Perspectives on the CAP Theorem

Seth Gilbert
National University of Singapore

Nancy A. Lynch
Massachusetts Institute of Technology

Abstract

Almost twelve years ago, in 2000, Eric Brewer introduced the idea that there is a fundamental trade-off between *consistency*, *availability*, and *partition tolerance*. This trade-off, which has become known as the *CAP Theorem*, has been widely discussed ever since. In this paper, we review the CAP Theorem and situate it within the broader context of distributed computing theory. We then discuss the practical implications of the CAP Theorem, and explore some general techniques for coping with the inherent trade-offs that it implies.

1 Introduction

Almost twelve years ago, in 2000, Eric Brewer introduced the idea that there is a fundamental trade-off between *consistency*, *availability*, and *partition tolerance*. This trade-off, which has become known as the *CAP Theorem*, has been widely discussed ever since.

Theoretical context. Our first goal in this paper is to situate the CAP Theorem in the broader context of distributed computing theory. Some of the interest in the CAP Theorem, perhaps, derives from the fact that it illustrates a more general trade-off that appears everywhere (e.g., [4,7,15,17,23]) in the study of distributed computing: the impossibility of guaranteeing both *safety* and *liveness* in an *unreliable distributed system*:

Safety: Informally, an algorithm is *safe* if nothing bad ever happens. A quiet, uneventful room is perfectly safe. Consistency (as defined in the CAP Theorem) is a classic safety property: every response sent to a client is correct.

Liveness: By contrast, an algorithm is *live* if eventually something good happens. In a busy pub, there may be some good things happening, and there may be some bad things happening—but overall, it is quite lively. Availability is a classic liveness property: eventually, every request receives a response.

Unreliable: There are many different ways in which a system can be unreliable. There may be partitions, as is discussed in the CAP Theorem. Alternatively, there may be crash failures, message loss, malicious attacks (or Byzantine failures), etc.

The CAP Theorem, in this light, is simply one example of the fundamental fact that you cannot achieve both safety and liveness in an unreliable distributed system.

Practical implications. Our second goal in this paper is to discuss some of the practical implications of the CAP Theorem. Since it is impossible to achieve both consistency and availability in an unreliable system, it is necessary in practice to sacrifice one of these two desired properties. On the one hand, there are systems that guarantee strong consistency and provide *best effort* availability. On the other hand, there are systems that guarantee availability, and provide *best effort* consistency. Perhaps surprisingly, there is a third option: some systems may sacrifice both consistency and availability! In doing so they may achieve a trade-off better suited for the application at hand.

Roadmap. We begin in Section 2 by reviewing the CAP Theorem, as it was formalized in [16]. We then examine in Section 3 how the CAP Theorem fits into the general framework of a trade-off between safety and liveness. Finally, in Section 4, we discuss the implications of this trade-off, and various strategies for coping with it. We conclude in Section 5 with a short discussion of some new directions to consider in the context of the CAP Theorem.

2 The CAP Theorem

We begin, in this section, by reviewing Brewer's original conjecture (or, more specifically, one interpretation of the original conjecture). We discuss how the conjecture can be proved, closely following the presentation in [16].

Brewer first presented the CAP Theorem in the context of a web service. A web service is implemented by a set of *servers*, perhaps distributed over a set of geographically distant data centers. *Clients* make *requests* of the service. When a server receives a request from the service, it sends a *response*. Notice that such a generic notion of a web service can capture a wide variety of applications, such as search engines, e-commerce, on-line music services, or cloud-based data storage. For the purpose of this discussion, we will imagine the service to consist of servers $p_1, p_2, \ldots, p_n$, along with an arbitrary set of clients.

The CAP Theorem was introduced as a trade-off between *consistency*, *availability*, and *partition tolerance*. We now discuss each of these terms.

Consistency. Consistency, informally, simply means that each server returns the *right* response to each request, i.e., a response that is correct according to the desired service specification. (There may, of course, be multiple possible correct responses.) The meaning of consistency depends on the service.

Trivial services: Some services are trivial in the sense that they do not require any coordination among the servers. For example, if the service is supposed to return the value of the constant π to 100 decimal places, then a correct response is exactly that. Since no coordination among the servers is required, trivial services do not fall within the scope of the CAP Theorem.

Weakly consistent services: In response to the inherent trade-offs implied by the CAP Theorem, there has been much work attempting to develop weaker consistency requirements that still provide useful services and yet avoid sacrificing availability. A distributed web cache is an example of one such system. We will discuss such systems later in Section 4.2.

Simple services: For the purposes of discussing the CAP Theorem, we focus on simple services that have straightforward correctness requirements: the semantics of the service are specified by a sequential specification, and operations are atomic. A sequential specification defines a service in terms of its execution on a single, centralized server: the centralized server maintains some state, and each request is processed in order, updating the state and generating a response. A web service is atomic if, for every operation, there is a single instant in between the request and the response at which the operation *appears* to occur. (Alternatively, this is equivalent to saying that, from the perspective of the clients, it is as if all the operations were executed by a single centralized server.) While there are several weaker consistency conditions used in practice (e.g., sequential consistency, causal consistency, etc.), we focus on atomicity due to its simplicity.

Complicated services: Many real services have more complicated semantics. Some services cannot be specified by sequential specifications. Others simply require more complicated coordination, transactional semantics, etc. The same CAP trade-offs typically apply, but for simplicity we do not focus on these cases.

For the purpose of this section, we focus on a service that implements a *read/write* atomic shared memory: the service provides its clients with a single (emulated) register, and each client can read or write from that register.

Availability. The second requirement of the CAP Theorem is that the service guarantee *availability*. Availability simply means that each request eventually receive a response. Obviously, a fast response is better than a slow response, but for the purpose of CAP, it turns out that even requiring an *eventual* response is sufficient to create problems. (In most real systems, of course, a response that is sufficiently late is just as bad as a response that never occurs.)

Partition-tolerance. The third requirement of the CAP theorem is that the service be partition tolerant. Unlike the other two requirements, this property can be seen as a statement regarding the underlying system: communication among the servers is not reliable, and the servers may be partitioned into multiple groups that cannot communicate with each other. For our purposes, we simply treat communication as faulty: messages may be delayed and, sometimes, lost forever. (Again, it is worth pointing out that a message that is delayed for sufficiently long may as well be considered lost, at least in the context of practical systems.)

The CAP Theorem. Thus, for the purpose of this section, the CAP Theorem can be stated as follows: *In a network subject to communication failures, it is impossible for any web service to implement an atomic read/write shared memory that guarantees a response to every request.*

Proof sketch. Having stated the CAP Theorem, it is relatively straightforward to prove it correct. (See [16] for more details.) Consider an execution in which the servers are partitioned into two disjoint sets: $\{p_1\}$ and $\{p_2, \ldots, p_n\}$. Some client sends a read request to server p_2. Since p_1 is in a different component of the partition from p_2, every message from p_1 to p_2 is lost. Thus, it is impossible for p_2 to distinguish the following two cases:

- There has been a previous write of value v_1 requested of p_1, and p_1 has sent an *ok* response.
- There has been a previous write of value v_2 requested of p_1, and p_1 has sent an *ok* response.

No matter how long p_2 waits, it cannot distinguish these two cases, and hence it cannot determine whether to return response v_1 or response v_2. It has the choice to either eventually return a response (and risk returning the wrong response) or to never return a response.

In fact, if communication is *asynchronous* (i.e., processes have no *a priori* bound on how long it takes for a message to be delivered), it is possible for the same situation to occur even in executions in which there are no messages lost, i.e., no actual partition! Notably, in the scenario describe above, server p_2 eventually must return a response, even if the system is partitioned; if the message delay from p_1 to p_2 is sufficiently large that p_2 believes the system to be partitioned, then it may return an incorrect response, despite the lack of partitions. Thus it is even impossible to guarantee consistency when there are no partitions, and return a bad (inconsistent) answer only when partitions occur.

3 Theoretical Context

The trade-off between consistency and availability in a partition-prone system is a particular example of the more general trade-off between safety and liveness in an unreliable system. This impossibility has played a key role in the study of distributed computing, and perhaps provides some insight into both the CAP Theorem and the manner in which algorithm designers and software engineers have circumvented it.

3.1 Consistent, Availability, and Partition-Tolerance

We first review what it means for a property to be a "safety property" or a "liveness property," indicating their connection to the CAP Theorem.

A *safety* property is one that states *nothing bad ever happens.* That is, it requires that at every point in every execution, the property holds. Consistency requirements are almost always safety properties. For example, when we say that an algorithm guarantees atomic consistency, we are claiming that in every execution, every response is correct (with respect to the other "prior" operations).

A *liveness* property is one that states that *eventually something good happens*. A liveness property says nothing about the state at any instant in time; it requires only that if an execution continues for long enough, then something desirable happens. Availability is a classic liveness property: it states that eventually, every request receives a response.

Thus, from this perspective, the CAP Theorem states that it is impossible for any protocol implementing an atomic read/write register to guarantee both safety and liveness in a system prone to partitions.

3.2 Agreement is Impossible

Understanding the relationship between safety and liveness properties has been a long-standing question in distributed computing. It achieved widespread prominence when, in 1985, Fischer, Lynch, and Paterson [15] showed that fault-tolerant agreement is impossible in an asynchronous system.

They focused on the problem of *consensus*, in which each process p_i begins with an initial value v_i, and the processes all have to agree on one of those values. There are three requirements: (i) *agreement*: every process must output the same value; (ii) *validity*: every value output must have been provided as the input for some process; and

(iii) *termination*: eventually, every process must output a value. It should be immediately clear that agreement and validity are safety properties, while termination is a liveness property, and hence the impossibility of consensus is an important example of the inherent trade-off between safety and liveness.

The problem of consensus has attracted significant interest, as it is at the heart of the *replicated state machine* approach [8, 19–21], one of the most common techniques for building reliable distributed systems. In order to improve availability, a service may be replicated at a set of servers. In order to maintain consistency, the servers agree on every update to the service—and notably, on the order of the updates[1].

The safety requirements of consensus are strictly harder than those we have considered with respect to the CAP Theorem: achieving agreement is (provably) harder than simply implementing an atomic read/write register. This means that the CAP Theorem also implies that you cannot achieve consensus in a system subject to partitions.

In [15], Fischer et al. considered a system with no partitions[2]. Instead, the focus on an even more benign failure model: crash failures. Specifically, they assume that one (unknown) process in the system may fail by crashing, i.e., it may cease operation. No partitions are possible, and almost all of the processes in the system can continue to communicate reliably.

The surprising conclusion of [15] is that in such a system, consensus is impossible. In fact, for every purported consensus protocol that guarantees agreement and validity, there is some execution *in which there are no failures* where the algorithm never terminates! In the case of consensus, safety and liveness are impossible if the system is even potentially slightly faulty.

3.3 Coping with the Safety/Liveness Trade-off for Consensus

Almost immediately after the publication of [15], researchers in distributed computing began examining this inherent trade-off between safety and liveness in more depth. While the results discussed in this section focus on the problem of consensus, they provide context for the analogous questions raised by the CAP Theorem.

Given that safety and liveness are impossible in systems that are *sufficiently unreliable*, the first natural question was under what conditions it *is possible* to achieve both. Much research has focused on the issue of *network synchrony*: what level of synchrony is necessary to avoid the inherent trade-off? How many failures can be tolerated? In the language of the CAP Theorem: what level of network reliability is needed to achieve both consistency and availability? In Section 3.3.1, we briefly review the connection between network synchrony and the (im)possibility of achieving consensus.

A second natural question focuses on the question of consistency: given that the network is unreliable, what is the maximum level of consistency that can be achieved? In Section 3.3.2 we briefly review one answer to this question.

3.3.1 Synchrony

In an attempt to answer the question of how much reliability is needed to solve consensus (i.e., to satisfy both the safety and liveness properties), researchers have focused on the issue of network synchrony.

A network is *synchronous* if it satisfies the following properties: (i) every process has a clock, and all the clocks are synchronized; (ii) every message is delivered within a fixed and known amount of time; and (iii) every process takes steps at a fixed and known rate. We can think of such systems as progressing in rounds, where within each round, each process: sends some messages, receives all the messages that were sent to it in that round, and performs some local computation.

Time complexity. If a system is wholly synchronous, consensus can be solved, i.e., the trade-off between safety and liveness can be avoided. Notably, consensus requires $f + 1$ rounds, if up to f servers may crash (see [18, 22]). Since consensus is impossible in an asynchronous system if there is even one crash failure, this motivates the question: how much synchrony is needed to solve consensus? And do real systems provide that necessary level of synchrony?

Soon after [15], Dwork et al. [14] attempted to answer this question, exploring a variety of models of *partial synchrony*. Most notably, they introduced the idea of *eventual synchrony*: a system may experience some periods

[1] Many distributed applications deployed today actually have more similarity to consensus than they do to atomic registers.

[2] Notice that as a result, the CAP Theorem does not imply the results in [15]. Conversely, [15] does not immediately imply the CAP Theorem.

of synchrony and some periods of asynchrony, but as long as it eventually stabilizes and maintains synchrony for a sufficiently long period of time, we can solve consensus.

How long a "window of synchrony" is necessary to solve consensus? Dutta and Guerraoui [13] showed that at least $f + 2$ rounds are necessary. Alistarh et al. [3] recently resolved the question, showing that $f + 2$ rounds of synchrony are also sufficient.

There is also a connection between the synchrony of a system and the crash-tolerance. In a synchronous system, we can solve consensus for any number of failures. In an asynchronous system, consensus is impossible for even one failure. In an eventually synchronous system, however, we can solve consensus only if there are $< n/2$ crash failures, where n is the number of servers. If there are $\geq n/2$ crash failures, consensus is again impossible. (In this case, a partitioning argument, much as in the CAP Theorem, leads to the impossibility.) Most practical consensus implementations today are designed (either explicitly or implicitly) for eventually synchronous systems [6, 8, 20].

Failure detectors. Another line of research has pursued a different approach to answering the question of how much synchrony is needed to solve consensus. Chandra et al. [7] introduced the idea of a *failure detector*, an oracle that provides sufficient information for processes to solve consensus in an asynchronous crash-prone system. In many ways, a failure detector exactly encapsulates the synchrony requirements for consensus. They showed [9] that a particular failure detector Ω is the weakest failure detector for solving consensus. The failure detector Ω essentially encapsulates a *leader election* service, and in fact, most practical consensus protocols today use such a service as an important component of their system—the leader is often referred to as the "master" or the "primary."

Explicit assumptions. Finally, Aguilera et al. [1, 2] have explored a specific minimal set of link reliability and synchrony assumptions sufficient for solving consensus. The protocols developed in these papers establish quite minimal conditions for solving consensus.

3.3.2 Consistency

Another response to the impossibility of agreement has been an attempt to answer the question: what is the strongest form of consistency we can guarantee in a system with f crash failures?

In an attempt to answer this question, Chaudhuri [10] introduced the problem of *set agreement*. Much like in consensus, each process begins with some value and eventually chooses an output. (That is, the validity and termination conditions are identical.) Unlike consensus, however, some disagreement in the output is allowed. Specifically, for the problem of k-set agreement, there may be up to k-different output values.

This weaker consistency guarantee leads to a sequence of problems: 1-set agreement, 2-set agreement, 3-set agreement, ..., n-set agreement. Notice that 1-set agreement is identical to consensus, and n-set agreement is trivial (i.e., each process simply outputs its own initial value). Thus we know that 1-set agreement is impossible if there is even one crash failure, and n-set agreement can tolerate an arbitrary number of crash failures.

In a seminal sequence of papers (a subset of which were awarded the Gödel Prize in 2004) Borowski, Gafni, Herlihy, Saks, Shavit, and Zaharoglou [4, 17, 23] showed that k-set agreement can be solved if and only if there are at most $k - 1$ crash failures. (At the same time, they revealed a deep connection between distributed computing and algebraic topology). Thus, in a sense, k-set agreement is the "most" agreement you can get in a system with $k - 1$ failures, if you want to ensure availability.

Chaudhuri et al. [11] further developed the techniques of [17], relating the degree of consistency to the running time of k-set agreement: in a synchronous system with t failures, at least $\lfloor t/k \rfloor + 1$ rounds are necessary and sufficient.

From a theoretical perspective, these results for k-set agreement relate the strength of the consistency requirement to the availability of the system.

4 Practical Implications

Despite the negative implications of the CAP Theorem, practitioners building distributed services must still do the impossible. And in fact, they have continued to successfully build and deploy systems, addressing and overcoming the challenges posed by the CAP Theorem in various ways.

When dealing with unreliable networks, there are seemingly only two reasonable approaches: sacrifice availability or sacrifice consistency. This is the implication of the CAP Theorem: we cannot achieve consistency and availability in a partition-prone network. However, there is also another interesting approach that shows up frequently in practice: a larger system is segmented into different subsystems, each of which may choose a different trade-off. Moreover, this segmentation may take place along several different dimensions. From the perspective of the entire system, it may seem as if the software architects have sacrificed *both* consistency and availability! Yet the resulting design often yields a system that both responds well to most user requests, even under bad network conditions, and also provides high levels of consistency where consistency is required.

In this section, we will give some examples of designs that weaken availability (Section 4.1) and consistency (Section 4.2), and then discuss several approaches that guarantee neither consistency nor availability (Sections 4.3 and 4.4).

4.1 Best Effort Availability

Perhaps the most common approach to dealing with unreliable networks is to design a service that guarantees consistency, i.e., correct operation, regardless of the network behavior. The service is then optimized to provide best effort availability, i.e., to be as responsive as is possible given the current network conditions.

This design makes sense when operating in a network in which communication is typically reliable and timely, and it is only on rare occasions that partitions or other network anomalies occur. When all the servers running a service are located in the same data center, this approach is a good one.

A recent popular example of this approach is the *Chubby Lock Service* [5, 8], which was built by Google and is used extensively in the Google infrastructure (e.g., it supports the Google File System, BigTable, etc.). The Chubby Service, essentially, provides a simple file system optimized for small files. It is used, primarily, to share metadata and to provide a centralized point to manage coarse-grained locks in a distrusted manner. (Google has reported that one of its primary uses, today, is as a naming service that replaces DNS.)

Chubby provides strong consistency: at its heart is a distributed database, based on a *primary-backup* design. Consistency among the servers is ensured by using a replicated state machine protocol (specifically, Paxos [20]) to maintain synchronized logs. Chubby continues to operate as long as no more than half the servers fail, and it is guaranteed to make progress whenever the network is reliable. If the Chubby servers were to be partitioned, the service would become unavailable.

On the other hand, Chubby is optimized for the case where there is a stable primary and there are no partitions. In this case, it delivers a very high degree of availability. Significant design and engineering efforts have gone into ensuring that under such circumstances, Chubby can respond to requests very quickly. (For example, “read-only” requests are prioritized, and the replicated state machine protocol is modified so that such requests do not need to incur any additional network traffic.)

This design works well for Chubby, as each Chubby “cell” is deployed in a single data center[3]. Communication within a Chubby cell is typically fast and reliable, and the failure of a primary is not too frequent. (Users of the Chubby service are still encouraged to expect, and not treat as an error, occasional delays due to changes in the primary.) Thus Chubby provides guaranteed consistency, and a very high level of availability in the common case.

4.2 Best Effort Consistency

For some applications, sacrificing availability is not an option: users require that it be responsive in all situations. Moreover, when the application is deployed over a wide area (rather than within a data center), the level of availability that can be achieved by a strongly consistent service may degrade rapidly. In such situations, designers sacrifice consistency: a response (and preferably a fast response) is guaranteed at all times. As a result, the response may not always be correct. Consistency is provided only in a best effort sense.

The classic example of this is web caching, as pioneered by Akamai (and others). Web content (e.g., images and video) is cached on servers that are placed in data centers throughout the world. Whenever a user requests a given

[3] There is, in fact, a global Chubby cell with servers deployed around the world. It would be interesting to know more about the availability and performance of this cell.

web page, the content can be delivered from a nearby web cache. Such a system guarantees a very high level of availability: the proximity of the cache servers to the end users ensure both that the responses are rapid, and also that network connectivity issues rarely prevent a response.

On the other hand, the consistency guarantees are (potentially) quite minimal. When a web page is updated, it may take some time for the new content to propagate to all the cache servers. There is no guarantee that all users accessing a web page at any given time receive the exact same content. The caching service does its best to provide up-to-date content.

For Akamai (and other content delivery networks), this trade-off makes sense. Users viewing content on the web do not necessarily require strong consistency (i.e., two different users in different locations may view two slightly different versions of a web page). If the content viewed by a user is slightly out-of-date, there is usually little harm. On the other hand, users loading a web page have little patience: a fast response is critical. (The problem of fast response time is even more problematic on mobile wireless devices.) In addition, users are geographically located around the world. (This can be contrasted with Chubby, which is primarily used within a single data center.) The geographic dispersal implies that to achieve sufficient availability (and performance), consistency must be sacrificed.

4.3 Trading Consistency for Availability

There has also been some effort to more precisely tune the trade-off between consistency and availability. For example, it may be acceptable for some content to be one hour out of date, but not one day out of date. As long as the network connectivity has been good recently (i.e., in the last hour), we should be able to provide this level of consistency with good availability. On the other hand, if there are long-lasting partitions (e.g., more than one day), than it will be impossible to remain available. By setting the threshold for how out of date the data can be, the system designer can precisely specify the CAP trade-off.

This idea was explored in some detail by Yu and Vahdat [24, 25]. They developed a theory of "continuous consistency" and implemented the TACT toolkit, which enables replicated applications to specify exactly the desired consistency (in terms of *conits*, a unit of continuous consistency).

An interesting component of the TACT toolkit is that the desired level of consistency can be updated dynamically, as the application executes. An airline reservation system is one example (discussed in [24]) of why this might be useful. When most of the seats on the airplane are available, it is usually safe for the reservation system to rely on somewhat out-of-date data—even if a few additional seats have been reserved, the new reservation can likely be accommodated. As the plane is filled, however, the reservation system requires increasingly accurate data to ensure that the plane is not overbooked. Using TACT, the reservation system could request increasing levels of consistency as the number of available seats diminishes.

Notice that such a system provides neither strong consistency nor guaranteed availability: data can be out of date (i.e,. inconsistent), and yet if there is a significant network partition, the service may still be unavailable. Even so, this type of trade-off often makes sense as it may significantly increase the level of network disruption that the service can tolerate before it must compromise consistency or availability.

4.4 Segmenting Consistency and Availability

Many systems do not have a single uniform requirement. Some aspects of the system require strong consistency, and some require high availability. A quite natural approach to the limitations of the CAP Theorem is to redesign the system, segmenting it into components that provide different types of guarantees. In doing so, we again may end up with a service that, as a whole, guarantees neither consistency nor availability. Yet in the end, each part of the service provides exactly what is needed.

In this section, we discuss some of the dimensions along which a system might be partitioned. It is not always clear the precise guarantees that such segmentation provides, as it tends to be specific to the given application and the particular partitioning. It remains an open question to better understand these types of partitioning schemes.

Data partitioning. Different types of data may require different levels of consistency and availability. For example, an on-line shopping cart may be highly available, responding rapidly to user requests; yet it may be occasionally inconsistent, losing a recent update in anomalous circumstances. The on-line product information for an e-commerce site may be somewhat inconsistent: users will tolerate somewhat out-of-date inventory information. The check-out/billing/shipping records, however, have to be strongly consistent: a user will be very unhappy if a finalized order does not reflect her intended purchase. When designing a system, different data may require different trade-offs.

Operation partitioning. A second line along which an application can be partitioned is the operational dimension. Different operations may require different levels of consistency and availability. As a simple example, consider a system that guarantees high availability for read-only operations, while operations that modify the database may not respond during network partitions[4]. Moreover, different types of updates might provide different levels of consistency: a "purchase" operation should guarantee consistency, while a "query" operation might return out of date data. The PNUTS system [12], implemented by Yahoo, provides exactly this style of guarantee with differing trade-offs for different types of read and write operations. In order to achieve a good user experience, different operations may require different trade-offs.

Functional partitioning. Many services can be divided into different subservices which have different requirements. For example, an application might use a service such as Chubby (discussed in Section 4.1) for coarse-grained locks and distributed coordination (i.e., strong consistency). It might at the same time use a service such as DNS to handle naming: DNS is a classic example of caching, providing relatively weak consistency but high availability. The same service might use yet a third subservice, with a different consistency/availability trade-off for content distribution.

User partitioning. Network partitions, and poor network performance in general, typically correlate with real geographic distance: users that are far away are more likely to see poor performance. Thus a service like Craigslist might elect to divide its servers among two different data centers—one on the east coast of the US and one on the west coast of the US. Craigslist users from cities in California rely on the west coast data center, and hence get high availability. Moreover, since the California data is stored and maintained within a single west coast data center, consistency among the west coast servers can be readily achieved. The same holds true for east coast users that rely on the east coast data center. (On the other hand, users from New York inquiring about Craigslist ads in San Francisco may see less good performance, under this design.) Similarly, one could imagine that a social networking site might try to partition its users, ensuring high availability among groups of friends. (Again, the geographic correlation of friendships makes this more feasible.)

Hierarchical partitioning. Some applications are organized hierarchically, partitioning along these different dimensions multiple times. At the top level, an application encompasses the entire world or the entire database; subsequent levels of the hierarchy partition the world into geographically smaller pieces, or the database into smaller parts. At each level of the hierarchy, the system may provide a different level of performance: better availability toward the leaves, or less consistency toward the root. For example, as you descend a geographically-organized hierarchy, the limitations of the CAP Theorem becomes less and less onerous as the relevant servers become better and better connected.

5 The CAP Theorem in Future Systems

As we have discussed in this paper, the CAP Theorem is one example of a fundamental trade-off between safety and liveness in fault-prone systems. Examining this inherent trade-off yields some insights into how systems can be designed to meet an application's needs, despite unreliable networks: software architects have explored strongly consistent solutions, with best-effort availability; they have explored weakly consistent solutions with high availability; and they have explored systems that mix both weaker availability and weaker consistency in varying ways.

At the same time, the networked world has changed significantly in the last ten years, creating new challenges for system designers, and new areas in which these same inherent trade-offs can be explored. We need new theoretical insights to address these challenges, and new techniques for coping with the problem in real-world systems.

[4] Note that if every read operation returns a response, and every write operation is consistent, then in the presence of a crashed server, such a system would necessarily become unresponsive for all write operations.

Scalability: Increasingly, we require that our systems be scalable, designed not just for today's customers but also for growth tomorrow. Intuitively, we think of a system as *scalable* if it can grow efficiently, using new resources efficiently to handle more load. There appear to be inherent trade-offs between scalability and consistency. For example, in order to efficiently use new resources, there must be coordination among those resources; the consistency required for this coordination appears subject to the CAP Theorem trade-offs. Studying this question may help to explain why even within a data center, where there are rarely partitions, it seems difficult to efficiently scale strongly consistent protocols (like Paxos [20]).

Tolerating attacks: The CAP Theorem focuses on network partitions: sometimes, some servers cannot communicate reliably. Increasingly, however, we are seeing more severe attacks on networks. For example, denial-of-service attacks are becoming a near continuous threat to everyday network operations. A denial-of-service attack, however, cannot simply be modeled as a network partition. Similarly, we are seeing problems with malicious users hacking servers and otherwise disrupting major internet services. Tolerating these more problematic forms of disruption requires a somewhat different understanding of the fundamental consistency/availability trade-offs.

Mobile wireless networks: The CAP Theorem initially focused on wide-area internet services. Today, however, a significant (and increasing) percentage of internet traffic is initiated by mobile devices. Many of the same trade-offs explored in the context of the CAP Theorem also hold in mobile networks—and many of the problems are even harder to resolve.

Notably, wireless communication is notoriously unreliable. The key problem that motivated the CAP Theorem was the frequency of semi-stable partitions that change every few minutes. In a wireless networks, partitions are less common. However, unpredictable message loss is very common, and message latencies can vary significantly.

In addition, the types of applications being deployed in wireless networks may be somewhat different. The CAP Theorem was motivated by internet search engines and e-commerce web sites. There is a new generation of wireless applications, however, that tend to focus on different priorities: geography and proximity are critical; social interactions are primary; and privacy has a somewhat more immediate meaning. For example, consider *foursquare*, an application in which users *check-in* to locations, and initiate comments and discussion based on where they are.

By re-examining the CAP Theorem in the context of wireless networks, we may hope to better understand the unique trade-offs that occur in these types of scenarios.

References

[1] Marcos Kawazoe Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. Communication-efficient leader election and consensus with limited link synchrony. In *The Proceedings of the International Symposium on Principles of Distributed Computing (PODC)*, pages 328–337, 2004.

[2] Marcos Kawazoe Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. On implementing omega in systems with weak reliability and synchrony assumptions. *Distributed Computing*, 21(4):285–314, 2008.

[3] Dan Alistarh, Seth Gilbert, Rachid Guerraoui, and Corentin Travers. How to solve consensus in the smallest window of synchrony. In *The Proceedings of the International Symposium on Distributed Computing (DISC)*, pages 32–46, 2008.

[4] E. Borowsky and E. Gafni. Generalized FLP impossibility result for t-resilient asynchronous computations. In *The Proceedings of the Symposium on Theory of Computing (STOC)*, pages 91–100, 1993.

[5] Michael Burrows. The chubby lock service for loosely-coupled distributed systems. In *The Proceedings of the Symposium on Operating System Design and Implementation (OSDI)*, pages 335–350, 2006.

[6] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, 2002.

[7] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[8] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *The Proceedings of the International Symposium on Principles of Distributed Computing (PODC)*, pages 398–407, New York, NY, USA, 2007.

[9] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, 1996.

[10] S. Chaudhuri. More choices allow more faults: Set consensus problems in totally asynchronous systems. *Information & Computation*, 105(1):132–158, 1993.

[11] S. Chaudhuri, M. Herlihy, N. A. Lynch, and M. R. Tuttle. A tight lower bound for k-set agreement. In *The Proceedings of the Symposium on Foundations of Computer Science (FOCS)*, pages 206–215. IEEE, 1993.

[12] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *PVLDB*, 1(2):1277–1288, 2008.

[13] P. Dutta and R. Guerraoui. The inherent price of indulgence. In *The Proceedings of the International Symposium on Principles of Distributed Computing (PODC)*, pages 88–97, 2002.

[14] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, 1988.

[15] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[16] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SigAct News*, June 2002.

[17] M. Herlihy and N. Shavit. The topological structure of asynchronous computability. *Journal of the ACM*, 46(6):858–923, 1999.

[18] L. Lamport and M. Fischer. Byzantine generals and transaction commit protocols. Unpublished, April 1982.

[19] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[20] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[21] Butler W. Lampson. How to build a highly available system using consensus. In *Workship on Distributed Algorithms (WDAG)*, pages 1–17, London, UK, 1996. Springer-Verlag.

[22] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufman, 1996.

[23] M. E. Saks and F. Zaharoglou. Wait-free k-set agreement is impossible: The topology of public knowledge. *SIAM Journal of Computing*, 29(5):1449–1483, 2000.

[24] Haifeng Yu and Amin Vahdat. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Transactions on Computer Systems*, 20(3):239–282, 2002.

[25] Haifeng Yu and Amin Vahdat. The costs and limits of availability for replicated services. *ACM Transactions on Computer Systems*, 24(1):70–113, 2006.