medium.com

Demystifying CAP theorem, Eventual Consistency and exactly once delivery guarantee

Márton Waszlavik

8-9 minutes

Many blog posts and articles exist today in these topics but I feel most of them is too complicated, not straight to the point and well, in some cases, they are inaccurate and misleading. So I decided to write my own.

CAP theorem

Long story short: it says you can have at most two out of Consistency (C), Availability (A) and Partition Tolerance (P) in a distributed environment. “Theorem”, by the way quite misleading as it has been actually proven since it first published a decade ago. I think the 3 main terms C, A and P also requires some clarification:

- **Consistency** (C) means a strong consistency. For example, if a server receives a data change request with strong consistency, that means that all upcoming requests to any server in the cluster shall see the same change immediately after that.
- **Availability** (A) means that the system is responsive. For example, when a client sends a data change request to a server in the cluster, a server will not refuse to persist the change — ie. it cannot reply “Sorry I cannot accept your change now”. To be clear, Availability does not mean that a specific server MUST be available — individual servers can fail but servers that are running shall be able to deal with the request.
- **Partition Tolerance** (P) means that certain sequence of communication events can be lost between the parties. For example, a server in a cluster might be temporarily unavailable while it is being restarted.

Dealing with it

Some misleading statements I often came across when reading about the topic:

1. “*You can pick at most 2 of C, A and P that can be satisfied in your system at the same time*” — This is true in theory. But practically it is quite misleading because **C and A can be chosen, but P is a fact**. Why? In the real life, we cannot design hardware and software that will never lose any communication in a distributed environment. Networks can go down, hardware failures happen, software needs to be upgraded and restarted. In other words, Partition Tolerance

(P) is a fact we cannot choose as it is there already. With that in mind, we can have CP or AP systems.

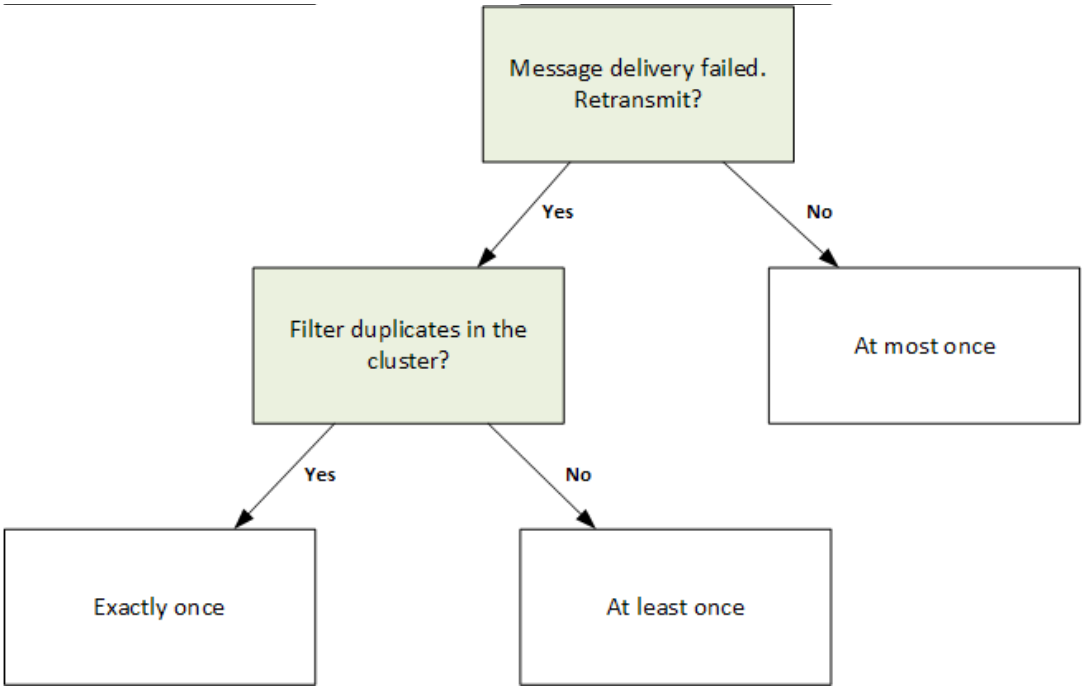
2. "*XY technology can offer either CP or AP*": I often see statements that some technology like Cassandra or MySQL is either CP or AP (like here, which is a good article otherwise). But that is not quite true. Mainly because most of these systems can be configured to behave either like CP or AP. Furthermore, this is really a dynamic behavior and not a static, baked-in feature in most systems. For example, a system may treat one request as CP while another one as AP at the exact same time. Imagine a service that has 2 requests at the same time when 1 server is down in a 2 server cluster. One results in a money transfer and the system may reply with a message that it cannot complete the request right now. The other request is about to like something on the products page, in which case the server persists the action and replies with success. In the first case, we sacrifice availability because we want to make sure the transaction is persisted by both servers, and we are not risking to loose the money transfer in case our only active server dies. This is a CP behavior. In the other case, we do take the (fairly low) risk that the customer's like on a product page disappears when our only remaining server dies before it could get synced to the second one. Here we sacrifice strong consistency in the favor of Availability. This means AP. Similarly, we can build a MySQL cluster with master-save replication which provides AP, or we can build a Galera cluster which is CP. **After all, C, A and P are dynamic behaviors and in most cases, not bound to one technology or another.**

3. "*AP = eventually consistent system*". First things first, Eventually Consistent means that a certain change or update may not become immediately visible on all servers of the cluster at the same time. For example, it can happen that you get a message on Facebook, you read it, then it suddenly disappears. After refreshing the page a

couple of times, finally, it reappears again. This is Eventual Consistency. It is the highest level of consistency (but still less than strong consistency, `C`) that can be achieved by an AP behavior. It is commonly used where strong consistency is not crucial — see some examples above. However, it does not mean that an AP system necessarily provides Eventual Consistency. It is still possible that after a node failure and replacement, the data conflict remains unresolved — in other words, it will not become consistent at all. This may be intentional (ie. consistency is sacrificed in the favor of performance), or unintentional but not auto-recoverable. This is also called *split-brain*.

Implementations and performance considerations

Imagine the following scenario: A Client sends a Message to a Server in a Cluster, but the request runs into timeout: The Message is sent but the Client did not receive anything back from the Server. What can potentially happen to fix this issue?



Design decisions on delivery guarantees

Before considering our options, it worth to highlight that the real pain point is that we cannot be entirely sure whether (A) the server

received and processed the Message but the reply got lost, or (B) the Message did not make it to the server at all. For more details, see the Two Generals' Problem.

When designing a distributed system, we can choose to ignore this problem in the Client — or, hold the Message and try to re-transmit it again to the cluster. The first choice means that the Message is either received once (option A above), or not received (option B). This can be called *at most once* delivery guarantee. In the other case, when the Client may resend the Message a couple of times until it gets confirmation from the server, the Message is either received once (option B) or multiple times (option A). This is also called as *at least once* delivery guarantee.

Sometimes it is ok to have *at least once* or *at most once* deliveries. But neither of them would be good enough when we wanted to transmit a money transfer like send $100 to X.Y., right? In those cases, and in many other practical cases, we need *exactly once* delivery guarantee.

As a side note: *at most once* guarantee is a trivial case, but anything else requires significant efforts in the implementation and have performance indications as well. Clients need to deal with retransmissions, ordering of messages, temporary message buffers etc. — this can add a lot of complexity.

Coming back to *exactly once* guarantee, we can start using a unique message id (UID) generated by the Client so we can filter duplicates on the server side as an enhancement of our previous *at most once* scenario. It is not enough to do this on a per server basis, we need to do this globally in the whole cluster, since the Message originally sent to Server #1 may have been resent to Server #2 by the Client after Server #1 became unavailable, which means that both servers have a copy that needs to be deduplicated. In other words: Server #1 and Server #2 needs to be

in sync, so we need Consistency (C). Without being completely in sync, the two servers could have a copy of the same Message so it could be potentially delivered to a Client twice.

To sum it up, *exactly once* guarantee in a distributed environment requires strong Consistency in the system. This means that such system cannot provide Availability, so it may return an error to the Client that is cannot process the Message right now, and it should be retransmitted later.
Choosing an eventually consistent way to filter duplicates, we could preserve Availability — but in this case, we have to accept the fact that during system failures consumers would occasionally receive duplicated Messages violating the *exactly once* attribute.