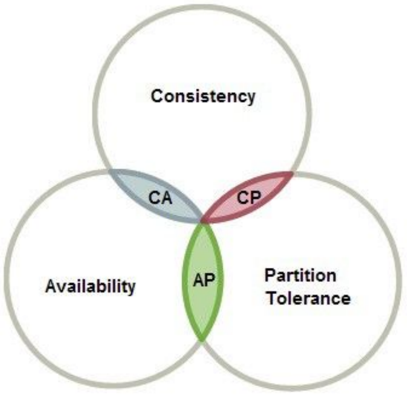
An Illustrated Proof of the CAP Theorem.

The CAP Theorem is a fundamental theorem in distributed systems that states any distributed system can have at most two of the following three properties:

- Consistency
- Availability
- Partition tolerance.

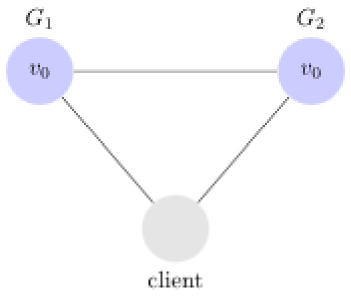
In the past, when we wanted to store more data or increase our processing power, the common option was to scale vertically (get more powerful machines) or further optimize the existing code base. However, with the advances in parallel processing and distributed systems, it is more common to expand horizontally, or have more machines to do the same task in parallel. We can already see a bunch of data manipulation tools in the Apache project like Spark, Hadoop, Kafka, Zookeeper and Storm. However, in order to efficiently pick the tool of choice, a basic idea of CAP theorem is necessary. CAP Theorem is a concept that a distributed database system can only have 2 of the 3: **Consistency, Availability** and **Partition Tolerance.**



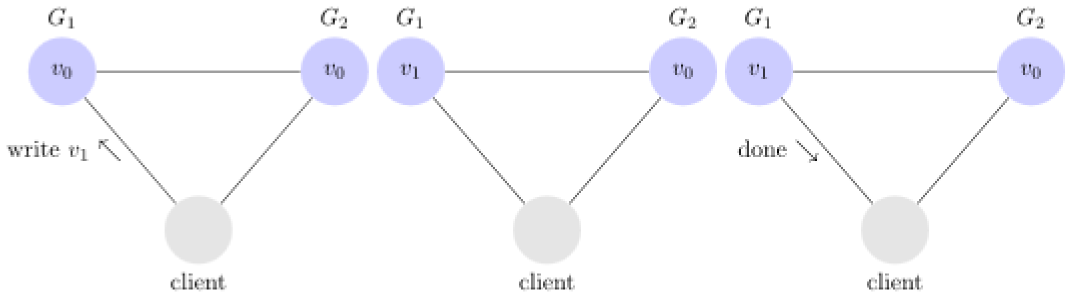
CAP Theorem is very important in the Big Data world, especially when we need to make trade off's between the three, based on our unique use case. On this blog, I will try to explain each of these concepts and the reasons for the trade off.

A Distributed System

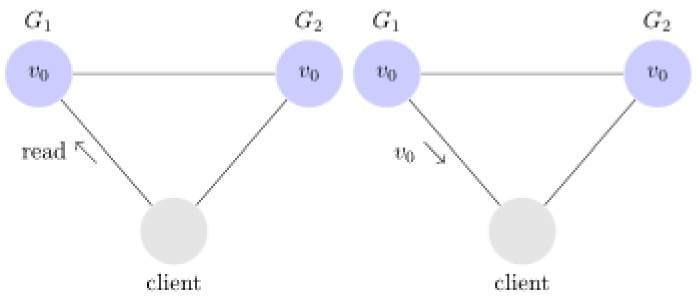
Let's consider a very simple distributed system. Our system is composed of two servers, G_1 and G_2. Both of these servers are keeping track of the same variable, v, whose value is initially v_0. G_1 and G_2 can communicate with each other and can also communicate with external clients. Here's what our system looks like:



A client can request to write and read from any server. When a server receives a request, it performs any computations it wants and then responds to the client. For example, here is what a write looks like.



And here is what a read looks like.



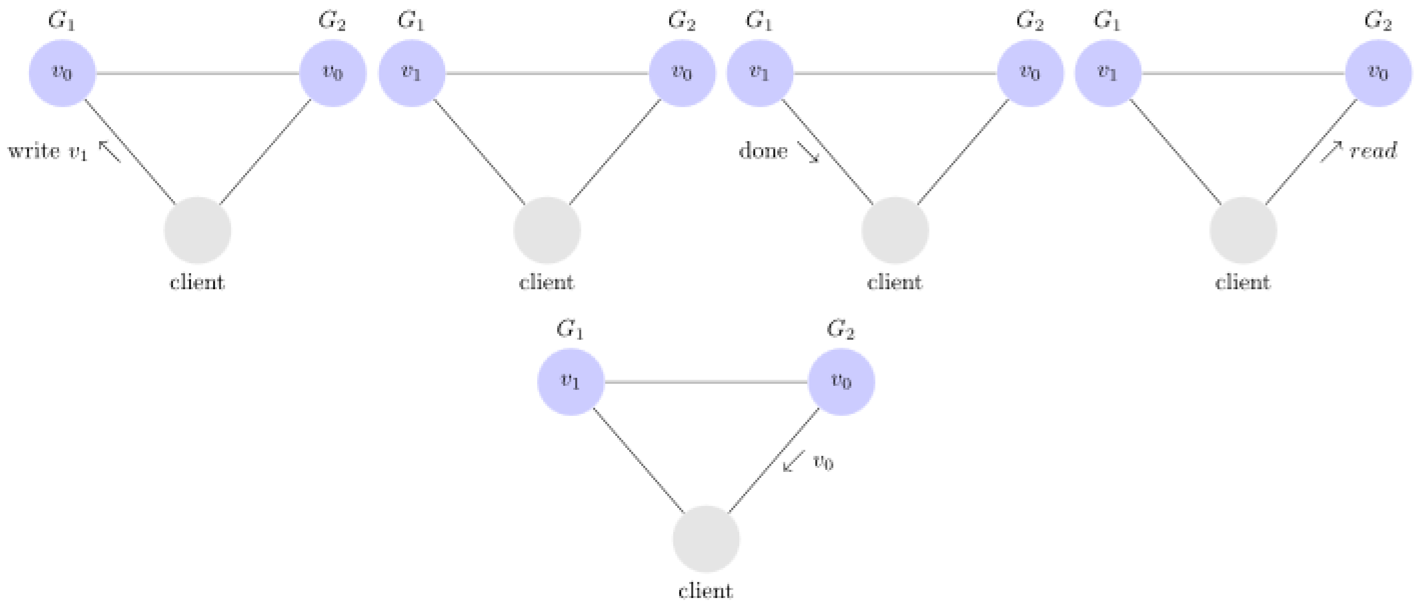
Now that we've gotten our system established, let's go over what it means for the system to be consistent, available, and partition tolerant.

Consistency
Here's how Gilbert and Lynch describe consistency.

Any read operation that begins after a write operation completes must return that value, or the result of a later write operations.

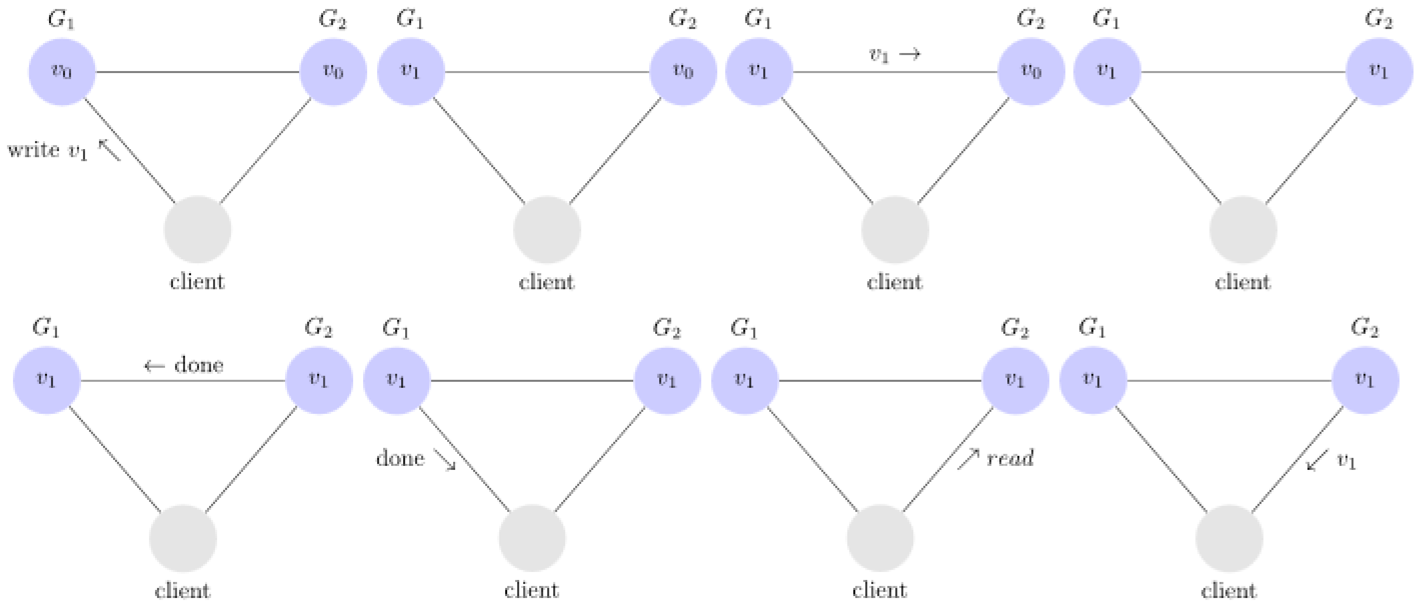
In a consistent system, once a client writes a value to any server and gets a response, it expects to get that value (or a fresher value) back from any server it reads from.

Here is an example of ***inconsistent*** system.



Our client writes $\boldsymbol{v_1}$ to $\boldsymbol{G_1}$ and $\boldsymbol{G_1}$ acknowledges, but when it reads from $\boldsymbol{G_2}$ it gets stale date: $\boldsymbol{v_0}$.

On the other hand, here is an example of a ***consistent*** system.



In this system, G_1 replicates its value to G_2 before sending an acknowledgement to the client. Thus, when the client reads from G_2, it gets the most up to date value of v: v_1.

Availability
Here's how Gilbert and Lynch describe availability.

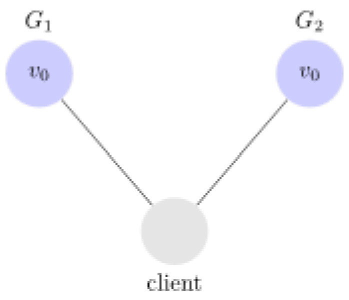
Every request received by a non-railing node in the system must result in a response.

In an available system, if our client sends a request to a server and the server has not crashed, then the server must eventually respond to the client. The server is not allowed to ignore the client's requests.

Partition Tolerance
Here's how Gilbert and Lynch describe partitions.

> The network will be allowed to lose arbitrarily many messages sent from one node to another.

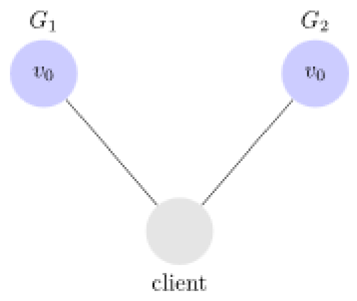
This means that any message ***G_1*** and ***G_2*** send to one another can be dropped. If all the messages were being dropped, then our system would look like this.



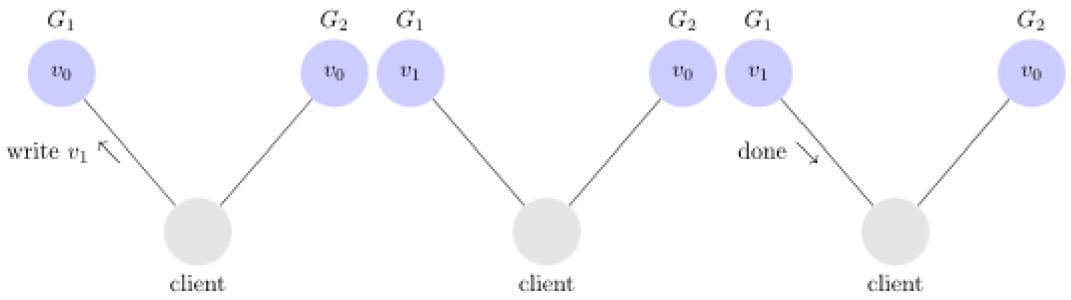
Our system has to be able to function correctly despite arbitrary network partitions in order to be partition tolerant.

The Proof
Now that we've acquainted ourselves with the notion of consistency, availability, and partition tolerance, we can prove that a system cannot simultaneously have all three.

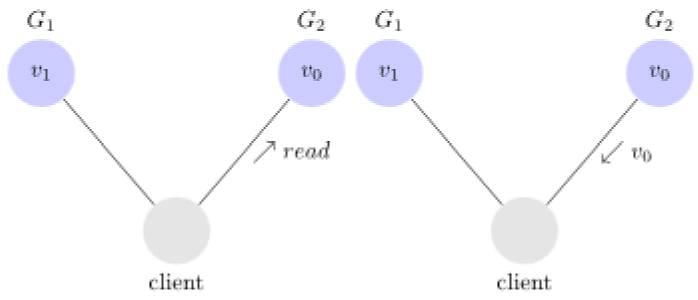
Assume for contradiction that there does exist a system that is consistent, available, and partition tolerant. The first thing we do is partition our system. It looks like this.



Next, we have our client request that ***v*₁** be written to ***G*₁.** Since our system is available, ***G₁*** must respond. Since the network is partitioned, however, ***G₁*** cannot replicate its data to ***G₂*.** Gilbert and Lynch call this phase of execution α_1.



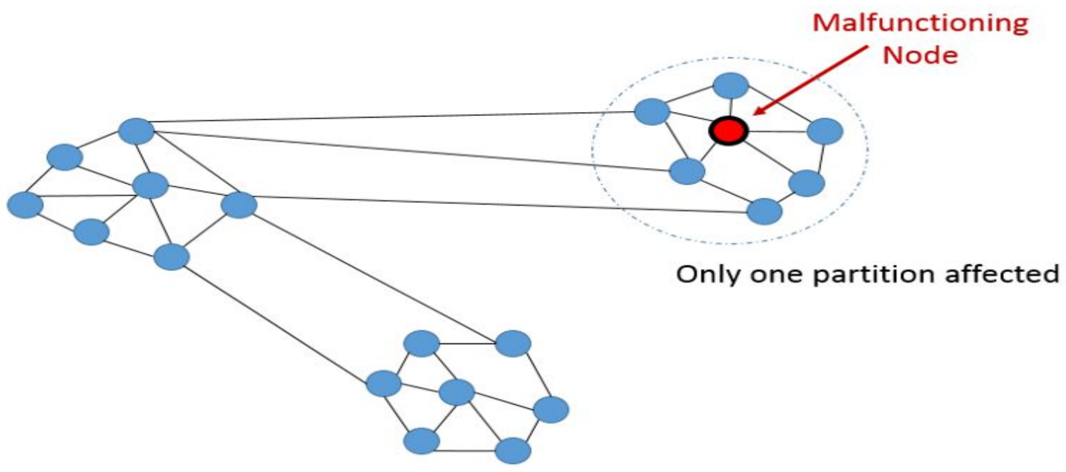
Next, we have our client issue a read request to ***G₂***. Again, since our system is available, ***G₂*** must respond. And since the network is partitioned, ***G₂*** cannot update its value from ***G₁***. It returns ***v₀***. Gilbert and Lynch call this phase of execution $\alpha 2$.



G₂ returns ***v₀*** to our client after the client had already written ***v₂*** to ***G₁***. This is inconsistent.

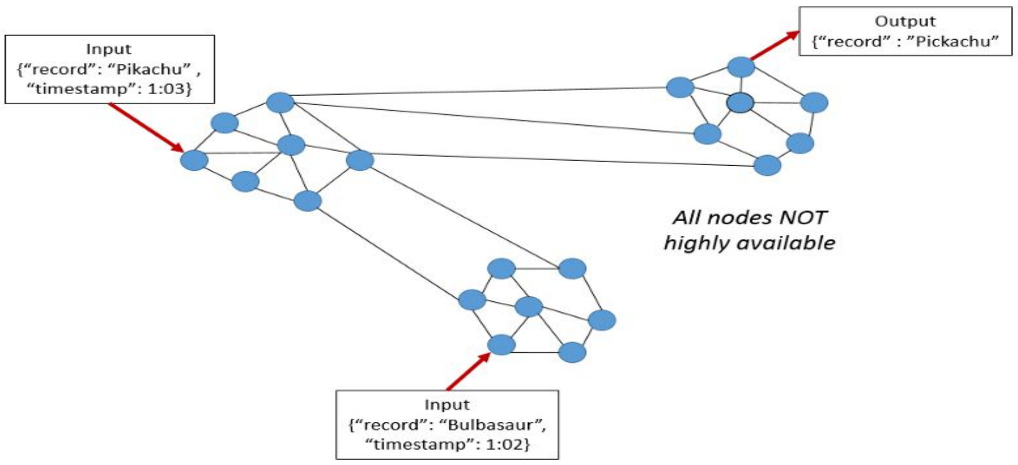
We assumed a consistent, available, partition tolerant system existed, but we just shoed that there exists an execution for any such system in which the system acts inconsistently. Thus, no such system exists.

Partition Tolerance



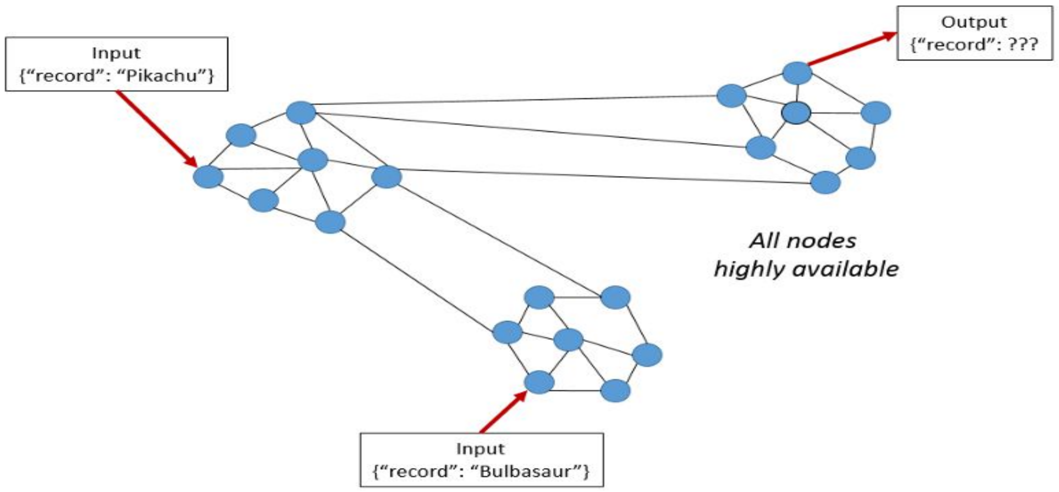
This condition states that the system continues to run, despite the number of messages being delayed by the network between nodes. A system that is partition-tolerant can sustain any amount of network failure that doesn't result in a failure of the entire network. Data records are sufficiently replied across combinations of nodes and networks to keep the system up through intermittent outages. When dealing with modern distributed systems, Partition Tolerance is not an option. It''s a necessity. Hence, we have to trade between Consistency and Availability.

High Consistency



This condition states that all data see the same data at the same time. Simply put, performing a *read* operation will return the value of the most recent *write* operation causing all nodes to return the same data. A system has consistency if a transaction starts with the system in a consistent state, and ends with the system in a consistent state. In this model, a system can (and does) shift into an inconsistent state during a transaction, but the entire transaction gets rolled back if there is an error during a transaction, but the entire transaction gets rolled back if there is an error during any stage in the process. In the image, we have 2 different records ("Bulbasaur" and "Pickachu") at different timestamps. The output on the third partition is "Pikachu", the latest input. However, the nodes will need time to update and will not be Available on the network as often.

High Availability



This condition states that every request gets a response on success/failure. Achieving availability in a distributed system requires that the system remains operational 100% of the time. Every client gets a response, regardless of the state of any individual node in the system. This metric is trivial to measure: either you can submit read/write commands, or you cannot. Hence, the databases are time independent as the nodes need to be available online at all times. This means that, unlike the previous example, we do not know if "Pikachu" or "Bulbasaur" was added first. The output could be either one. Hence why, high availability isn't feasible when analyzing streaming data at high frequency.