The CAP Theorem

A database *can* provide strong consistency *and* system availability during network partitions. The common belief that this combination is impossible is based on a misunderstanding of the CAP theorem.

What is the CAP Theorem?

In 2000, Eric Brewer conjectured that a distributed system cannot simultaneously provide all three of the following desirable properties:

- **C**onsistency: A read sees all previously completed writes.
- **A**vailability: Reads and writes always succeed.
- **P**artition tolerance: Guaranteed properties are maintained even when network failures prevent some machines from communicating with others.

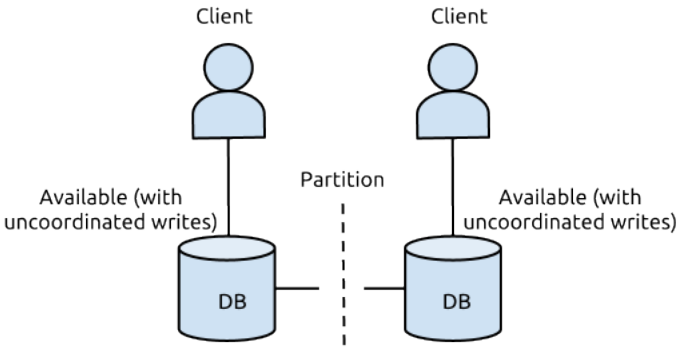
In 2002, Gilbert and Lynch proved this in the asynchronous and partially synchronous network models, so it is now commonly called the CAP Theorem (http://en.wikipedia.org/wiki/CAP_theorem) .

Brewer originally described this impossibility result as forcing a choice of "two out of the three" **CAP** properties, leaving three viable design options: **CP**, **AP**, and **CA**. However, further consideration shows that **CA** is not really a coherent option because a system that is not **P**artition-tolerant will, by definition, be forced to give up **C**onsistency or **A**vailability during a partition. Therefore, a more modern interpretation (http://dbmsmusings.blogspot.com/2010/04/problems-with-cap-and-yahoos-little.html) of the theorem is: *during a network partition, a distributed system must choose either **C**onsistency or **A**vailability.*

What does choosing Availability mean?

Let's consider an **AP** database. In such a database, reads and writes would always succeed, even when network connectivity is unavailable between nodes. If possible, these would certainly seem like desirable properties!

However, the downside is stark. Imagine a simple distributed database consisting of two nodes and a network partition making them unable communicate. To be **A**vailable, each of the two nodes must continue to accept writes from clients.



Data divergence in an AP system during partition

Of course, because the partition makes communication impossible, a write on one node cannot be seen by the other. Such a system is now "a database" in name only. As long as the partition lasts, the system is fully equivalent to two independent databases whose contents need not even be related, much less consistent.

Where's the confusion?

Confusion about the CAP theorem usually involves the interpretation of the **A**vailability property. **A**vailability in the **CAP** sense means that *all* nodes remain able to read and write even when partitioned. A system that keeps some, but not all, of its nodes able to read and write is not **A**vailable in the **CAP** sense, *even if it remains available to clients* and satisfies its SLAs for high availability (http://en.wikipedia.org/wiki/High_availability) .

What does FoundationDB choose?

As any ACID database must, during a network partition FoundationDB chooses **C**onsistency over **A**vailability. This does *not* mean that the database becomes unavailable for clients. When multiple machines or datacenters hosting a FoundationDB database are unable to communicate, *some* of them will be unable to

execute writes. In a wide variety of real-world cases, the database and the application using it will remain up. A network partition affecting some machines is no worse than a failure of those same machines, which FoundationDB handles gracefully due to its fault tolerant design.

FoundationDB fault tolerance

FoundationDB's design goal is to make sure that, even if some machines are down or unable to communicate reliably with the network, the database and the application connected to it remain up. This is high availability as usually understood, but it is *not* **A**vailability in the **CAP** sense because the database will be unavailable *on the affected machines*.

FoundationDB seeks to present user applications with a single (logical) database. The challenge of handling a partition is to determine which machines should continue to accept reads and writes. To make this determination, FoundationDB is configured with set of *coordination servers*. FoundationDB selects the partition in which a majority of these servers are available as the one that will remain responsive. If failures are so pervasive that there is *no* such partition, then the database really will be unavailable.

The coordination servers use the Paxos (http://en.wikipedia.org/wiki/Paxos_(computer_science)) algorithm to maintain a small amount of shared state that itself is **C**onsistent and **P**artition-tolerant. Like the database as a whole, the shared state is not **A**vailable but *is* available for reads and writes in the partition containing a majority of the coordination servers.

FoundationDB uses this shared state to maintain and update a replication topology. When a failure occurs, the coordination servers are used to change the replication topology. It's worth noting that the coordination servers aren't involved at all in committing transactions.

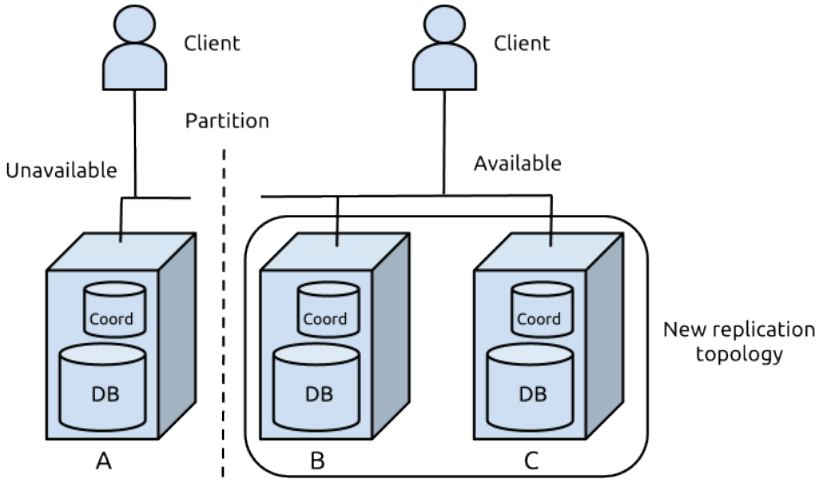
Example: a minimal configuration

To illustrate how the coordination servers support fault tolerance, let's look at a FoundationDB cluster of the minimal size that allows for data replication. Of course, the fault tolerance and availability provided by coordination are higher when the cluster is larger.

Imagine a small web startup that wants its application, served by FoundationDB within a datacenter, to stay available even if a machine fails. It sets up a cluster of three machines—A, B, and C—each running a database server and a coordination server. Applying the majority rule to this cluster, any pair of machines that can communicate will remain available. The startup configures FoundationDB in its `double` redundancy mode, in which the system will make two copies of each piece of data, each on a different machine.

Imagine that a rack-top switch fails, and A is partitioned from the network. A will be unable to commit new transactions because FoundationDB requires an acknowledgment from B or C. The database server on A can only communicate with the coordination server on A, so it will not be able to achieve a majority to set up a new replication topology. For any client communicating only with A, the database is down.

However, for all other clients, the database servers can reach a majority of coordination servers, B and C. The replication configuration has ensured there is a full copy of the data available even without A. For these clients, the database will remain available for reads and writes and the web servers will continue to serve traffic.



Maintenance of availability during partition

When the partition ends, A will again be able to communicate with the majority of coordination servers and will rejoin the database. Depending on how long the communications failure lasted, A will rejoin by either receiving transactions that occurred in its absence or, in the worst case, transferring the contents of the database. After A has rejoined the database, all machines will again be able to handle transactions in a fault tolerant manner.

In contrast to the minimal cluster above, an actual production system would typically be configured in

`triple` redundancy mode on five or more machines, giving it correspondingly higher availability. For further details, read our discussion of fault tolerance (/white-papers/fault-tolerance/).
