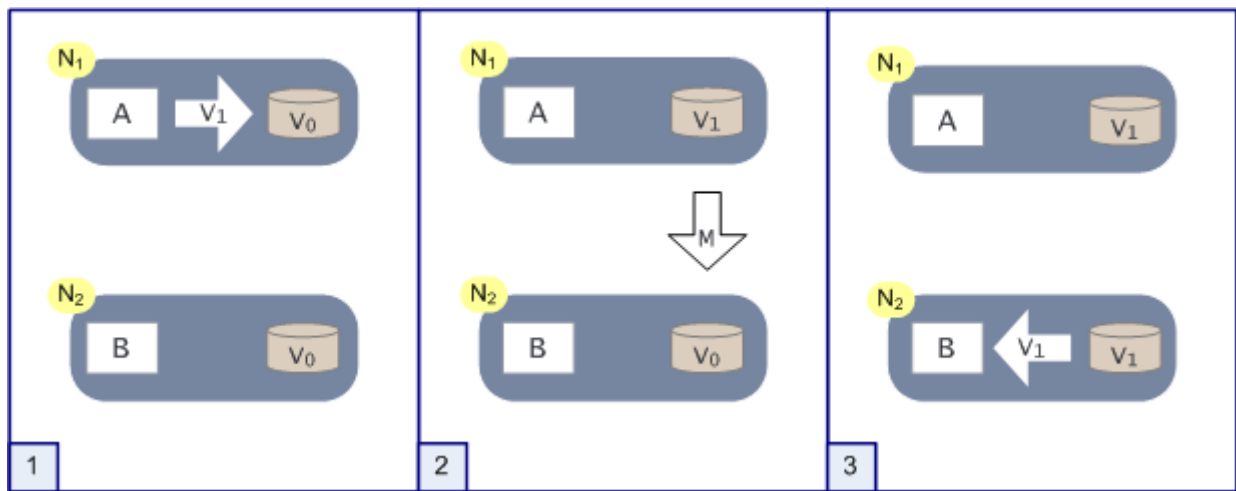
The Proof in Pictures

Here's a simplified proof, in pictures because I find it much easier to understand that way. I've mostly used the same terms as Gilbert and Lynch so that this ties up with their paper.

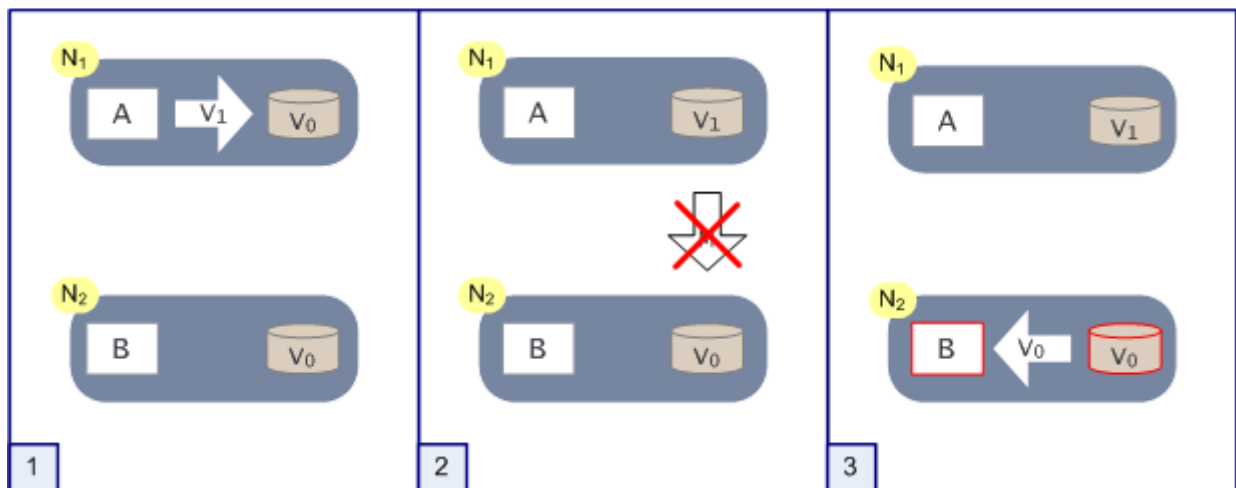


The diagram above shows two nodes in a network, N₁ and N₂. They both share a piece of data V (how many

physical copies of War and Peace are in stock), 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. In this experiment, A writes new values of V and B reads values of V.



In a sunny-day scenario this is what happens: (1) First A writes a new value of V, which we'll call V_1 . (2) Then a message (M) is passed from N_1 to N_2 which updates the copy of V there. (3) Now any read by B of V will return V_1 .



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.

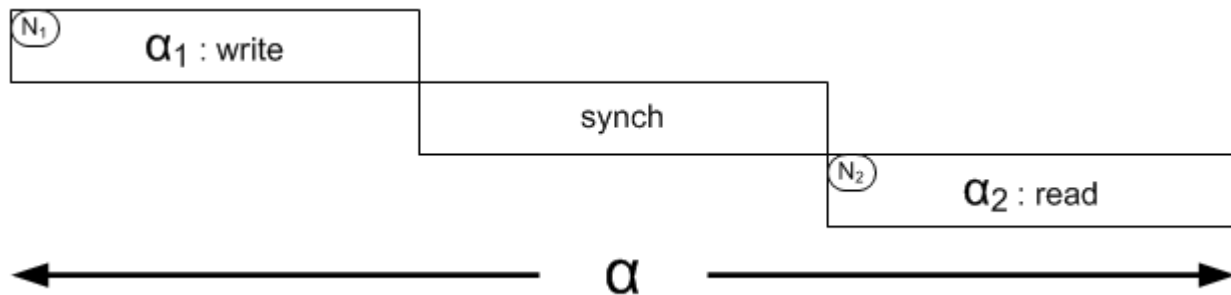
Hopefully that seems fairly obvious. Scale this up to even a few hundred transactions and it becomes a major issue. If M is an asynchronous message then N_1 has no way of knowing whether N_2 gets the message. Even with guaranteed delivery of M, N_1 has no way of knowing if a message is delayed by a partition 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, using a slight variation on this, that even in a partially-synchronous model (with ordered clocks on each node) atomicity cannot be guaranteed.

So what CAP tells us is that if we want A and B to be highly available (i.e. working with minimal latency) and we want our nodes N_1 to N_n (where n 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 (one copy of War and Peace in stock) and other nodes will think that V is V_1 (no copies of War and Peace in stock).

We'd really like everything to be structured, consistent and harmonious, like the music of a [prog rock](#) band from the early seventies, but what we are faced with is a little bit of punk-style anarchy. And actually, whilst it might scare our grandmothers, it's OK once you know this, because both can work together quite happily.

Let's quickly analyse this from a transactional perspective.



If we have a transaction (i.e. unit of work based around the persistent data item V) called α , then α_1 could be the write operation from before and α_2 could be the read. On a local system this would be easily handled by a database with some simple locking, isolating any attempt to read in α_2 until α_1 completes safely. In the distributed model though, with nodes N_1 and N_2 to worry about, the intermediate synchronising message has also to complete. Unless we can control *when* α_2 happens, we can *never* guarantee it will see the same data values α_1 writes. All methods to add control (blocking, isolation, centralised management, etc) will impact either partition tolerance or the availability of α_1 (A) and/or α_2 (B).

Dealing with CAP

You've got a few choices when addressing the issues thrown up by CAP. The obvious ones are:

1. Drop Partition Tolerance

If you want to run without partitions you have to stop them happening. One way to do this is to put everything (related to that transaction) on one machine, or in one atomically-failing unit like a rack. It's not 100% guaranteed because you can still have partial failures, but you're less likely to get partition-like side-effects. There are, of course, significant scaling limits to this.

2. Drop Availability

This is the flip side of the drop-partition-tolerance coin. On encountering a partition event, affected services simply wait until data is consistent and therefore remain unavailable during that time. Controlling this could get fairly complex over many nodes, with re-available nodes needing logic to handle coming back online gracefully.

3. Drop Consistency

Or, as Werner Vogels puts it, accept that things will become "[Eventually Consistent](#)" (updated Dec 2008). Vogels' article is well worth a read. He goes into a lot more detail on operational specifics than I do here.

Lots of inconsistencies don't actually require as much work as you'd think (meaning continuous

consistency is probably not something we need anyway). In my book order example if two orders *are* received for the one book that's in stock, the second just becomes a back-order. As long as the customer is told of this (and remember this is a rare case) everybody's probably happy.

4. The BASE Jump

The notion of accepting eventual consistency is supported via an architectural approach known as BASE (**B**asically **A**vailable, **S**oft-state, **E**ventually consistent). BASE, as its name indicates, is the logical opposite of ACID, though it would be quite wrong to imply that any architecture should (or could) be based wholly on one or the other. This is an important point to remember, given our industry's habit of "oooh shiny" strategy adoption.

And here I defer to Professor Brewer himself who emailed me some comments on this article, saying:

the term "BASE" was first presented in the 1997 SOSP article that you cite. I came up with acronym with my students in their office earlier that year. I agree it is contrived a bit, but so is "ACID" -- much more than people realize, so we figured it was good enough. Jim Gray and I discussed these acronyms and he readily admitted that ACID was a stretch too -- the A and D have high overlap and the C is ill-defined at best. But the pair connotes the idea of a spectrum, which is one of the points of the PODC lecture as you correctly point out.

Dan Pritchett of EBay has a [nice presentation](#) on BASE.

5. Design around it

Guy Pardon, CTO of [atomikos](#) wrote an interesting post which he called "[A CAP Solution \(Proving Brewer Wrong\)](#)", suggesting an architectural approach that would deliver Consistency, Availability and Partition-tolerance, though with some caveats (notably that you don't get all three guaranteed in the same instant).

It's worth a read as Guy eloquently represents an opposing view in this area.



Summary

That you can only guarantee two of Consistency, Availability and Partition Tolerance is real and evidenced by the most successful websites on the planet. If it works for them I see no reason why the same trade-offs shouldn't be considered in everyday design in corporate environments. If the business explicitly doesn't want to scale then fine, simpler solutions are available, but it's a conversation worth having. In any case these discussions will be about appropriate designs for specific operations, not the whole shebang. As Brewer said in his email "the only other thing I would add is that different parts of the same service can choose different points in the spectrum". Sometimes you [absolutely need consistency](#) whatever the scaling cost, because the risk of not having it is too great.

These days I'd go so far as to say that Amazon and EBay don't have a scalability problem. I think they *had* one and now they have the tools to address it. That's why they can freely talk about it. Any scaling they do now (given the size they already are) is really more of the same. Once you've scaled, your problems shift to those of operational maintenance, monitoring, rolling out software updates etc. - tough to solve, certainly, but nice to have when you've got those revenue streams coming in.

References

1. HP's take on CAP Theorem, a white paper entitled "[There is no free lunch with distributed data](#)"

2. Computer Science Notes on [Distributed Transactions](#) and [Network Partitions](#) from University of Sussex
3. Nice [post](#) by Jens Alfke on databases, scaling and Twitter.
4. [Pat Helland](#)'s Microsoft paper on distributed transactions and SOA called [Data on the Outside versus Data on the Inside](#) , which he later related to CAP Theorem [here](#)
5. Another set of Computer Science [course slides](#) , this time from George Mason University in Virginia, on [Distributed Software Systems](#) and particularly CAP Theorem and the clash between ACID and BASE ideologies.
6. The 4th June 1976 is considered to be the birth of Punk Rock in the UK. Thanks to Charlie Dellacona for pointing out that the [Ramones](#) take credit for starting the movement in early 1974 in the US, though their official [punk recordings](#) are more or less contemporaneous.
7. Thanks to [Hiroshi Yuki](#), a [Japanese translation](#) of this article is available.
8. Thanks to [Daniel Cohen](#), a [two part](#) Hebrew translation of this article is available.
9. Thanks to [Wang Qi](#), a [Chinese translation](#) of this article is available.

© Copyright: 2011 Julian Browne. All rights reserved.