Most of IT professionals must have heard about CAP theorem that Eric Brewer made public in 1999. Since then it became a de facto standard for distributed application architecture. Recently I came across quite an interesting article by British-born physicist and computer scientist Mark Burgess that revisits CAP theorem, and claims that theory it is not, and offers some interesting alternatives. Reading Burgess' article led me to read the pile of other related papers that shed some light on contemporary CAP theorem meaning.

Mr Burgess argues that CAP theorem is simply wrong. He starts with the fact that as no mathematical proof exists (a detailed explanation by Setch Gilbert and Nancy Lynch exists) it is not a real theorem but rather a conjecture. He uses relativity theory to prove that CAP theorem's ignorance of time makes it invalid. Finally, instead of sticking to CAP he offers another approach, Promise Theory, that emphasises each agent, or node, autonomy and responsibility for its own actions only.

The article by Gilbert and Lynch that I mentioned above is an attempt to provide more precise frame for CAP theorem. The authors provide much more detailed definitions for the three components as well as take time and networking into consideration. The article provides a lot of insight on the subject and defines multiple terms thoroughly but does lack mathematical proof indeed. Burgess argues that lack of this very proof is one of the weaknesses of CAP.

According to Einstein's special relativity theory two events that happen in different places can't happen simultaneously in absolute terms. It depends on an observer's location and orientation whether they are simultaneous or one of them precedes another. If we apply relativity of simultaneity to CAP theorem then we will immediately realise that in a distributed system (as components of such a system are located across the space by design) we are not able to talk about simultaneity and therefore consistency in absolute terms.

From this perspective Burgess tries to define three components of CAP more strictly. First of all he attempts to add time dimension to all three definitions. Availability becomes ability to promise that every request will be responded in some predefined amount of time – and this is the only simple definition. Consistency, as it depends on other agents keeping their promises, is not something that a single agent can promise lightly. Burgess argues that only eventual consistency can be promised as it is impossible to ensure immediate consistency in a running system as we need time to equilibrate the environment.

This is partition tolerance that seems to be the most vague, however. It is possible to define it but the definition (a promise to deliver correct response within predefined amount of time, even if the response depends on information from on agent in different partition) seems to be impractical and would require us to perform quite a lot of work to prepare a response that would be correct for this very partition of the system only.

CAP theorem seen from this very perspective seems to be broken indeed as it is really difficult to define clear boundaries for its components. Burgess points out that the theorem ignores an end user completely and claims that this very factor can contribute a lot to its irrelevance nowadays. To make it easier to understand Burgess refers to git and concept of continuous deployment. When you use git you will probably create number of inconsistent environments, branches, that can (but does not have to) be merged together at some point. When continuous deployment is added to the equation then you can see your changes to be merged gradually (from developer's local environment, via staging, to production). Merging leads to reaching system equilibrium again but is a process that takes time and should not happen on every single change that introduce inconsistency – try to imagine yourself that you deploy every single commit from a feature branch to production; application would became unstable very soon.

If we stick to git analogy then we will realise that process of reaching equilibrium should be initiated from within rather then from the outer space. When a branch exists in the repository then it is being kept up-to-date by pulling external changes and merging them rather then by receiving a message from some central entity that oversees system's consistency. It is up to an agent (a developer in this case) to decide when it needs to make sure that its knowledge is consistent with the knowledge of other agents; some changes might be simply irrelevant or can be integrated at later stage.

Burgess argues that a push-based system (the one with central omnipotent entity responsible for maintaining consistency) can't even tell whether system is consistent or not. On the contrary – pull-based system can judge it's own consistency on relativity basis as every single agent is capable of maintaining its own integrity and is the only party that can judge whether it is consistent or not. This very thought seems to be the key conclusion of Burgess' article.

If we stick to this principle when designing distributed systems then we can gain quite a lot. Instead of creating centralised applications that are prone to failure by relying on connectivity to some central agent or cluster of agents we should rather make every single agent as independent as possible and let it to equilibrate by communicating with its own peers. Even if the system might be inconsistent for some time then it will reach equilibrium eventually while not sacrificing availability.

If you were interested in more details on Burgess ideas start with his article and take a look at the list of other articles that he provides at the bottom. I would recommend Brewer's article on CAP theorem written twelve years after its original publication and Abadi's alternative or extension to CAP – PACELC.