A Critique of the CAP Theorem

Martin Kleppmann

Abstract

The *CAP Theorem* is a frequently cited impossibility result in distributed systems, especially among *NoSQL* distributed databases. In this paper we survey some of the confusion about the meaning of CAP, including inconsistencies and ambiguities in its definitions, and we highlight some problems in its formalization. CAP is often interpreted as proof that eventually consistent databases have better availability properties than strongly consistent databases; although there is some truth in this, we show that more careful reasoning is required. These problems cast doubt on the utility of CAP as a tool for reasoning about trade-offs in practical systems. As alternative to CAP, we propose a *delay-sensitivity* framework, which analyzes the sensitivity of operation latency to network delay, and which may help practitioners reason about the trade-offs between consistency guarantees and tolerance of network faults.

1 Background

Replicated databases maintain copies of the same data on multiple nodes, potentially in disparate geographical locations, in order to tolerate faults (failures of nodes or communication links) and to provide lower latency to users (requests can be served by a nearby site). However, implementing reliable, fault-tolerant applications in a distributed system is difficult: if there are multiple copies of the data on different nodes, they may be inconsistent with each other, and an application that is not designed to handle such inconsistencies may produce incorrect results.

In order to provide a simpler programming model to application developers, the designers of distributed data systems have explored various consistency guarantees that can be implemented by the database infrastructure, such as linearizability [30], sequential consistency [38], causal consistency [4] and pipelined RAM (PRAM) [42]. When multiple processes execute operations on a shared storage abstraction such as a database, a consistency model describes what values are allowed to be returned by operations accessing the storage, depending on other operations executed previously or concurrently, and the return values of those operations.

Similar concerns arise in the design of multiprocessor computers, which are not geographically distributed, but nevertheless present inconsistent views of memory to different threads, due to the various caches and buffers employed by modern CPU architectures. For example, x86 microprocessors provide a level of consistency that is weaker than sequential, but stronger than causal consistency [48]. However, in this paper we focus our attention on distributed systems that must tolerate partial failures and unreliable network links.

A strong consistency model like linearizability provides an easy-to-understand guarantee: informally, all operations behave as if they executed atomically on a *single copy* of the data. However, this guarantee comes at the cost of reduced performance [6] and fault tolerance [22] compared to weaker consistency models. In particular, as we discuss in this paper, algorithms that ensure stronger consistency properties among replicas are more sensitive to message delays and faults in the network. Many real computer networks are prone to unbounded delays and lost messages [10], making the fault tolerance of distributed consistency algorithms an important issue in practice.

A *network partition* is a particular kind of communication fault that splits the network into subsets of nodes such that nodes in one subset cannot communicate with nodes in another. As long as the partition exists, any data modifications made in one subset of nodes cannot be visible to nodes in another subset, since all messages between them are lost. Thus, an algorithm that maintains the illusion of a single copy may have to delay operations until the partition is healed, to avoid the risk of introducing inconsistent data in different subsets of nodes.

This trade-off was already known in the 1970s [22, 23, 32, 40], but it was rediscovered in the early 2000s, when the web's growing commercial popularity made

geographic distribution and high availability important to many organizations [18, 50]. It was originally called the *CAP Principle* by Fox and Brewer [16, 24], where CAP stands for *Consistency*, *Availability* and *Partition tolerance*. After the principle was formalized by Gilbert and Lynch [25, 26] it became known as the *CAP Theorem*.

CAP became an influential idea in the NoSQL movement [50], and was adopted by distributed systems practitioners to critique design decisions [31]. It provoked a lively debate about trade-offs in data systems, and encouraged system designers to challenge the received wisdom that strong consistency guarantees were essential for databases [17].

The rest of this paper is organized as follows: in section 2 we compare various definitions of consistency, availability and partition tolerance. We then examine the formalization of CAP by Gilbert and Lynch [25] in section 3. Finally, in section 4 we discuss some alternatives to CAP that are useful for reasoning about trade-offs in distributed systems.

2 CAP Theorem Definitions

CAP was originally presented in the form of *"consistency, availability, partition tolerance: pick any two"* (i.e. you can have CA, CP or AP, but not all three). Subsequent debates concluded that this formulation is misleading [17, 28, 47], because the distinction between CA and CP is unclear, as detailed later in this section. Many authors now prefer the following formulation: if there is no network partition, a system can be both consistent and available; when a network partition occurs, a system must choose between either consistency (CP) or availability (AP).

Some authors [21, 41] define a CP system as one in which a majority of nodes on one side of a partition can continue operating normally, and a CA system as one that may fail catastrophically under a network partition (since it is designed on the assumption that partitions are very rare). However, this definition is not universally agreed, since it is counter-intuitive to label a system as "available" if it fails catastrophically under a partition, while a system that continues partially operating in the same situation is labelled "unavailable" (see section 2.3).

Disagreement about the definitions of terms like *availability* is the source of many misunderstandings about CAP, and unclear definitions lead to problems with its formalization as a theorem. In sections 2.1 to 2.3 we survey various definitions that have been proposed.

2.1 Availability

In practical engineering terms, *availability* usually refers to the proportion of time during which a service is able to successfully handle requests, or the proportion of requests that receive a successful response. A response is usually considered successful if it is valid (not an error, and satisfies the database's safety properties) and it arrives at the client within some timeout, which may be specified in a *service level agreement* (SLA). Availability in this sense is a metric that is empirically observed during a period of a service's operation. A service may be available (up) or unavailable (down) at any given time, but it is nonsensical to say that some software package or algorithm is 'available' or 'unavailable' in general, since the uptime percentage is only known in retrospect, after a period of operation (during which various faults may have occurred).

There is a long tradition of *highly available* and *fault-tolerant* systems, whose algorithms are designed such that the system can remain available (up) even when some part of the system is faulty, thus increasing the expected mean time to failure (MTTF) of the system as a whole. Using such a system does not automatically make a service 100% available, but it may increase the observed availability during operation, compared to using a system that is not fault-tolerant.

2.1.1 The A in CAP

Does the A in CAP refer to a property of an algorithm, or to an observed metric during system operation? This distinction is unclear. Brewer does not offer a precise definition of availability, but states that "availability is obviously continuous from 0 to 100 percent" [17], suggesting an observed metric. Fox and Brewer also use the term *yield* to refer to the proportion of requests that are completed successfully [24] (without specifying any timeout).

On the other hand, Gilbert and Lynch [25] write: "For a distributed system to be continuously available, every request received by a non-failing node in the system must result in a response".[1] In order to prove a re-

[1] This sentence appears to define a property of *continuous availability*, but the rest of the paper does not refer to this "continuous" aspect.

sult about systems in general, this definition interprets availability as a property of an algorithm, not as an observed metric during system operation – i.e. they define a system as being "available" or "unavailable" statically, based on its algorithms, not its operational status at some point in time.

One particular execution of the algorithm is available if every request in that execution eventually receives a response. Thus, an algorithm is "available" under Gilbert and Lynch's definition if *all* possible executions of the algorithm are available. That is, the algorithm must guarantee that requests always result in responses, no matter what happens in the system (see section 2.3.1).

Note that Gilbert and Lynch's definition requires *any* non-failed node to be able to generate valid responses, even if that node is completely isolated from the other nodes. This definition is at odds with Fox and Brewer's original proposal of CAP, which states that "data is considered highly available if a given consumer of the data can always reach *some* replica" [24, emphasis original].

Many so-called highly available or fault-tolerant systems have very high uptime in practice, but are in fact "unavailable" under Gilbert and Lynch's definition [34]: for example, in a system with an elected leader or primary node, if a client that cannot reach the leader due to a network fault, the client cannot perform any writes, even though it may be able to reach another replica.

2.1.2 No maximum latency

Note that Gilbert and Lynch's definition of availability does not specify any upper bound on operation latency: it only requires requests to *eventually* return a response within some unbounded but finite time. This is convenient for proof purposes, but does not closely match our intuitive notion of availability (in most situations, a service that takes a week to respond might as well be considered unavailable).

This definition of availability is a pure liveness property, not a safety property [5]: that is, at any point in time, if the response has not yet arrived, there is still hope that the availability property might still be fulfilled, because the response may yet arrive – it is never too late. This aspect of the definition will be important in section 3, when we examine Gilbert and Lynch's proofs in more detail.

(In section 4 we will discuss a definition of availability that takes latency into account.)

2.1.3 Failed nodes

Another noteworthy aspect of Gilbert and Lynch's definition of availability is the proviso of applying only to *non-failed* nodes. This allows the aforementioned definition of a CA system as one that fails catastrophically if a network partition occurs: if the partition causes all nodes to fail, then the availability requirement does not apply to any nodes, and thus it is trivially satisfied, even if no node is able to respond to any requests. This definition is logically sound, but somewhat counter-intuitive.

2.2 Consistency

Consistency is also an overloaded word in data systems: consistency in the sense of ACID is a very different property from consistency in CAP [17]. In the distributed systems literature, consistency is usually understood as not one particular property, but as a spectrum of models with varying strengths of guarantee. Examples of such consistency models include linearizability [30], sequential consistency [38], causal consistency [4] and PRAM [42].

There is some similarity between consistency models and *transaction isolation models* such as serializability [15], snapshot isolation [14], repeatable read and read committed [3, 27]. Both describe restrictions on the values that operations may return, depending on other (prior or concurrent) operations. The difference is that transaction isolation models are usually formalized assuming a single replica, and operate at the granularity of transactions (each transaction may read or write multiple objects). Consistency models assume multiple replicas, but are usually defined in terms of single-object operations (not grouped into transactions). Bailis et al. [12] demonstrate a unified framework for reasoning about both distributed consistency and transaction isolation in terms of CAP.

2.2.1 The C in CAP

Fox and Brewer [24] define the C in CAP as one-copy serializability (1SR) [15], whereas Gilbert and Lynch [25] define it as linearizability. Those definitions are not identical, but fairly similar.[2] Both are

[2] Linearizability is a recency guarantee, whereas 1SR is not. 1SR requires isolated execution of multi-object transactions, which linearizability does not. Both require *coordination*, in the sense of section 4.4 [12].

safety properties [5], i.e. restrictions on the possible executions of the system, ensuring that certain situations never occur.

In the case of linearizability, the situation that may not occur is a *stale read*: stated informally, once a write operation has completed or some read operation has returned a new value, all following read operations must return the new value, until it is overwritten by another write operation. Gilbert and Lynch observe that if the write and read operations occur on different nodes, and those nodes cannot communicate during the time when those operations are being executed, then the safety property cannot be satisfied, because the read operation cannot know about the value written.

The C of CAP is sometimes referred to as *strong consistency* (a term that is not formally defined), and contrasted with *eventual consistency* [9, 49, 50], which is often regarded as the weakest level of consistency that is useful to applications. Eventual consistency means that if a system stops accepting writes and sufficient[3] communication occurs, then eventually all replicas will converge to the same value. However, as the aforementioned list of consistency models indicates, it is overly simplistic to cast 'strong' and eventual consistency as the only possible choices.

2.2.2 Probabilistic consistency

It is also possible to define consistency as a quantitative metric rather than a safety property. For example, Fox and Brewer [24] define *harvest* as "the fraction of the data reflected in the response, i.e. the completeness of the answer to the query," and probabilistically bounded staleness [11] studies the probability of a read returning a stale value, given various assumptions about the distribution of network latencies. However, these stochastic definitions of consistency are not the subject of CAP.

2.3 Partition Tolerance

A *network partition* has long been defined as a communication failure in which the network is split into disjoint sub-networks, with no communication possible across sub-networks [32]. This is a fairly narrow class of fault, but it does occur in practice [10], so it is worth studying.

[3] It is not clear what amount of communication is 'sufficient'. A possible formalization would be to require all replicas to converge to the same value within finite time, assuming fair-loss links (see section 2.3.2).

2.3.1 Assumptions about system model

It is less clear what *partition tolerance* means. Gilbert and Lynch [25] define a system as partition-tolerant if it continues to satisfy the consistency and availability properties in the presence of a partition. Fox and Brewer [24] define *partition-resilience* as "the system as whole can survive a partition between data replicas" (where *survive* is not defined).

At first glance, these definitions may seem redundant: if we say that an algorithm provides some guarantee (e.g. linearizability), then we expect *all* executions of the algorithm to satisfy that property, regardless of the faults that occur during the execution.

However, we can clarify the definitions by observing that the correctness of a distributed algorithm is always subject to assumptions about the faults that may occur during its execution. If you take an algorithm that assumes fair-loss links and crash-stop processes, and subject it to Byzantine faults, the execution will most likely violate safety properties that were supposedly guaranteed. These assumptions are typically encoded in a *system model*, and non-Byzantine system models rule out certain kinds of fault as impossible (so algorithms are not expected to tolerate them).

Thus, we can interpret *partition tolerance* as meaning "a network partition is among the faults that are assumed to be possible in the system." Note that this definition of partition tolerance is a statement about the system model, whereas consistency and availability are properties of the possible executions of an algorithm. It is misleading to say that an algorithm "provides partition tolerance," and it is better to say that an algorithm "assumes that partitions may occur."

If an algorithm assumes the absence of partitions, and is nevertheless subjected to a partition, it may violate its guarantees in arbitrarily undefined ways (including failing to respond even after the partition is healed, or deleting arbitrary amounts of data). Even though it may seem that such arbitrary failure semantics are not very useful, various systems exhibit such behavior in practice [35, 36]. Making networks highly reliable is very expensive [10], so most distributed programs must assume that partitions will occur sooner or later [28].

2.3.2 Partitions and fair-loss links

Further confusion arises due to the fact that network partitions are only one of a wide range of faults that can occur in distributed systems, including nodes failing or

restarting, nodes pausing for some amount of time (e.g. due to garbage collection), and loss or delay of messages in the network. Some faults can be modeled in terms of other faults (for example, Gilbert and Lynch state that the loss of an individual message can be modeled as a short-lived network partition).

In the design of distributed systems algorithms, a commonly assumed system model is *fair-loss links* [19]. A network link has the fair-loss property if the probability of a message *not* being lost is non-zero, i.e. the link sometimes delivers messages. The link may have intervals of time during which all messages are dropped, but those intervals must be of finite duration. On a fair-loss link, message delivery can be made reliable by retrying a message an unbounded number of times: the message is guaranteed to be eventually delivered after a finite number of attempts [19].

We argue that fair-loss links are a good model of most networks in practice: faults occur unpredictably; messages are lost while the fault is occurring; the fault lasts for some finite duration (perhaps seconds, perhaps hours), and eventually it is healed (perhaps after human intervention). There is no malicious actor in the network who can cause systematic message loss over unlimited periods of time – such malicious actors are usually only assumed in the design of Byzantine fault tolerant algorithms.

Is "partitions may occur" equivalent to assuming fair-loss links? Gilbert and Lynch [25] define partitions as "the network will be allowed to lose arbitrarily many messages sent from one node to another." In this definition it is unclear whether the number of lost messages is unbounded but finite, or whether it is potentially infinite.

Partitions of a finite duration are possible with fair-loss links, and thus an algorithm that is correct in a system model of fair-loss links can tolerate partitions of a finite duration. Partitions of an infinite duration require some further thought, as we shall see in section 3.

3 The CAP Proofs

In this section, we build upon the discussion of definitions in the last section, and examine the proofs of the theorems of Gilbert and Lynch [25]. We highlight some ambiguities in the reasoning of the proofs, and then suggest a more precise formalization.

3.1 Theorems 1 and 2

Gilbert and Lynch's Theorem 1 is stated as follows:

It is impossible in the asynchronous network model to implement a read/write data object that guarantees the following properties:

- *Availability*
- *Atomic consistency*[4]

in all fair executions (including those in which messages are lost).

Theorem 2 is similar, but specified in a system model with bounded network delay. The discussion in this section 3.1 applies to both theorems.

3.1.1 Availability of failed nodes

The first problem with this proof is the definition of availability. As discussed in section 2.1.3, only non-failing nodes are required to respond.

If it is possible for the algorithm to declare nodes as failed (e.g. if a node may crash itself), then the availability property can be trivially satisfied: all nodes can be crashed, and thus no node is required to respond. Of course, such an algorithm would not be useful in practice. Alternatively, if a minority of nodes is permanently partitioned from the majority, an algorithm could define the nodes in the minority partition as failed (by crashing them), while the majority partition continues implementing a linearizable register [7].

This is not the intention of CAP – the raison d'être of CAP is to characterize systems in which a minority partition can continue operating independently of the rest – but the present formalization of availability does not exclude such trivial solutions.

3.1.2 Finite and infinite partitions

Gilbert and Lynch's proofs of theorems 1 and 2 construct an execution of an algorithm *A* in which a write is followed by a read, while simultaneously a partition exists in the network. By showing that the execution is not linearizable, the authors derive a contradiction.

Note that this reasoning is only correct if we assume a system model in which partitions may have infinite duration.

If the system model is based on fair-loss links, then all partitions may be assumed to be of unbounded but

[4] In this context, *atomic consistency* is synonymous with linearizability, and it is unrelated to the A in ACID.

finite duration (section 2.3.2). Likewise, Gilbert and Lynch's availability property does not place any upper bound on the duration of an operation, as long as it is finite (section 2.1.2). Thus, if a linearizable algorithm encounters a network partition in a fair-loss system model, it is acceptable for the algorithm to simply wait for the partition to be healed: at any point in time, there is still hope that the partition will be healed in future, and so the availability property may yet be satisfied. For example, the ABD algorithm [7] can be used to implement a linearizable read-write register in an asynchronous network with fair-loss links.[5]

On the other hand, in an execution where a partition of infinite duration occurs, the algorithm is forced to make a choice between waiting until the partition heals (which never happens, thus violating availability) and exhibiting the execution in the proof of Theorem 1 (thus violating linearizability). We can conclude that Theorem 1 is only valid in a system model where infinite partitions are possible.

3.1.3 Linearizability vs. eventual consistency

Note that in the case of an infinite partition, no information can ever flow from one sub-network to the other. Thus, even eventual consistency (replica convergence in finite time, see section 2.2.1) is not possible in a system with an infinite partition.

Theorem 1 demonstrated that in a system model with infinite partitions, no algorithm exists which ensures linearizability and availability in all executions. However, we can also see that in the same system model, no algorithm exists which ensures eventual consistency in all executions.

The CAP theorem is often understood as demonstrating that linearizability cannot be achieved with high availability, whereas eventual consistency can. However, the results so far do not differentiate between linearizable and eventually consistent algorithms: both are possible if partitions are always finite, and both are impossible in a system model with infinite partitions.

To distinguish between linearizability and eventual consistency, a more careful formalization of CAP is required, which we give in section 3.2.

[5] ABD [7] is an algorithm for a single-writer multi-reader register. It was extended to the multi-writer case by Lynch and Shvartsman [44].

3.2 The partitionable system model

In this section we suggest a more precise formulation of CAP, and derive a result similar to Gilbert and Lynch's Theorem 1 and Corollary 1.1. This formulation will help us gain a better understanding of CAP and its consequences.

3.2.1 Definitions

Define a *partitionable link* to be a point-to-point link with the following properties:

1. *No duplication:* If a process p sends a message m once to process q, then m is delivered at most once by q.
2. *No creation:* If some process q delivers a message m with sender p, then m was previously sent to q by process p.

(A partitionable link is allowed to drop an infinite number of messages and cause unbounded message delay.)

Define the *partitionable model* as a system model in which processes can only communicate via partitionable links, in which processes never crash,[6] and in which every process has access to a local clock that is able to generate timeouts (the clock progresses monotonically at a rate approximately equal to real time, but clocks of different processes are not synchronized).

Define an execution E as *admissible* in a system model M if the processes and links in E satisfy the properties defined by M.

Define an algorithm A as *terminating* in a system model M if, for every execution E of A, if E is admissible in M, then every operation in E terminates in finite time.[7]

Define an execution E as *loss-free* if for every message m sent from p to q during E, m is eventually delivered to q. (There is no delay bound on delivery.)

An execution E is *partitioned* if it is not loss-free. Note: we may assume that links automatically resend lost messages an unbounded number of times. Thus, an execution in which messages are transiently lost during

[6] The assumption that processes never crash is of course unrealistic, but it makes the impossibility results in section 3.2.2 stronger. It also rules out the trivial solution of section 3.1.1.

[7] Our definition of *terminating* corresponds to Gilbert and Lynch's definition of *available*. We prefer to call it *terminating* because the word *available* is widely understood as referring to an empirical metric (see section 2.1). There is some similarity to *wait-free* data structures [29], although these usually assume reliable communication and unreliable processes.

some finite time period is not partitioned, because the links will eventually deliver all messages that were lost. An execution is only partitioned if the message loss persists forever.

For the definition of *linearizability* we refer to Herlihy and Wing [30].

There is no generally agreed formalization of *eventual consistency*, but the following corresponds to a liveness property that has been proposed [8, 9]: eventually, every read operation $read(q)$ at process q must return a set of all the values v ever written by any process p in an operation $write(p,v)$. For simplicity, we assume that values are never removed from the read set, although an application may only see one of the values (e.g. the one with the highest timestamp).

More formally, an infinite execution E is *eventually consistent* if, for all processes p and q, and for every value v such that operation $write(p,v)$ occurs in E, there are only finitely many operations in E such that $v \notin read(q)$.

3.2.2 Impossibility results

Assertion 1. *If an algorithm A implements a terminating read-write register R in the partitionable model, then there exists a loss-free execution of A in which R is not linearizable.*

Proof. Consider an execution E_1 in which the initial value of R is v_1, and no messages are delivered (all messages are lost, which is admissible for partitionable links). In E_1, p first performs an operation $write(p,v_2)$ where $v_2 \neq v_1$. This operation must terminate in finite time due to the termination property of A.

After the write operation terminates, q performs an operation $read(q)$, which must return v_1, since there is no way for q to know the value v_2, due to all messages being lost. The read must also terminate. This execution is not linearizable, because the read did not return v_2.

Now consider an execution E_2 which extends E_1 as follows: after the termination of the $read(q)$ operation, every message that was sent during E_1 is delivered (this is admissible for partitionable links). These deliveries cannot affect the execution of the write and read operations, since they occur after the termination of both operations, so E_2 is also non-linearizable. Moreover, E_2 is loss-free, since every message was delivered. □

Corollary 2. *There is no algorithm that implements a terminating read-write register in the partitionable model that is linearizable in all loss-free executions.*

This corresponds to Gilbert and Lynch's Corollary 1, and follows directly from the existence of a loss-free, non-linearizable execution (assertion 1).

Assertion 3. *There is no algorithm that implements a terminating read-write register in the partitionable model that is eventually consistent in all executions.*

Proof. Consider an execution E in which no messages are delivered, and in which process p performs operation $write(p,v)$. This write must terminate, since the algorithm is terminating. Process q with $p \neq q$ performs $read(q)$ infinitely many times. However, since no messages are delivered, q can never learn about the value written, so $read(q)$ never returns v. Thus, E is not eventually consistent. □

3.2.3 Opportunistic properties

Note that corollary 2 is about loss-free executions, whereas assertion 3 is about *all* executions. If we limit ourselves to loss-free executions, then eventual consistency *is* possible (e.g. by maintaining a replica at each process, and broadcasting every write to all processes).

However, everything we have discussed in this section pertains to the partitionable model, in which we cannot assume that all executions are loss-free. For clarity, we should specify the properties of an algorithm such that they hold for *all* admissible executions of a given system model, not only selected executions.

To this end, we can transform a property $\mathscr{P}$ into an *opportunistic* property $\mathscr{P}'$ such that:

$$\forall E : (E \models \mathscr{P}') \Leftrightarrow (lossfree(E) \Rightarrow (E \models \mathscr{P}))$$

or, equivalently:

$$\forall E : (E \models \mathscr{P}') \Leftrightarrow (partitioned(E) \vee (E \models \mathscr{P}))\,.$$

In other words, $\mathscr{P}'$ is trivially satisfied for executions that are partitioned. Requiring $\mathscr{P}'$ to hold for all executions is equivalent to requiring $\mathscr{P}$ to hold for all loss-free executions.

Hence we define an execution E as *opportunistically eventually consistent* if E is partitioned or if E is eventually consistent. (This is a weaker liveness property than eventual consistency.)

Similarly, we define an execution E as *opportunistically terminating linearizable* if E is partitioned, or if E is linearizable and every operation in E terminates in finite time.

From the results above, we can see that opportunistic terminating linearizability is impossible in the partitionable model (corollary 2), whereas opportunistic eventual consistency is possible. This distinction can be understood as the key result of CAP. However, it is arguably not a very interesting or insightful result.

3.3 Mismatch between formal model and practical systems

Many of the problems in this section are due to the fact that availability is defined by Gilbert and Lynch as a liveness property (section 2.1.2). Liveness properties make statements about something happening *eventually* in an infinite execution, which is confusing to practitioners, since real systems need to get things done in a finite (and usually short) amount of time.

Quoting Lamport [39]: "Liveness properties are inherently problematic. The question of whether a real system satisfies a liveness property is meaningless; it can be answered only by observing the system for an infinite length of time, and real systems don't run forever. Liveness is always an approximation to the property we really care about. We want a program to terminate within 100 years, but proving that it does would require the addition of distracting timing assumptions. So, we prove the weaker condition that the program eventually terminates. This doesn't prove that the program will terminate within our lifetimes, but it does demonstrate the absence of infinite loops."

Brewer [17] and some commercial database vendors [1] state that "all three properties [consistency, availability, and partition tolerance] are more continuous than binary". This is in direct contradiction to Gilbert and Lynch's formalization of CAP (and our restatement thereof), which expresses consistency and availability as safety and liveness properties of an algorithm, and partitions as a property of the system model. Such properties either hold or they do not hold; there is no degree of continuity in their definition.

Brewer's informal interpretation of CAP is intuitively appealing, but it is not a theorem, since it is not expressed formally (and thus cannot be proved or disproved) – it is, at best, a rule of thumb. Gilbert and Lynch's formalization can be proved correct, but it does not correspond to practitioners' intuitions for real systems. This contradiction suggests that although the formal model may be true, it is not useful.

4 Alternatives to CAP

In section 2 we explored the definitions of the terms *consistency*, *availability* and *partition tolerance*, and noted that a wide range of ambiguous and mutually incompatible interpretations have been proposed, leading to widespread confusion. Then, in section 3 we explored Gilbert and Lynch's definitions and proofs in more detail, and highlighted some problems with the formalization of CAP.

All of these misunderstandings and ambiguity lead us to asserting that CAP is no longer an appropriate tool for reasoning about systems. A better framework for describing trade-offs is required. Such a framework should be simple to understand, match most people's intuitions, and use definitions that are formal and correct.

In the rest of this paper we develop a first draft of an alternative framework called *delay-sensitivity*, which provides tools for reasoning about trade-offs between consistency and robustness to network faults. It is based on to several existing results from the distributed systems literature (most of which in fact predate CAP).

4.1 Latency and availability

As discussed in section 2.1.2, the latency (response time) of operations is often important in practice, but it is deliberately ignored by Gilbert and Lynch.

The problem with latency is that it is more difficult to model. Latency is influenced by many factors, especially the delay of packets on the network. Many computer networks (including Ethernet and the Internet) do not guarantee bounded delay, i.e. they allow packets to be delayed arbitrarily. Latencies and network delays are therefore typically described as probability distributions.

On the other hand, network delay can model a wide range of faults. In network protocols that automatically retransmit lost packets (such as TCP), transient packet loss manifests itself to the application as temporarily increased delay. Even when the period of packet loss exceeds the TCP connection timeout, application-level protocols often retry failed requests until they succeed, so the effective latency of the operation is the time from the first attempt until the successful completion. Even network partitions can be modelled as large packet delays (up to the duration of the partition), provided that the duration of the partition is finite and lost packets are retransmitted an unbounded number of times.

Abadi [2] argues that there is a trade-off between consistency and latency, which applies even when there is no network partition, and which is as important as the consistency/availability trade-off described by CAP. He proposes a "PACELC" formulation to reason about this trade-off.

We go further, and assert that availability should be modeled in terms of operation latency. For example, we could define the availability of a service as the proportion of requests that meet some latency bound (e.g. returning successfully within 500 ms, as defined by an SLA). This empirically-founded definition of availability closely matches our intuitive understanding.

We can then reason about a service's tolerance of network problems by analyzing how operation latency is affected by changes in network delay, and whether this pushes operation latency over the limit set by the SLA. If a service can sustain low operation latency, even as network delay increases dramatically, it is more tolerant of network problems than a service whose latency increases.

4.2 How operation latency depends on network delay

To find a replacement for CAP with a latency-centric viewpoint we need to examine how operation latency is affected by network latency at different levels of consistency. In practice, this depends on the algorithms and implementation of the particular software being used. However, CAP demonstrated that there is also interest in theoretical results identifying the fundamental limits of what can be achieved, regardless of the particular algorithm in use.

Several existing impossibility results establish lower bounds on the operation latency as a function of the network delay d. These results show that any algorithm guaranteeing a particular level of consistency cannot perform operations faster than some lower bound. We summarize these results in table 1 and in the following sections.

Our notation is similar to that used in complexity theory to describe the running time of an algorithm. However, rather than being a function of the size of input, we describe the latency of an operation as a function of network delay.

In this section we assume unbounded network delay, and unsynchronized clocks (i.e. each process has access to a clock that progresses monotonically at a rate approximately equal to real time, but the synchronization error between clocks is unbounded).

Consistency level	*write* latency	*read* latency
linearizability	$O(d)$	$O(d)$
sequential consistency	$O(d)$	$O(1)$
causal consistency	$O(1)$	$O(1)$

Table 1: Lowest possible operation latency at various consistency levels, as a function of network delay d.

4.2.1 Linearizability

Attiya and Welch [6] show that any algorithm implementing a linearizable read-write register must have an operation latency of at least $u/2$, where u is the uncertainty of delay in the network between replicas.[8]

In this proof, network delay is assumed to be at most d and at least $d-u$, so u is the difference between the minimum and maximum network delay. In many networks, the maximum possible delay (due to network congestion or retransmitting lost packets) is much greater than the minimum possible delay (due to the speed of light), so $u \approx d$. If network delay is unbounded, operation latency is also unbounded.

For the purposes of this survey, we can simplify the result to say that linearizability requires the latency of read and write operations to be proportional to the network delay d. This is indicated in table 1 as $O(d)$ latency for reads and writes. We call these operations *delay-sensitive*, as their latency is sensitive to changes in network delay.

4.2.2 Sequential consistency

Lipton and Sandberg [42] show that any algorithm implementing a sequentially consistent read-write register must have $|r| + |w| \geq d$, where $|r|$ is the latency of a read operation, $|w|$ is the latency of a write operation, and d is the network delay. Mavronicolas and Roth [46] further develop this result.

This lower bound provides a degree of choice for the application: for example, an application that performs

[8] Attiya and Welch [6] originally proved a bound of $u/2$ for write operations (assuming two writer processes and one reader), and a bound of $u/4$ for read operations (two readers, one writer). The $u/2$ bound for read operations is due to Mavronicolas and Roth [46].

more reads than writes can reduce the average operation latency by choosing $|r| = 0$ and $|w| \geq d$, whereas a write-heavy application might choose $|r| \geq d$ and $|w| = 0$. Attiya and Welch [6] describe algorithms for both of these cases (the $|r| = 0$ case is similar to the Zab algorithm used by Apache ZooKeeper [33]).

Choosing $|r| = 0$ or $|w| = 0$ means the operation can complete without waiting for any network communication (it may still send messages, but need not wait for a response from other nodes). The latency of such an operation thus only depends on the local database algorithms: it might be constant-time $O(1)$, or it might be $O(\log n)$ where n is the size of the database, but either way it is independent of the network delay d, so we call it *delay-independent*.

In table 1, sequential consistency is described as having fast reads and slow writes (constant-time reads, and write latency proportional to network delay), although these roles can be swapped if an application prefers fast writes and slow reads.

4.2.3 Causal consistency

If sequential consistency allows the latency of *some* operations to be independent of network delay, which level of consistency allows *all* operation latencies to be independent of the network? Recent results [8, 45] show that causal consistency [4] with eventual convergence is the strongest possible consistency guarantee with this property.[9]

Read Your Writes [49], PRAM [42] and other weak consistency models (all the way down to eventual consistency, which provides no safety property [9]) are weaker than causal consistency, and thus achievable without waiting for the network.

If tolerance of network delay is the only consideration, causal consistency is the optimal consistency level. There may be other reasons for choosing weaker consistency levels (for example, the metadata overhead of tracking causality [8, 20]), but these trade-offs are outside of the scope of this discussion, as they are also outside the scope of CAP.

[9] There are a few variants of causal consistency, such as *real time causal* [45], *causal+* [43] and *observable causal* [8] consistency. They have subtle differences, but we do not have space in this paper to compare them in detail.

4.3 Heterogeneous delays

A limitation of the results in section 4.2 is that they assume the distribution of network delays is the same between every pair of nodes. This assumption is not true in general: for example, network delay between nodes in the same datacenter is likely to be much lower than between geographically distributed nodes communicating over WAN links.

If we model network faults as periods of increased network delay (section 4.1), then a network partition is a situation in which the delay between nodes within each partition remains small, while the delay across partitions increases dramatically (up to the duration of the partition).

For $O(d)$ algorithms, which of these different delays do we need to assume for d? The answer depends on the communication pattern of the algorithm.

4.3.1 Modeling network topology

For example, a replication algorithm that uses a single leader or primary node requires all write requests to contact the primary, and thus d in this case is the network delay between the client and the leader (possibly via other nodes). In a geographically distributed system, if client and leader are in different locations, d includes WAN links. If the client is temporarily partitioned from the leader, d increases up to the duration of the partition.

By contrast, the ABD algorithm [7] waits for responses from a majority of replicas, so d is the largest among the majority of replicas that are fastest to respond. If a minority of replicas is temporarily partitioned from the client, the operation latency remains independent of the partition duration.

Another possibility is to treat network delay within the same datacenter, d_{local}, differently from network delay over WAN links, d_{remote}, because usually $d_{local} \ll d_{remote}$. Systems such as COPS [43], which place a leader in each datacenter, provide linearizable operations within one datacenter (requiring $O(d_{local})$ latency), and causal consistency across datacenters (making the request latency independent of d_{remote}).

4.4 Delay-independent operations

The big-O notation for operation latency ignores constant factors (such as the number of network round-trips required by an algorithm), but it captures the essence of

what we need to know for building systems that can tolerate network faults: what happens if network delay dramatically degrades? In a delay-sensitive $O(d)$ algorithm, operation latency may increase to be as large as the duration of the network interruption (i.e. minutes or even hours), whereas a delay-independent $O(1)$ algorithm remains unaffected.

If the SLA calls for operation latencies that are significantly shorter than the expected duration of network interruptions, delay-independent algorithms are required. In such algorithms, the time until replica convergence is still proportional to d, but convergence is decoupled from operation latency. Put another way, delay-independent algorithms support *disconnected* or *offline operation*. Disconnected operation has long been used in network file systems [37] and automatic teller machines [18].

For example, consider a calendar application running on a mobile device: a user may travel through a tunnel or to a remote location where there is no cellular network coverage. For a mobile device, regular network interruptions are expected, and they may last for days. During this time, the user should still be able to interact with the calendar app, checking their schedule and adding events (with any changes asynchronously propagated when an internet connection is next available).

However, even in environments with fast and reliable network connectivity, delay-independent algorithms have been shown to have performance and scalability benefits: in this context, they are known as *coordination-free* [13] or *ALPS* [43] systems. Many popular database integrity constraints can be implemented without synchronous coordination between replicas [13].

4.5 Proposed terminology

Much of the confusion around CAP is due to the ambiguous, counter-intuitive and contradictory definitions of terms such as *availability*, as discussed in section 2. In order to improve the situation and reduce misunderstandings, there is a need to standardize terminology with simple, formal and correct definitions that match the intuitions of practitioners.

Building upon the observations above and the results cited in section 4.2, we propose the following definitions as a first draft of a *delay-sensitivity framework* for reasoning about consistency and availability trade-offs. These definitions are informal and intended as a starting point for further discussion.

Availability is an *empirical metric*, not a property of an algorithm. It is defined as the percentage of successful requests (returning a non-error response within a predefined latency bound) over some period of system operation.

Delay-sensitive describes algorithms or operations that need to wait for network communication to complete, i.e. which have latency proportional to network delay. The opposite is *delay-independent*. Systems must specify the nature of the sensitivity (e.g. an operation may be sensitive to intra-datacenter delay but independent of inter-datacenter delay). A fully delay-independent system supports *disconnected (offline) operation*.

Network faults encompass packet loss (both transient and long-lasting) and unusually large packet delay. *Network partitions* are just one particular type of network fault; in most cases, systems should plan for all kinds of network fault, and not only partitions. As long as lost packets or failed requests are retried, they can be modeled as large network delay.

Fault tolerance is used in preference to *high availability* or *partition tolerance*. The maximum fault that can be tolerated must be specified (e.g. "the algorithm can tolerate up to a minority of replicas crashing or disconnecting"), and the description must also state what happens if more faults occur than the system can tolerate (e.g. all requests return an error, or a consistency property is violated).

Consistency refers to a spectrum of different *consistency models* (including linearizability and causal consistency), not one particular consistency model. When a particular consistency model such as linearizability is intended, it is referred to by its usual name. The term *strong consistency* is vague, and may refer to linearizability, sequential consistency or one-copy serializability.

5 Conclusion

In this paper we discussed several problems with the CAP theorem: the definitions of consistency, availability and partition tolerance in the literature are somewhat contradictory and counter-intuitive, and the distinction that CAP draws between "strong" and "eventual" consistency models is less clear than widely believed.

CAP has nevertheless been very influential in the design of distributed data systems. It deserves credit for catalyzing the exploration of the design space of systems with weak consistency guarantees, e.g. in the NoSQL movement. However, we believe that CAP has now reached the end of its usefulness; we recommend that it should be relegated to the history of distributed systems, and no longer be used for justifying design decisions.

As an alternative to CAP, we propose a simple *delay-sensitivity* framework for reasoning about trade-offs between consistency guarantees and tolerance of network faults in a replicated database. Every operation is categorized as either $O(d)$, if its latency is sensitive to network delay, or $O(1)$, if it is independent of network delay. On the assumption that lost messages are retransmitted an unbounded number of times, we can model network faults (including partitions) as periods of greatly increased delay. The algorithm's sensitivity to network delay determines whether the system can still meet its service level agreement (SLA) when a network fault occurs.

The actual sensitivity of a system to network delay depends on its implementation, but – in keeping with the goal of CAP – we can prove that certain levels of consistency cannot be achieved without making operation latency proportional to network delay. These theoretical lower bounds are summarized in Table 1. We have not proved any new results in this paper, but merely drawn on existing distributed systems research dating mostly from the 1990s (and thus predating CAP).

For future work, it would be interesting to model the *probability distribution* of latencies for different concurrency control and replication algorithms (e.g. by extending PBS [11]), rather than modeling network delay as just a single number d. It would also be interesting to model the network communication topology of distributed algorithms more explicitly.

We hope that by being more rigorous about the implications of different consistency levels on performance and fault tolerance, we can encourage designers of distributed data systems to continue the exploration of the design space. And we also hope that by adopting simple, correct and intuitive terminology, we can help guide application developers towards the storage technologies that are most appropriate for their use cases.

Acknowledgements

Many thanks to Niklas Ekström, Seth Gilbert and Henry Robinson for fruitful discussions around this topic.

References

[1] ACID support in Aerospike. Aerospike, Inc., June 2014. URL `http://www.aerospike.com/docs/architecture/assets/AerospikeACIDSupport.pdf`.

[2] Daniel J Abadi. Consistency tradeoffs in modern distributed database system design. *IEEE Computer Magazine*, 45(2):37–42, February 2012. doi:10.1109/MC.2012.33.

[3] Atul Adya. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions*. PhD thesis, Massachusetts Institute of Technology, March 1999.

[4] Mustaque Ahamad, Gil Neiger, James E Burns, Prince Kohli, and Phillip W Hutto. Causal memory: definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, March 1995. doi:10.1007/BF01784241.

[5] Bowen Alpern and Fred B Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985. doi:10.1016/0020-0190(85)90056-0.

[6] Hagit Attiya and Jennifer L Welch. Sequential consistency versus linearizability. *ACM Transactions on Computer Systems (TOCS)*, 12(2):91–122, May 1994. doi:10.1145/176575.176576.

[7] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM*, 42(1):124–142, January 1995. doi:10.1145/200836.200869.

[8] Hagit Attiya, Faith Ellen, and Adam Morrison. Limitations of highly-available eventually-consistent data stores. In *ACM Symposium on Principles of Distributed Computing (PODC)*, July 2015. doi:10.1145/2767386.2767419.

[9] Peter Bailis and Ali Ghodsi. Eventual consistency today: Limitations, extensions, and beyond. *ACM Queue*, 11(3), March 2013. doi:10.1145/2460276.2462076.

[10] Peter Bailis and Kyle Kingsbury. The network is reliable. *ACM Queue*, 12(7), July 2014. doi:10.1145/2639988.2639988.

[11] Peter Bailis, Shivaram Venkataraman, Michael J Franklin, Joseph M Hellerstein, and Ion Stoica. Probabilistically bounded staleness for practical partial quorums. *Proceedings of the VLDB Endowment*, 5(8):776–787, April 2012.

[12] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. Highly available transactions: Virtues and limitations. In *40th International Conference on Very Large Data Bases (VLDB)*, September 2014.

[13] Peter Bailis, Alan Fekete, Michael J Franklin, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. Coordination-avoiding database systems. *Proceedings of the VLDB Endowment*, 8(3): 185–196, November 2014.

[14] Hal Berenson, Philip A Bernstein, Jim N Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ANSI SQL isolation levels. In *ACM International Conference on Management of Data (SIGMOD)*, May 1995. doi:10.1145/568271.223785.

[15] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987. ISBN 0201107155. URL `http://research.microsoft.com/en-us/people/philbe/ccontrol.aspx`.

[16] Eric A Brewer. Towards robust distributed systems (keynote). In *19th ACM Symposium on Principles of Distributed Computing (PODC)*, July 2000.

[17] Eric A Brewer. CAP twelve years later: How the "rules" have changed. *IEEE Computer Magazine*, 45(2):23–29, February 2012. doi:10.1109/MC.2012.37.

[18] Eric A Brewer. NoSQL: Past, present, future. In *QCon San Francisco*, November 2012. URL `http://www.infoq.com/presentations/NoSQL-History`.

[19] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to Reliable and Secure Distributed Programming*. Springer, 2nd edition, February 2011. ISBN 978-3-642-15259-7.

[20] Bernadette Charron-Bost. Concerning the size of logical clocks in distributed systems. *Information Processing Letters*, 39(1): 11–16, July 1991. doi:10.1016/0020-0190(91)90055-M.

[21] Jeff Darcy. When partitions attack, October 2010. URL `http://pl.atyp.us/wordpress/index.php/2010/10/when-partitions-attack/`.

[22] Susan B Davidson, Hector Garcia-Molina, and Dale Skeen. Consistency in partitioned networks. *ACM Computing Surveys*, 17(3):341–370, September 1985. doi:10.1145/5505.5508.

[23] Michael J Fischer and Alan Michael. Sacrificing serializability to attain high availability of data in an unreliable network. In *1st ACM Symposium on Principles of Database Systems (PODS)*, pages 70–75, March 1982. doi:10.1145/588111.588124.

[24] Armando Fox and Eric A Brewer. Harvest, yield, and scalable tolerant systems. In *7th Workshop on Hot Topics in Operating Systems (HotOS)*, pages 174–178, March 1999. doi:10.1109/HOTOS.1999.798396.

[25] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51–59, 2002. doi:10.1145/564585.564601.

[26] Seth Gilbert and Nancy Lynch. Perspectives on the CAP theorem. *IEEE Computer Magazine*, 45(2):30–36, February 2012. doi:10.1109/MC.2011.389.

[27] Jim N Gray, Raymond A Lorie, Gianfranco R Putzolu, and Irving L Traiger. Granularity of locks and degrees of consistency in a shared data base. In G M Nijssen, editor, *Modelling in Data Base Management Systems: Proceedings of the IFIP Working Conference on Modelling in Data Base Management Systems*, pages 364–394. Elsevier/North Holland Publishing, 1976.

[28] Coda Hale. You can't sacrifice partition tolerance, October 2010. URL `http://codahale.com/you-cant-sacrifice-partition-tolerance/`.

[29] Maurice P Herlihy. Impossibility and universality results for wait-free synchronization. In *7th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 276–290, August 1988. doi:10.1145/62546.62593.

[30] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990. doi:10.1145/78969.78972.

[31] Jeff Hodges. Notes on distributed systems for young bloods, January 2013. URL `http://www.somethingsimilar.com/2013/01/14/notes-on-distributed-systems-for-young-bloods/`.

[32] Paul R Johnson and Robert H Thomas. RFC 677: The maintenance of duplicate databases. Network Working Group, January 1975. URL `https://tools.ietf.org/html/rfc677`.

[33] Flavio P Junqueira, Benjamin C Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *41st IEEE International Conference on Dependable Systems and Networks (DSN)*, pages 245–256, June 2011. doi:10.1109/DSN.2011.5958223.

[34] Won Kim. Highly available systems for database application. *ACM Computing Surveys*, 16(1):71–98, March 1984. doi:10.1145/861.866.

[35] Kyle Kingsbury. Call me maybe: RabbitMQ, June 2014. URL `https://aphyr.com/posts/315-call-me-maybe-rabbitmq`.

[36] Kyle Kingsbury. Call me maybe: Elasticsearch 1.5.0, April 2015. URL `https://aphyr.com/posts/323-call-me-maybe-elasticsearch-1-5-0`.

[37] James J Kistler and M Satyanarayanan. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1):3–25, February 1992. doi:10.1145/146941.146942.

[38] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, September 1979. doi:10.1109/TC.1979.1675439.

[39] Leslie Lamport. Fairness and hyperfairness. *Distributed Computing*, 13(4):239–245, November 2000. doi:10.1007/PL00008921.

[40] Bruce G Lindsay, Patricia Griffiths Selinger, C Galtieri, Jim N Gray, Raymond A Lorie, Thomas G Price, Gianfranco R Putzolu, Irving L Traiger, and Bradford W Wade. Notes on distributed databases. Technical Report RJ2571(33471), IBM Research, July 1979.

[41] Nicolas Liochon. You do it too: Forfeiting network partition tolerance in distributed systems, July 2015. URL `http://blog.thislongrun.com/2015/07/Forfeit-Partition-Tolerance-Distributed-System-CAP-Theorem.html`.

[42] Richard J Lipton and Jonathan S Sandberg. PRAM: A scalable shared memory. Technical Report CS-TR-180-88, Princeton University Department of Computer Science, September 1988.

[43] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *23rd ACM Symposium on Operating Systems Principles (SOSP)*, pages 401–416, October 2011. doi:10.1145/2043556.2043593.

[44] Nancy Lynch and Alex Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *27th Annual International Symposium on Fault-Tolerant Computing (FTCS)*, pages 272–281, June 1997. doi:10.1109/FTCS.1997.614100.

[45] Prince Mahajan, Lorenzo Alvisi, and Mike Dahlin. Consistency, availability, and convergence. Technical Report UTCS TR-11-22, University of Texas at Austin, Department of Computer Science, May 2011.

[46] Marios Mavronicolas and Dan Roth. Linearizable read/write objects. *Theoretical Computer Science*, 220(1):267–319, June 1999. doi:10.1016/S0304-3975(98)90244-4.

[47] Henry Robinson. CAP confusion: Problems with 'partition tolerance', April 2010. URL `http://blog.cloudera.com/blog/2010/04/cap-confusion-problems-with-partition-tolerance/`.

[48] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O Myreen. x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM*, 53(7):89–97, July 2010. doi:10.1145/1785414.1785443.

[49] Douglas B Terry, Alan J Demers, Karin Petersen, Mike J Spreitzer, Marvin M Theimer, and Brent B Welch. Session guarantees for weakly consistent replicated data. In *3rd International Conference on Parallel and Distributed Information Systems (PDIS)*, pages 140–149, September 1994. doi:10.1109/PDIS.1994.331722.

[50] Werner Vogels. Eventually consistent. *ACM Queue*, 6(6):14–19, October 2008. doi:10.1145/1466443.1466448.