CAP THEOREM

A Guide to Understanding the Cap Theorem

Zeynep Iyigorur
Informatics
Technical University of Munich
Munich, Germany
ga53keq@mytum.de

Iremnur Kidil
Informatics
Technical University of Munich
Munich, Germany
ge34hoz@mytum.de

ABSTRACT

This report is about the Cap Theorem, also known as Brewer's Theorem, named after its founder, Eric Brewer. The Cap Theorem mainly centers around the idea that a distributed data system can have at most two of the three desired properties: consistency, availability and partition tolerance. This idea is called the "2 of 3 principle" and is discussed further in this report. Consequently, the theorem was named after these properties; CAP is an acronym that stands for consistency, availability and partition tolerance.

This report starts with the origin of the Cap Theorem, continues with the clarification of the above-mentioned properties concerning a distributed data system: consistency, availability and partition tolerance, as well as their applications.

Following that, this report aims to clarify the formal proof of the theorem. It is then proceeded with the re-evaluation of the initial theorem. The focus of this part is an article called "Cap Twelve Years Later, How the 'Rules' Have Changed" by Eric Brewer, published in 2012. This article by Brewer is examined precisely in this report, since it is of great importance in terms of perfecting the Cap Theorem. The two main points of Brewer's article that are mentioned in this report are firstly, the concept of trade-offs replacing the "2 of 3" principle and secondly, the deeper understanding of the partitions.

In the final paragraph, the fundamentals of the Cap Theorem are summarized once more, along with the core ideas that are heavily emphasized during the report, mainly concerning what to take away from this report.

CCS CONCEPTS

Data Storing and managing data Data systems Distributed data systems Properties of distributed data systems the Cap theorem

KEYWORDS

CAP Theorem, consistency, availability, partition tolerance, distributed data system, trade-offs, partition, partition recovery, partition mode, partition detection, client, server, node, network

1 The Origin of the Cap Theorem

In the first part of the report, the motivation behind the theorem, as well as the conditions that have led to its discovery are discussed.

In mid-1990's Eric Brewer and his colleagues were developing cluster-based wide-area systems. While working on these systems, Brewer had the opportunity to observe the behavior of distributed data systems in the presence of partitions, thus came up with the "2 of 3" principle. A distributed data system can have at most two of the following three properties: Consistency, Availability and Partition Tolerance [1].

The Cap theorem was first published in 1999, and became publicly known in 2000, during a symposium called "Symposium on Principles of Distributed Computing".

2 Properties of a Distributed Data System: Consistency, Availability and Partition Tolerance

CAP is an acronym that stands for consistency, availability and partition tolerance. Therefore, it is of great importance that this report clarifies each of these properties concerning a distributed data system separately and in-depth.

2.1 Consistency

'C' in CAP referring to the property consistency, ensures for a system to maintain its up-to-dateness. The goal of a consistent distributed system is to respond to the client with the value from the last committed write operation. That being the case, whenever a client wants to read from any server, the response has to contain the most recent state of that value. Thus, the key of consistency is communication between the nodes; once a client writes something to a node, the changes have to be replicated to all the other nodes to preserve consistency in the system.

In the Figure 1 below, there are two systems, one of which demonstrates a consistent system, whereas the other an inconsistent system. Firstly, the consistent system is examined. There is a network consisting of two nodes, namely Node A and Node B, and the initial value of x is 2. First, the client executes a write operation by changing the value of x from 2 to 5. Before Node A acknowledges, it replicates its value to Node B. After the replication, when the client reads from Node B, the response contains the latest version of x, which is 5. On the other hand, in an inconsistent system, the client again writes to Node A and changes the value of x from 2 to 5; Node A acknowledges this change, but does not replicate its value to Node B. Therefore reading back from Node B, the response contains stale data, the older version of x, making the nodes inconsistent.

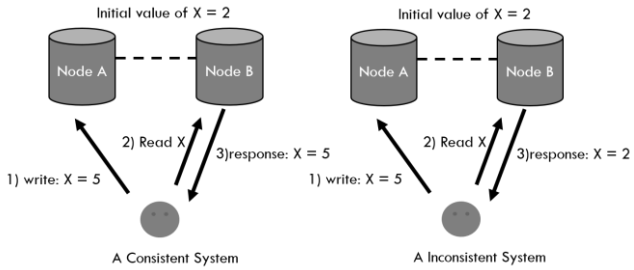


Figure 1: **Consistent vs. Inconsistent System**

2.2 Availability

Being available for a distributed system means that if a client wants to read from any server of the system, a response has to be returned to the client, from the server. If no response is returned, one concludes the system is unavailable, possibly due to a crashed node, or due to any other sort of partition. That being the case, every algorithm must eventually terminate. (Brewer originally only required almost all requests to receive a response. As allowing probabilistic availability does not change the result when arbitrary failures occur, for simplicity we are requiring 100% availability [4].)bu cumleyi daha kendi cumlemismiz gibi tamamlayalim.

In the Figure 2, the first model represents an available system. The client wants to read from Node A and as a response the value of x is returned. On the other hand, in an unavailable system, when the client requests to read from Node B, Node B does not respond. Here, one may conclude that the reason why the system is not available is a partition. Clearly, in the presence of a partition, the second system favours consistency over availability. Consequently, availability is sacrificed.

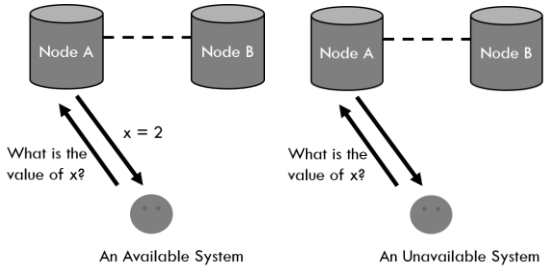


Figure 2: **Available vs. Unavailable System**

2.3 Partition Tolerance

Before clarifying the concept of partition tolerance, the term "partition" must be defined: A partition means a communication break, such as an interruption or a failure. Some examples of partitions can be crashed servers, dropped packets, power outages, computer dock-outs, network interruptions or nodes gone offline.

Partition tolerance means that the distributed system manages to operate even if an error occurs in the system, resulting in the messages sent between the nodes being lost. In other words, a partition tolerant system continues to operate, even with message loss between its nodes. As seen in Figure 3, a partition may occur in the communication channel between the nodes, as well as within the node itself, an example for the latter can be crashed node.

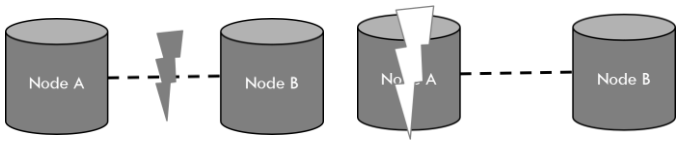


Figure 3: **Different types of partitions in a distributed system**

3 Application of the Theorem: Forfeiting Partition Tolerance, Availability or Consistency

In this part, the significance of each property, partition tolerance, availability and consistency, is discussed in the context of a distributed data system.

The importance of each CAP property is examined by observing the consequences of the absence of each property as a separate scenario. Therefore, there are three scenarios examined, first being the case in which the partition tolerance is forfeited, in order to achieve a distributed data system that is available and consistent. Secondly, a distributed data system is discussed that forfeits availability for the pursuit of consistency and partition tolerance; as for the third and last scenario, the effects of forfeiting availability are measured.

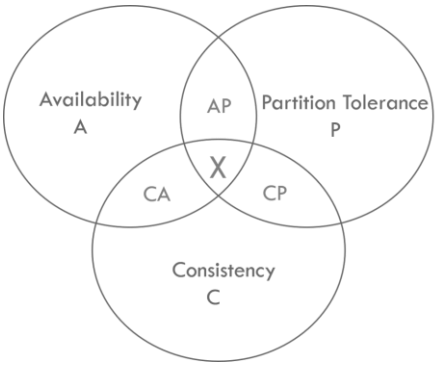


Figure 4: **CAP Theorem - "2 of 3" principle**

3.1 Forfeiting Partition Tolerance

Before even considering forfeiting partition tolerance, it is important to recall partitions, which are failures in a system that are not desired by the developer, such as network failures, crashed nodes and communication errors. In an ideal world, the developer could choose to not have them, and therefore have the possibility of giving up partition tolerance, however realistically, this is not an option. In a realistic world, the developer cannot choose to not have them, thus not having the opportunity to forfeit partition tolerance, which makes partition tolerance a must in distributed data systems. Otherwise, without the ability to tolerate partitions, the system will either not function properly or crash altogether.

Realistically speaking, it is only acceptable to forfeit partition tolerance, if the data system consists of a single node, since in that scenario, if the one and only node crashes or goes offline, there will not be a system remaining, thus making the ability of partition tolerance obsolete.

If the developer is unsure about whether to forfeit partition tolerance or not, an intelligent approach would be to estimate the probability of partitions and act accordingly. The probability of failures in a system can be calculated with the following formula [3].

$$P(\text{any failure}) = 1 - P(\text{individual node not failing})^{\text{number of nodes}}$$

Lastly, in a distributed data system that forfeits partition tolerance, there are various methods the developer can deploy in order to recover from temporary partitions. However, it is crucial to realize that such methods will only benefit in case of a temporary partition, and are usually useless when the partition is permanent, long-term or large-scale. One of such methods is called the "2-Phase-Commit Protocol". The essence of this method is that there is a coordinator node in the system. When change occurs in the system, the coordinator node sends a commit message to all of the participating nodes, which took part in this change. The change is committed, only if the coordinator node receives a positive response from all of the participating nodes, otherwise, the change is aborted.

Ultimately, what can be concluded about partition tolerance is that it is a must-have in a distributed system and should not be forfeited in most cases. This conclusion also means that since partition tolerance is a must-have, the real decision that must be made according to the Cap theorem is not really between consistency, availability and partition tolerance, since partition tolerance cannot be given up in most cases. In conclusion, the designer is usually supposed to choose between availability and consistency, in the presence of partitions.

3.2 Forfeiting Availability

A concept that is crucial in the subject of forfeiting availability is called "pessimistic locking". Pessimistic locking simply is to allow only one user to access a shared resource at a time, thus forbidding simultaneous accesses to the resource, with the premise of preserved consistency. By prohibiting the other user to reach the shared resource concurrently, the availability of the resource is inherently reduced. Pessimistic locking is a principle that is used widely in financial transactions, since consistency is of great importance in finances.

Another concept that needs to be mentioned here is called "high latency". In some cases, designers program their systems to retry communication in the presence of a communication error caused by a partition. Such systems favor consistency over availability, since they make the system unavailable by retrying communication instead of returning a response to the user, but also consistent, because the result is only returned to the user, when the partition is over, thus resulting in consistency.

3.3 Forfeiting Consistency

To start with, the notion of "optimistic locking" must be mentioned while discussing giving up consistency. Unlike pessimistic locking mentioned above, optimistic locking allows concurrent accesses to a shared resource. The resource is available to multiple users at the same time, and they can make changes to the resource simultaneously. In conclusion, optimistic locking favors availability, but comes with a price of reduced consistency. This approach is used mostly in areas, in which the consequences of inconsistent, stale data is not of great importance, and recovering from inconsistencies is not complicated. Examples of that are website editing or editing files on Google Drive.

Finally, the concept of "weaker consistency" is worth mentioning while evaluating the effects of reduced consistency. The notion of weaker consistency simply describes the fact that some systems, although favoring availability, do not give up consistency completely away, so are not completely inconsistent. By deploying methods like eventual consistency, those systems are guaranteed to become consistent, not right away, but sometime in the future.

4 ACID vs. BASE

In the fourth part of this report, a comparison between ACID and BASE approaches takes place. The motivation behind this comparison is the fact that these approaches serve as a starting point for most modern database systems, while being quite the opposite of each other in terms of the CAP properties. In fact, the comparison of these two approaches can be regarded as the first and foremost example of the availability versus consistency argument.

ACID and BASE represent two design philosophies at opposite ends of the consistency-availability spectrum [1]. (buraya da according to cap twelve years tarsi bisiler yazalim) When designing a database system, there are two data structure options, namely NoSql, also known as non-relational, and Sql, also known as relational. The most distinguishing feature of both data structures is that relational databases use ACID properties, which focuses on consistency and in contrast, non-relational databases are using BASE, which favors high availability over consistency. Eric Brewer and his colleagues created BASE in the mid-1990s because of the need of a more flexible system than ACID. BASE was designed for non-relational databases to use and benefit from increased availability.

4.1 ACID Properties

ACID is an acronym which stands for: atomicity, consistency, isolation and durability. The main focus of ACID is to guarantee the consistent state of the database, whether or not partitions are present.

Atomicity in ACID implies that the transactions are atomic, meaning they happen in a single step, and cannot be divided into smaller steps. As a result, they either happen completely or not happen at all, implying an "all or nothing" strategy.

C in ACID refers to consistency. However, the application of consistency in ACID is different from its application in CAP. "In ACID, the C means that a transaction pre-serves all the database rules, such as unique keys. In contrast, the C in CAP refers only to single copy consistency, a strict subset of ACID consistency" [1]. Thus, in database systems, which use ACID approach, consistency is maintained by all the nodes; on the other hand, consistency in CAP Theorem can be maintained by just a group of nodes in the system.

Continuing with the 'I' in ACID, isolation in the context of ACID means that every single transaction has the database system to itself and is not influenced by other transactions that may be taking place concurrently. "Isolation is at the core of the CAP theorem: if the system requires ACID isolation, it can operate on at most one side during a partition" [1]. An example to demonstrate isolation could be a network consisting of 4 servers. A partition occurs in the system, which causes a communication failure between the first and the second server. If the system requires isolation, it can operate only on one the groups, for an example the group consisting of the third and the fourth servers.

The last letter of ACID stands for durability, which guarantees the data of a terminated transaction will be stored in the database system, even in the presence of a failure that may occur after the data is stored.

4.2 BASE Properties

BASE is an acronym and stands for basic availability, soft state and eventual consistency. Basic availability is in the core of the BASE, since its development was inspired by the pursuit of increased availability in the first place. In essence, basic availability satisfies the availability part of the CAP Theorem.

Even if a latency occurs in the system, the client is promised to get a response to his or her request. The response may not be consistent, since availability requirement is solely responsible for returning any value, not the latest value specifically.

The second property, called soft state, indicates that even though the client does not write to a server and make modifications, state changes are observable within the system. Soft state is strongly related to the eventual consistency model.

Eventual consistency means that the distributed system is guaranteed to become consistent in the future, although the exact time of the system becoming consistent is unknown. Eventual consistency is promised, provided that the client does not commit any write operation during that time. An example for the eventual consistency can be shown with a file hosting service: Dropbox. While working on Dropbox files, the changes made on the files are automatically saved when the user is connected to the internet. When the user device is disconnected from the internet, the changes are not saved until reconnection. Dropbox puts the emphasis on availability, by never prohibiting the user from making changes, even though they may not be saved. When user reconnects to the internet, the changes are saved, which makes Dropbox an example for the eventual consistency model. The system did not become consistent immediately after change, but over time, consistency is achieved.

5 Formal Proof of the Theorem

The focus of this part is how the theorem was formally proven by Gilbert & Lynch. Seth Gilbert and Nancy Lynch are two computer scientists, who proved the CAP theorem at MIT in 2002. Together they wrote the report "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services", which provided the proof of the theorem for three different network types: Asynchronous network with message loss, asynchronous network without message loss and partially synchronous network with a message loss. In this report, only the proof for the first network type is discussed.

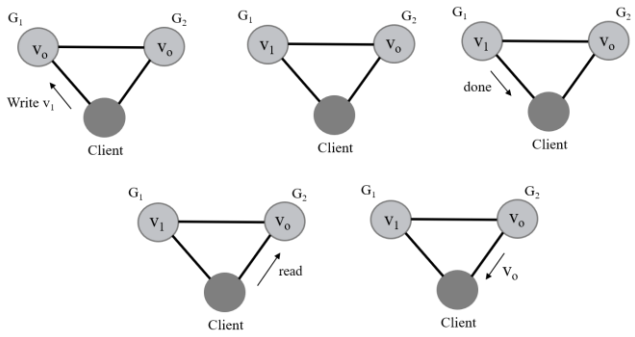


Figure 5: **Proof of the CAP Theorem: An available, partition-tolerant and inconsistent System**

Gilbert and Lynch proved Brewer's conjecture with a contradiction. They assumed a system exists, which maintains all of the three desired attributes from the CAP Theorem simultaneously. Then they proved the impossibility of this system with a model. In their model, there is a network consisting of two nodes, namely G_1 and G_2. The initial state of the nodes is v_0. First, the client writes on G_1 and changes its state from v_0 to v_1. Ideally, before G_1 acknowledges, it should replicate its state to G_2 to maintain consistency, however because of the message loss in the network, the replication is not performed. After G_1 acknowledges this change, without replicating its state to G2, the client reads from G_2. The response contains v_0, which is the old version of the state, not the freshest version v_1, as one may hope, making the system inconsistent.

With this straightforward proof, Gilbert and Lynch managed to show that the system they initially assumed to exist is not achievable. In their example, the system achieved to maintain partition tolerance, because even with a message loss the system could operate properly. Secondly, the client could perform write and read operations, thus the system provided availability as well. However, consistency could not be maintained by the system, as shown.

Consequently, Gilbert and Lynch demonstrated the validity of the conjecture stating it is only possible for a distributed system to have at most two of the three desired properties concurrently.

6 Improvements to the Initial Theorem

The theorem was later improved after its initial release. The most important contributor of this improvement was again Brewer. With his 2012 article, “Cap 12 Years Later: How the ‘Rules’ Have Changed”, he introduced new concepts that advanced the initial theorem further. The core of the theorem, concerning the decision one has to make between availability and consistency remained the same, however the newly-introduced trade-off concept replaced the former “2 of 3” principle.

With the mentioned article, Brewer also gave his readers more insight on partition detection and recovery strategies, which are investigated in-depth in the next section of this report.

6.1 Trade-offs Replacing “2 of 3”

Simply put, trade-off concept means that one does not have to forfeit availability or consistency completely in the presence of partitions, but instead find the optimal way to preserve both properties to a certain extent. There are three main reasons why this trade-off concept is superior to the “2 of 3” principle.

The first reason why the trade-off concept is favoured over the “2 of 3” principle is the fact that partitions are rare, realistically. It is still incredibly crucial to be able to tolerate them, because otherwise the system will not function properly but in reality, systems do not have partitions most of the time, which means for the most part, there is no need to give up availability or consistency, since a system without any failure will function perfectly normal, preserving both properties.

On the other hand, in case the partitions do exist, then there are ways that designers elegantly deal with them, namely partition detection and recovery strategies. These are discussed further in the next section of this report.

Secondly, designers do not have to decide between consistency and availability for the whole system. The decision may occur many times within the system, a part of the system may favour availability, whereas for another part, consistency is of greater importance.

Lastly, none of these three properties is binary, as a matter of fact, they should be thought of as a spectrum. A system for example can be available forty percent of the time, and down for the remaining sixty. The same spectrum argument is also valid for consistency, and partition tolerance. It is also worth mentioning that not all partitions are the same; a partition can either be a large-scale failure, in which none of the nodes can communicate with each other, or a much smaller one, in which only one node has crashed. Thus, making it presumably misleading to think of them as binary.

7 Partition Detection & Recovery Strategies

In this part, the partition concept is examined further, by dividing the timeline of a partition in three, namely partition detection, partition mode and partition recovery. By investigating each stage of a partition in depth, this report aims to offer the reader an insight into handling partitions.

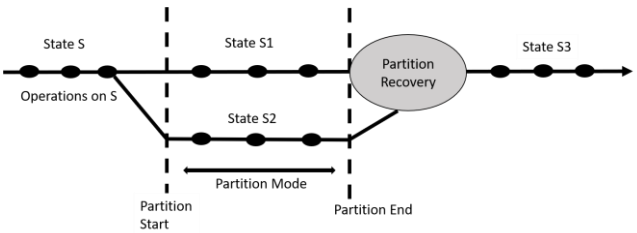


Figure 6: **Timeline of a partition**

This report attempts to clarify the concept of partitions once again with the image above. At first, the object has a particular state called S. Then, the partition starts, during which the both sides of the partition have concurrent and possibly different states, S1 and S2. After the partition, the recovery process takes place, and partition ends up with a mutual state called S3.

First the partition start needs to be detected. Detection of the partition start can be handled with different approaches, for example by putting time bounds in the system. The working principle of time bounds is that a specific time is determined, in which the response to a request made to the system should be returned under normal circumstances. If the actual response time exceeds the determined amount, the system becomes aware of the partition and activates the partition mode, which is the second step. Lastly the partition terminates, and the recovery process takes place.

7.1 The Partition Mode

During the partition, the partition mode needs to be discussed. There are two common options in terms of how a system acts during a partition. The choice of option mainly depends on how sensitive the designer is about the invariants.

In the first option, some operations are limited during the partition. By deploying this strategy, the designer guarantees that a particular invariant during the partition mode is maintained. Although it is the less risky way in terms of consistency, it comes with a cost of reduced availability.

The other option during a partition is that extra information about the operations that take place during the partition can be recorded. This information is available to use during the recovery. The problem with this option could be violated invariants, however, they are intended to be restored during partition recovery.

An example of an invariant that is suitable to risk is having unique keys in the table. If there is a table consisting of unique keys, and during the partition, the invariant is violated by some duplicate keys, it is not a major issue for the system to restore the invariant, thus making it okay to risk. [1]

On how to exactly record that extra information during the partition, a common way is to use database management system logs. DBMS logs store information about transactions in a sequential order; information about each operation is stored in a different log entry. A log entry consists of a sequence number, which acts as an identifier for the log entry, a transaction number, which is an identifier for the transaction, in which the operation takes place, pageID, redo and undo information and previous log sequence number.

In the Figure 7 below, there is a log entry with the sequence number 3. It belongs to the transaction T_1, it represents the operation A-50, which is why there is A-50 as the redo information and A+50 as the undo information since, undo represents how one can recover from that operation. It has 1 as previous log sequence

number, since it follows the log entry with the identification number 1.

As seen in the Figure 7, there is a crash between log entry 7 and 8, thus the system failed before committing transaction T_2.

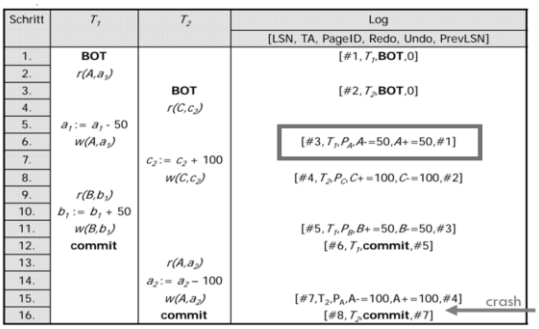
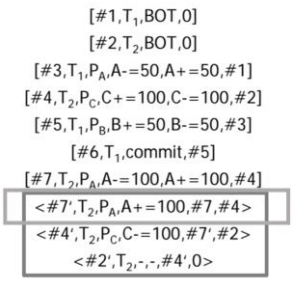


Figure 7: **Log Entry Table,** Source: TUM Grundlagen Datenbanken Lecture Slides, by Prof. Kemper

Therefore, a partition recovery must take place, in order to prevent stale data. The purpose of the recovery in this example is to is roll back Transaction 2 and discard the uncommitted changes. This can be provided by using compensating log records.



A compensating log record starts with the identification number of the most recent log entry, which includes an uncommitted change; this is followed by the transaction number, pageID and the undo information of the log entry it is fixing, as well as the previous log sequence number and lastly the identification number of the log entry that should be fixed next.

Figure 8: **Compensating log record,** Source: TUM Grundlagen Datenbanken Lecture Slides, by Prof. Kemper

7.2 The Partition Recovery

After the partition ends and the communication resumes, the system must proceed with the partition recovery stage. During the partition, each side was available and thus making forward progress, but partition has delayed some operations and violated some invariants. Therefore, there are two problems to deal with: first being, coming up with a consistent state while the second is compensating for mistakes made during the partition.

As for the first problem, coming up with a consistent state, a common way is to use compensating log records, which are discussed in the previous chapter of this report. The problem with using compensating logs is, however, is that it only works with commutative operations and using only commutative operations is neither simple nor straightforward. The reason why roll back from transaction can be achieved in the Figure 8 is that both the addition and subtraction are commutative operations.

Lastly, the second problem, compensating for mistakes needs to be solved; the main goal here is to restore the invariants. A trivial way to do that is the "last writer wins" approach, which ignores the previous writes, and only takes the very last write into account, as its name suggests.

Another way, through which one may solve this problem is called the "human factor", which suggests that instead of dealing with the mistakes through technical approaches, the methods that centre around human psychology and human contact are deployed. Compensating for mistakes using the human factor can be best explained through an example: A customer makes a purchase on a website. Although she does everything correctly with the purchase, she realizes later, in her credit card bill, that she has been charged twice for that purchase. Here, the company that is responsible for that mistake, should not only give her the money back, but should also come up with a way of compensation to satisfy her as a customer, an example of that could be a ten-dollar-coupon for her next purchase.

CONCLUSION

In conclusion, the CAP theorem was initially a conjecture made by Eric Brewer, pointing out the fact that it is impossible for a distributed data system to possess the following properties simultaneously: consistency, availability and partition tolerance. The initial version of the theorem suggested that one of these properties must be forfeited. However, over the years, through experience and observation, which was a consequence of the increased engagement with distributed data systems, as well as through the further understanding of the partition concept and discovering ways to handle them more precisely, the theorem has been majorly improved to be more flexible. "The blind sacrifice of a property approach" as known as the "2 of 3" principle was abandoned, and the trade-off concept was adopted, which introduced the "attributes are a spectrum rather than binary" understanding.

Ultimately, the most important takeaway from this report is the following:

> "System designers should not blindly sacrifice consistency or availability when partitions exist. Using the proposed approach, they can optimize both properties through careful management of invariants during partitions."
>
> -Eric Brewer, "Cap Twelve Years Later"

ACKNOWLEDGMENTS

Unless explicitly stated otherwise, all the figures and illustrations belong to the writers of this report. Every quote, example or idea taken from another paper is explicitly given credit for (with the number of the reference in square brackets); every other resource that has been used indirectly is mentioned in the references below.

REFERENCES

[1] E. Brewer, "Cap Twelve Years Later: How the 'Rules' Have Changed?",2012. https://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed

[2] E. Brewer, "Towards Robust Distributed Systems," *Proc. 19th Ann. ACM Symposium Principles of Distributed Computing* (PODC 00), ACM, 2000, pp. 7-10. https://people.eecs.berkeley.edu/~brewer/PODC2000.pdf
[3] C. Hale, "You Can't Sacrifice Partition Tolerance," 7 Oct. 2010. https://codahale.com/you-cant-sacrifice-partition-tolerance/
[4] Seth Gilbert and Nancy Lynch, "Brewer's Conjecture and The Feasibility of Consistent, Available, Partition-Tolerant Web Services", SigAct News, June 2002.
[5] Salome Simon, "Report to Brewer's Cap Theorem", University of Basel, 2012.
[6] Seth Gilbert and Nancy Lynch, "Perspectives on the CAP Theorem",2012. https://groups.csail.mit.edu/tds/papers/Gilbert/Brewer2.pdf
[7] Nathan Marz, "How to beat the CAP theorem", 2011. http://nathanmarz.com/blog/how-to-beat-the-captheorem.html
[8] Prof. Kemper, "Grundlagen Datenbanken" Lecture Slides, Technical University of Munich, 2018.