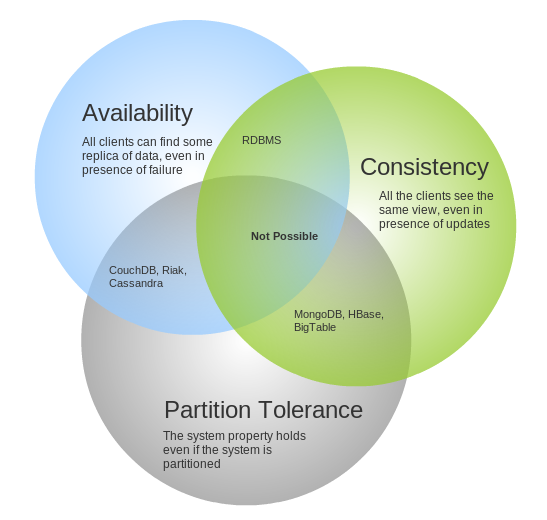
**CAP theorem**, also known as **Brewer's theorem**, states that it is impossible for a distributed computer system to simultaneously provide all three of the following guarantees:

|  |  |
| --- | --- |
| **Consistency** | All the nodes see the same data at the same time.    A read from any node is guaranteed to return all the previously completed writes and the most recent write for a given client |
| **Availability** | A guarantee that every request receives a response about whether it succeeded or failed.    It means that a Read or Write always succeeds. A non-failing node will return a response within a reasonable amount of time (no error or timeout). |
| **Partition Tolerance** | The system will continue to function even when network failures prevent some machines from communicating with others.    It means that the system will continue to work unless there is a total network failure. A few nodes can fail and the system keeps going. |

Brewer originally described a choice of "two out of the three" CAP properties, leaving three viable design options to choose from: CP, AP, and CA.

However, further consideration shows that CA is not really a coherent option because a system that is not Partition-tolerant will, by definition, be forced to give up Consistency or Availability during a partition. Therefore, a more modern interpretation of the theorem is: during a network partition, a distributed system must choose either Consistency or Availability.

One of the "*Fallacy of Distributed Computing*" is that networks are reliable. They aren't. Networks and parts of networks go down frequently and unexpectedly. Network failures *happen to your system* and you don't get to choose when they occur.



 Given that networks aren't completely reliable, you must tolerate partitions in a distributed system, period. Fortunately, though, you get to choose what to do when a partition does occur. According to the CAP theorem, this means we are left with two options: Consistency and Availability.

* **CP** - Consistency/Partition Tolerance - Wait for a response from the partitioned node which could result in a timeout error. The system can also choose to return an error, depending on the scenario you desire. *Choose Consistency over Availability when your business requirements dictate atomic reads and writes*.
* **AP** - Availability/Partition Tolerance - Return the most recent version of the data you have, which could be stale. This system state will also accept writes that can be processed later when the partition is resolved. *Choose Availability over Consistency when your business requirements allow for some flexibility around when the data in the system synchronizes*. Availability is also a compelling option when the system needs to continue to function in spite of external errors (shopping carts, etc.)

The decision between Consistency and Availability is a *software trade off*. You can choose what to do in the face of a network partition - the control is in your hands. Network outages, both temporary and permanent, are a fact of life and occur whether you want them to or not - this exists outside of your software.

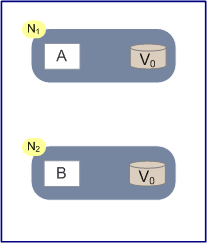
**BASE**

It stands for **B**asic **A**vailability, **S**oft State & **E**ventual Consistency. A BASE system gives up on consistency.

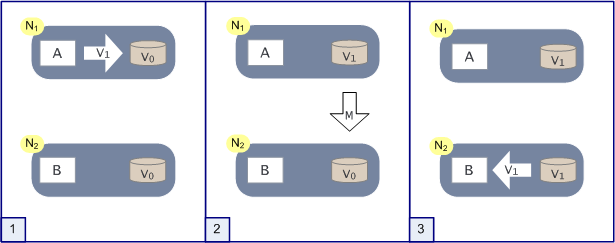
* **Basically available** indicates that the system *does* guarantee availability, in terms of the CAP theorem.

* **Soft state** indicates that the state of the system may change over time, even without input. This is because of the eventual consistency model.

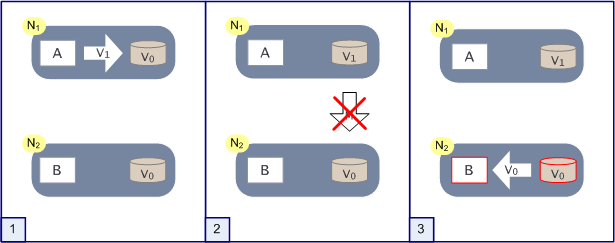
* **Eventual consistency** indicates that the system will become consistent over time, given that the system doesn't receive input during that time.



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 V0. Running on N1 is an algorithm called A which we can consider to be safe, bug free, predictable and reliable. Running on N2 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 V1. (2) Then a message (M) is passed from N1 to N2 which updates the copy of V there. (3) Now any read by B of V will return V1.

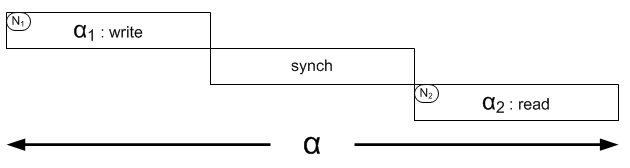


If the network partitions (that is messages from N1 to N2 are not delivered) then N2 contains an inconsistent value of V when step (3) occurs.

Hopefully that seems fairly obvious. Scale this is up to even a few hundred transactions and it becomes a major issue. If M is an asynchronous message then N1 has no way of knowing whether N2 gets the message. Even with guaranteed delivery of M, N1 has no way of knowing if a message is delayed by a partition event or something failing in N2. Making M synchronous doesn’t help because that treats the write by A on N1 and the update event from N1 to N2 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 N1 to Nn (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 V0 (one copy of War and Peace in stock) and other nodes will think that V is V1 (no copies of War and Peace in stock).

Let’s quickly analyze 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 α1 could be the write operation from before and α2 could be the read. On a local system this would easily be handled by a database with some simple locking, isolating any attempt to read in α2 until α1 completes safely. In the distributed model though, with nodes N1 and N2 to worry about, the intermediate synchronizing message has also to complete. Unless we can control *when* α2happens, we can *never* guarantee it will see the same data values α1 writes. All methods to add control (blocking, isolation, centralized management, etc) will impact either partition tolerance or the availability of α1 (A) and/or α2 (B).

**Dealing with CAP**

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

* **Drop Partition Tolerance**Choose this 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.
* **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.
* **Drop Consistency**Or, as Werner Vogels puts it, accept that things will become “[Eventually Consistent](http://www.allthingsdistributed.com/2008/12/eventually_consistent.html)”. 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.

**ACID**

* **Atomicity** states that database modifications must follow ‘all or nothing’ rule. Each transaction is said to be atomic. If one part of the transaction fails, the entire transaction fails. It is critical that the database management system maintain the atomic nature of transactions in spite of any DBMS, operating system or hardware failure.
* **Consistency** states that only valid data will be written to the database. If, for some reason, a transaction is executed that violates the database’s consistency rules, the entire transaction will be rolled back and the database will be restored to a state consistent with those rules. On the other hand, if a transaction successfully executes, it will take the database from one state that is consistent with the rules to another state that is also consistent with the rules.
* **Isolation** requires that multiple transactions occurring at the same time not impact each other’s execution. For example, if Joe issues a transaction against a database at the same time that Mary issues a different transaction; both transactions should operate on the database in an isolated manner. The database should either perform Joe’s entire transaction before executing Mary’s or vice-versa. This prevents Joe’s transaction from reading intermediate data produced as a side effect of part of Mary’s transaction that will not eventually be committed to the database. Note that the isolation property does not ensure which transaction will execute first, merely that they will not interfere with each other.
* **Durability** ensures that any transaction committed to the database will not be lost. Durability is ensured through the use of database backups and transaction logs that facilitate the restoration of committed transactions in spite of any subsequent software or hardware failures.