infoq.com

# CAP Twelve Years Later: How the "Rules" Have Changed

*Eric Brewer*

33-42 minutes

---


*This article first appeared in Computer magazine and is brought to you by InfoQ & IEEE Computer Society.*

The CAP theorem asserts that any networked shared-data system can have only two of three desirable properties. However, by explicitly handling partitions, designers can optimize consistency and availability, thereby achieving some trade-off of all three.

In the decade since its introduction, designers and researchers have used (and sometimes abused) the CAP theorem as a reason to explore a wide variety of novel distributed systems. The NoSQL movement also has applied it as an argument against traditional databases.



The CAP theorem states that any networked shared-data system

can have at most two of three desirable properties:

- consistency (C) equivalent to having a single up-to-date copy of the data;
- high availability (A) of that data (for updates); and
- tolerance to network partitions (P).

This expression of CAP served its purpose, which was to open the minds of designers to a wider range of systems and tradeoffs; indeed, in the past decade, a vast range of new systems has emerged, as well as much debate on the relative merits of consistency and availability. The "2 of 3" formulation was always misleading because it tended to oversimplify the tensions among properties. Now such nuances matter. CAP prohibits only a tiny part of the design space: perfect availability and consistency in the presence of partitions, which are rare.

Although designers still need to choose between consistency and availability when partitions are present, there is an incredible range of flexibility for handling partitions and recovering from them. The modern CAP goal should be to maximize combinations of consistency and availability that make sense for the specific application. Such an approach incorporates plans for operation during a partition and for recovery afterward, thus helping designers think about CAP beyond its historically perceived limitations.

## Why "2 of 3" is missleading

The easiest way to understand CAP is to think of two nodes on opposite sides of a partition. Allowing at least one node to update state will cause the nodes to become inconsistent, thus forfeiting C. Likewise, if the choice is to preserve consistency, one side of the partition must act as if it is unavailable, thus forfeiting A. Only when nodes communicate is it possible to preserve both consistency and availability, thereby forfeiting P. The general belief is that for wide-

area systems, designers cannot forfeit P and therefore have a difficult choice between C and A. In some sense, the NoSQL movement is about creating choices that focus on availability first and consistency second; databases that adhere to ACID properties (atomicity, consistency, isolation, and durability) do the opposite. The "ACID, BASE, and CAP" sidebar explains this difference in more detail.

In fact, this exact discussion led to the CAP theorem. In the mid-1990s, my colleagues and I were building a variety of cluster-based wide-area systems (essentially early cloud computing), including search engines, proxy caches, and content distribution systems.[1] Because of both revenue goals and contract specifications, system availability was at a premium, so we found ourselves regularly choosing to optimize availability through strategies such as employing caches or logging updates for later reconciliation. Although these strategies did increase availability, the gain came at the cost of decreased consistency.

The first version of this consistency-versus-availability argument appeared as ACID versus BASE,[2] which was not well received at the time, primarily because people love the ACID properties and are hesitant to give them up. The CAP theorem's aim was to justify the need to explore a wider design space-hence the "2 of 3" formulation. The theorem first appeared in fall 1998. It was published in 1999[3] and in the keynote address at the 2000 Symposium on Principles of Distributed Computing,[4] which led to its proof.

As the "CAP Confusion" sidebar explains, the "2 of 3" view is misleading on several fronts. First, because partitions are rare, there is little reason to forfeit C or A when the system is not partitioned. Second, the choice between C and A can occur many times within the same system at very fine granularity; not only can

subsystems make different choices, but the choice can change according to the operation or even the specific data or user involved. Finally, all three properties are more continuous than binary. Availability is obviously continuous from 0 to 100 percent, but there are also many levels of consistency, and even partitions have nuances, including disagreement within the system about whether a partition exists.

Exploring these nuances requires pushing the traditional way of dealing with partitions, which is the fundamental challenge. Because partitions are rare, CAP should allow perfect C and A most of the time, but when partitions are present or perceived, a strategy that detects partitions and explicitly accounts for them is in order. This strategy should have three steps: detect partitions, enter an explicit partition mode that can limit some operations, and initiate a recovery process to restore consistency and compensate for mistakes made during a partition.

## Acid, base, and cap

ACID and BASE represent two design philosophies at opposite ends of the consistency-availability spectrum. The ACID properties focus on consistency and are the traditional approach of databases. My colleagues and I created BASE in the late 1990s to capture the emerging design approaches for high availability and to make explicit both the choice and the spectrum. Modern large-scale wide-area systems, including the cloud, use a mix of both approaches.

Although both terms are more mnemonic than precise, the BASE acronym (being second) is a bit more awkward: Basically Available, Soft state, Eventually consistent. Soft state and eventual consistency are techniques that work well in the presence of partitions and thus promote availability.

The relationship between CAP and ACID is more complex and

often misunderstood, in part because the C and A in ACID represent different concepts than the same letters in CAP and in part because choosing availability affects only some of the ACID guarantees. The four ACID properties are:

**Atomicity (A).** All systems benefit from atomic operations. When the focus is availability, both sides of a partition should still use atomic operations. Moreover, higher-level atomic operations (the kind that ACID implies) actually simplify recovery.

**Consistency (C).** In ACID, the C means that a transaction preserves all the database rules, such as unique keys. In contrast, the C in CAP refers only to single]copy consistency, a strict subset of ACID consistency. ACID consistency also cannot be maintained across partitions.partition recovery will need to restore ACID consistency. More generally, maintaining invariants during partitions might be impossible, thus the need for careful thought about which operations to disallow and how to restore invariants during recovery.

**Isolation (I).** Isolation is at the core of the CAP theorem: if the system requires ACID isolation, it can operate on at most one side during a partition. Serializability requires communication in general and thus fails across partitions. Weaker definitions of correctness are viable across partitions via compensation during partition recovery.

**Durability (D).** As with atomicity, there is no reason to forfeit durability, although the developer might choose to avoid needing it via soft state (in the style of BASE) due to its expense. A subtle point is that, during partition recovery, it is possible to reverse durable operations that unknowingly violated an invariant during the operation. However, at the time of recovery, given a durable history from both sides, such operations can be detected and corrected. In general, running ACID transactions on each side of a partition

makes recovery easier and enables a framework for compensating transactions that can be used for recovery from a partition.

## Cap-latency connection

In its classic interpretation, the CAP theorem ignores latency, although in practice, latency and partitions are deeply related. Operationally, the essence of CAP takes place during a timeout, a period when the program must make a fundamental decision-the *partition decision*:

- cancel the operation and thus decrease availability, *or*
- proceed with the operation and thus risk inconsistency.

Retrying communication to achieve consistency, for example, via Paxos or a two-phase commit, just delays the decision. At some point the program must make the decision; retrying communication indefinitely is in essence choosing C over A.

Thus, pragmatically, a partition is a time bound on communication. Failing to achieve consistency within the time bound implies a partition and thus a choice between C and A for this operation. These concepts capture the core design issue with regard to latency: are two sides moving forward without communication?

This pragmatic view gives rise to several important consequences. The first is that there is no global notion of a partition, since some nodes might detect a partition, and others might not. The second consequence is that nodes can detect a partition and enter a *partition mode*-a central part of optimizing C and A.

Finally, this view means that designers can set time bounds intentionally according to target response times; systems with tighter bounds will likely enter partition mode more often and at times when the network is merely slow and not actually partitioned.

Sometimes it makes sense to forfeit strong C to avoid the high

latency of maintaining consistency over a wide area. Yahoo's PNUTS system incurs inconsistency by maintaining remote copies asynchronously.[5] However, it makes the master copy local, which decreases latency. This strategy works well in practice because single user data is naturally partitioned according to the user's (normal) location. Ideally, each user's data master is nearby.

Facebook uses the opposite strategy:[6] the master copy is always in one location, so a remote user typically has a closer but potentially stale copy. However, when users update their pages, the update goes to the master copy directly as do all the user's reads for a short time, despite higher latency. After 20 seconds, the user's traffic reverts to the closer copy, which by that time should reflect the update.

## Cap confusion

Aspects of the CAP theorem are often misunderstood, particularly the scope of availability and consistency, which can lead to undesirable results. If users cannot reach the service at all, there is no choice between C and A except when part of the service runs on the client. This exception, commonly known as disconnected operation or offline mode,[7] is becoming increasingly important. Some HTML5 features-in particular, on-client persistent storage-make disconnected operation easier going forward. These systems normally choose A over C and thus must recover from long partitions.

Scope of consistency reflects the idea that, within some boundary, state is consistent, but outside that boundary all bets are off. For example, within a primary partition, it is possible to ensure complete consistency and availability, while outside the partition, service is not available. Paxos and atomic multicast systems typically match this scenario.[8] In Google, the primary partition usually resides

within one datacenter; however, Paxos is used on the wide area to ensure global consensus, as in Chubby,[9] and highly available durable storage, as in Megastore.[10]

Independent, self-consistent subsets can make forward progress while partitioned, although it is not possible to ensure global invariants. For example, with sharding, in which designers prepartition data across nodes, it is highly likely that each shard can make some progress during a partition. Conversely, if the relevant state is split across a partition or global invariants are necessary, then at best only one side can make progress and at worst no progress is possible.

Does choosing consistency and availability (CA) as the "2 of 3" make sense? As some researchers correctly point out, exactly what it means to forfeit P is unclear.[11,12] Can a designer choose not to have partitions? If the choice is CA, and then there is a partition, the choice must revert to C or A. It is best to think about this probabilistically: choosing CA should mean that the probability of a partition is far less than that of other systemic failures, such as disasters or multiple simultaneous faults.

Such a view makes sense because real systems lose both C and A under some sets of faults, so all three properties are a matter of degree. In practice, most groups assume that a datacenter (single site) has no partitions within, and thus design for CA within a single site; such designs, including traditional databases, are the pre-CAP default. However, although partitions are less likely within a datacenter, they are indeed possible, which makes a CA goal problematic. Finally, given the high latency across the wide area, it is relatively common to forfeit perfect consistency across the wide area for better performance.

Another aspect of CAP confusion is the hidden cost of forfeiting consistency, which is the need to know the system’s invariants. The

subtle beauty of a consistent system is that the invariants tend to hold even when the designer does not know what they are. Consequently, a wide range of reasonable invariants will work just fine. Conversely, when designers choose A, which requires restoring invariants after a partition, they must be explicit about all the invariants, which is both challenging and prone to error. At the core, this is the same concurrent updates problem that makes multithreading harder than sequential programming.

## Managing partitions

The challenging case for designers is to mitigate a partition's effects on consistency and availability. The key idea is to manage partitions very explicitly, including not only detection, but also a specific recovery process and a plan for all of the invariants that might be violated during a partition. This management approach has three steps:

(Click on the image to enlarge it)

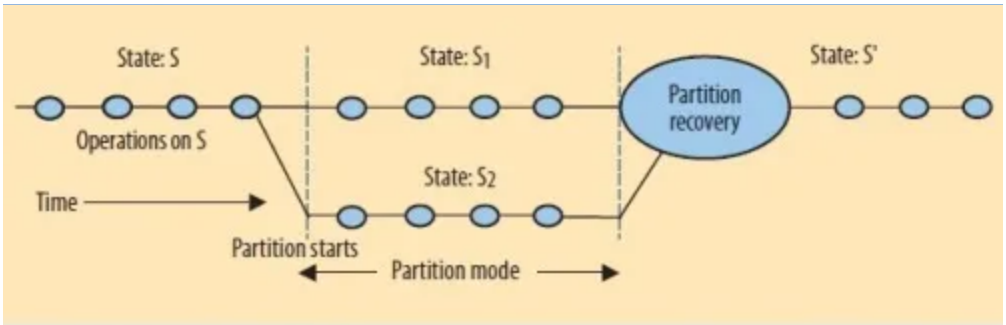


**Figure 1.** The state starts out consistent and remains so until a partition starts. To stay available, both sides enter partition mode and continue to execute operations, creating concurrent states $S_1$ and $S_2$, which are inconsistent. When the partition ends, the truth becomes clear and partition recovery starts. During recovery, the system merges $S_1$ and $S_2$ into a consistent state S' and also compensates for any mistakes made during the partition.

- detect the start of a partition,
- enter an explicit partition mode that may limit some operations, and
- initiate partition recovery when communication is restored.

The last step aims to restore consistency and compensate for

mistakes the program made while the system was partitioned.

Figure 1 shows a partition's evolution. Normal operation is a sequence of atomic operations, and thus partitions always start between operations. Once the system times out, it detects a partition, and the detecting side enters partition mode. If a partition does indeed exist, both sides enter this mode, but one-sided partitions are possible. In such cases, the other side communicates as needed and either this side responds correctly or no communication was required; either way, operations remain consistent. However, because the detecting side could have inconsistent operations, it must enter partition mode. Systems that use a quorum are an example of this one-sided partitioning. One side will have a quorum and can proceed, but the other cannot. Systems that support disconnected operation clearly have a notion of partition mode, as do some atomic multicast systems, such as Java's JGroups.

Once the system enters partition mode, two strategies are possible. The first is to limit some operations, thereby reducing availability. The second is to record extra information about the operations that will be helpful during partition recovery. Continuing to attempt communication will enable the system to discern when the partition ends.

## Which operations should proceed?

Deciding which operations to limit depends primarily on the invariants that the system must maintain. Given a set of invariants, the designer must decide whether or not to maintain a particular invariant during partition mode or risk violating it with the intent of restoring it during recovery. For example, for the invariant that keys in a table are unique, designers typically decide to risk that invariant and allow duplicate keys during a partition. Duplicate keys are easy to detect during recovery, and, assuming that they can be merged,

the designer can easily restore the invariant.

For an invariant that must be maintained during a partition, however, the designer must prohibit or modify operations that might violate it. (In general, there is no way to tell if the operation will actually violate the invariant, since the state of the other side is not knowable.) Externalized events, such as charging a credit card, often work this way. In this case, the strategy is to record the intent and execute it after the recovery. Such transactions are typically part of a larger workflow that has an explicit order-processing state, and there is little downside to delaying the operation until the partition ends. The designer forfeits A in a way that users do not see. The users know only that they placed an order and that the system will execute it later.

More generally, partition mode gives rise to a fundamental user-interface challenge, which is to communicate that tasks are in progress but not complete. Researchers have explored this problem in some detail for disconnected operation, which is just a long partition. Bayou's calendar application, for example, shows potentially inconsistent (tentative) entries in a different color.[13] Such notifications are regularly visible both in workflow applications, such as commerce with e-mail notifications, and in cloud services with an offline mode, such as Google Docs.

One reason to focus on explicit atomic operations, rather than just reads and writes, is that it is vastly easier to analyze the impact of higher-level operations on invariants. Essentially, the designer must build a table that looks at the cross product of all operations and all invariants and decide for each entry if that operation could violate the invariant. If so, the designer must decide whether to prohibit, delay, or modify the operation. In practice, these decisions can also depend on the known state, on the arguments, or on both. For example, in systems with a home node for certain data, 5 operations can typically proceed on the home node but not on other

nodes.

The best way to track the history of operations on both sides is to use version vectors, which capture the causal dependencies among operations. The vector's elements are a pair (node, logical time), with one entry for every node that has updated the object and the time of its last update. Given two versions of an object, A and B, A is newer than B if, for every node in common in their vectors, A's times are greater than or equal to B's and at least one of A's times is greater.

If it is impossible to order the vectors, then the updates were concurrent and possibly inconsistent. Thus, given the version vector history of both sides, the system can easily tell which operations are already in a known order and which executed concurrently. Recent work[14] proved that this kind of causal consistency is the best possible outcome in general if the designer chooses to focus on availability.

## Partition recovery

At some point, communication resumes and the partition ends. During the partition, each side was available and thus making forward progress, but partitioning has delayed some operations and violated some invariants. At this point, the system knows the state and history of both sides because it kept a careful log during partition mode. The state is less useful than the history, from which the system can deduce which operations actually violated invariants and what results were externalized, including the responses sent to the user. The designer must solve two hard problems during recovery:

- the state on both sides must become consistent, and
- there must be compensation for the mistakes made during partition mode.

It is generally easier to fix the current state by starting from the state at the time of the partition and rolling forward both sets of operations in some manner, maintaining consistent state along the way. Bayou did this explicitly by rolling back the database to a correct time and replaying the full set of operations in a well-defined, deterministic order so that all nodes reached the same state.[15] Similarly, source-code control systems such as the Concurrent Versioning System (CVS) start from a shared consistent point and roll forward updates to merge branches.

Most systems cannot always merge conflicts. For example, CVS occasionally has conflicts that the user must resolve manually, and wiki systems with offline mode typically leave conflicts in the resulting document that require manual editing.[16]

Conversely, some systems can always merge conflicts by choosing certain operations. A case in point is text editing in Google Docs,[17] which limits operations to applying a style and adding or deleting text. Thus, although the general problem of conflict resolution is not solvable, in practice, designers can choose to constrain the use of certain operations during partitioning so that the system can automatically merge state during recovery. Delaying risky operations is one relatively easy implementation of this strategy.

Using commutative operations is the closest approach to a general framework for automatic state convergence. The system concatenates logs, sorts them into some order, and then executes them. Commutativity implies the ability to rearrange operations into a preferred consistent global order. Unfortunately, using only commutative operations is harder than it appears; for example, addition is commutative, but addition with a bounds check is not (a zero balance, for example).

Recent work by Marc Shapiro and colleagues at INRIA[18,19] has greatly improved the use of commutative operations for state

convergence. The team has developed commutative replicated data types (CRDTs), a class of data structures that provably converge after a partition, and describe how to use these structures to

- ensure that all operations during a partition are commutative, *or*
- represent values on a lattice and ensure that all operations during a partition are monotonically increasing with respect to that lattice.

  The latter approach converges state by moving to the maximum of each side's values. It is a formalization and improvement of what Amazon does with its shopping cart:[20] after a partition, the converged value is the union of the two carts, with union being a monotonic set operation. The consequence of this choice is that deleted items may reappear.

  However, CRDTs can also implement partition-tolerant sets that both add and delete items. The essence of this approach is to maintain two sets: one each for the added and deleted items, with the difference being the set's membership. Each simplified set converges, and thus so does the difference. At some point, the system can clean things up simply by removing the deleted items from both sets. However, such cleanup generally is possible only while the system is not partitioned. In other words, the designer must prohibit or postpone some operations during a partition, but these are cleanup operations that do not limit perceived availability. Thus, by implementing state through CRDTs, a designer can choose A and still ensure that state converges automatically after a partition.

## Compensating for mistakes

In addition to computing the postpartition state, there is the somewhat harder problem of fixing mistakes made during partitioning. The tracking and limitation of partition-mode operations

ensures the knowledge of which invariants could have been violated, which in turn enables the designer to create a restoration strategy for each such invariant. Typically, the system discovers the violation during recovery and must implement any fix at that time.

There are various ways to fix the invariants, including trivial ways such as "last writer wins" (which ignores some updates), smarter approaches that merge operations, and human escalation. An example of the latter is airplane overbooking: boarding the plane is in some sense partition recovery with the invariant that there must be at least as many seats as passengers. If there are too many passengers, some will lose their seats, and ideally customer service will compensate those passengers in some way.

The airplane example also exhibits an externalized mistake: if the airline had not said that the passenger had a seat, fixing the problem would be much easier. This is another reason to delay risky operations: at the time of recovery, the truth is known. The idea of compensation is really at the core of fixing such mistakes; designers must create compensating operations that both restore an invariant and more broadly correct an externalized mistake.

Technically, CRDTs allow only locally verifiable invariants-a limitation that makes compensation unnecessary but that somewhat decreases the approach’s power. However, a solution that uses CRDTs for state convergence could allow the temporary violation of a global invariant, converge the state after the partition, and then execute any needed compensations.

Recovering from externalized mistakes typically requires some history about externalized outputs. Consider the drunk "dialing" scenario, in which a person does not remember making various telephone calls while intoxicated the previous night. That person’s state in the light of day might be sound, but the log still shows a list of calls, some of which might have been mistakes. The calls are the

external effects of the person's state (intoxication). Because the person failed to remember the calls, it could be hard to compensate for any trouble they have caused.

In a machine context, a computer could execute orders twice during a partition. If the system can distinguish two intentional orders from two duplicate orders, it can cancel one of the duplicates. If externalized, one compensation strategy would be to autogenerate an e-mail to the customer explaining that the system accidentally executed the order twice but that the mistake has been fixed and to attach a coupon for a discount on the next order. With-out the proper history, however, the burden of catching the mistake is on the customer.

Some researchers have formally explored compensating transactions as a way to deal with long-lived transactions.[21,22] Long-running transactions face a variation of the partition decision: is it better to hold locks for a long time to ensure consistency, or release them early and expose uncommitted data to other transactions but allow higher concurrency? A typical example is trying to update all employee records as a single transaction. Serializing this transaction in the normal way locks all records and prevents concurrency. Compensating transactions take a different approach by breaking the large transaction into a saga, which consists of multiple subtransactions, each of which commits along the way. Thus, to abort the larger transaction, the system must undo each already committed subtransaction by issuing a new transaction that corrects for its effects-the compensating transaction.

In general, the goal is to avoid aborting other transactions that used the incorrectly committed data (no cascading aborts). The correctness of this approach depends not on serializability or isolation, but rather on the net effect of the transaction sequence on state and outputs. That is, after compensations, does the database

essentially end up in a place equivalent to where it would have been had the subtransactions never executed? The equivalence must include externalized actions; for example, refunding a duplicate purchase is hardly the same as not charging that customer in the first place, but it is arguably equivalent. The same idea holds in partition recovery. A service or product provider cannot always undo mistakes directly, but it aims to admit them and take new, compensating actions. How best to apply these ideas to partition recovery is an open problem. The "Compensation Issues in an Automated Teller Machine" sidebar describes some of the concerns in just one application area.

System designers should not blindly sacrifice consistency or availability when partitions exist. Using the proposed approach, they can optimize both properties through careful management of invariants during partitions. As newer techniques, such as version vectors and CRDTs, move into frameworks that simplify their use, this kind of optimization should become more wide-spread. However, unlike ACID transactions, this approach requires more thoughtful deployment relative to past strategies, and the best solutions will depend heavily on details about the service's invariants and operations.

## Compensation issues in an automated teller machine

In the design of an automated teller machine (ATM), strong consistency would appear to be the logical choice, but in practice, A trumps C. The reason is straightforward enough: higher availability means higher revenue. Regardless, ATM design serves as a good context for reviewing some of the challenges involved in compensating for invariant violations during a partition.

The essential ATM operations are deposit, withdraw, and check balance. The key invariant is that the balance should be zero or

higher. Because only withdraw can violate the invariant, it will need special treatment, but the other two operations can always execute.

The ATM system designer could choose to prohibit withdrawals during a partition, since it is impossible to know the true balance at that time, but that would compromise availability. Instead, using stand-in mode (partition mode), modern ATMs limit the net withdrawal to at most k, where k might be $200. Below this limit, withdrawals work completely; when the balance reaches the limit, the system denies withdrawals. Thus, the ATM chooses a sophisticated limit on availability that permits withdrawals but bounds the risk.

When the partition ends, there must be some way to both restore consistency and compensate for mistakes made while the system was partitioned. Restoring state is easy because the operations are commutative, but compensation can take several forms. A final balance below zero violates the invariant. In the normal case, the ATM dispensed the money, which caused the mistake to become external. The bank compensates by charging a fee and expecting repayment. Given that the risk is bounded, the problem is not severe. However, suppose that the balance was below zero at some point during the partition (unknown to the ATM), but that a later deposit brought it back up. In this case, the bank might still charge an overdraft fee retroactively, or it might ignore the violation, since the customer has already made the necessary payment.

In general, because of communication delays, the banking system depends not on consistency for correctness, but rather on auditing and compensation. Another example of this is "check kiting," in which a customer withdraws money from multiple branches before they can communicate and then flees. The overdraft will be caught later, perhaps leading to compensation in the form of legal action.

## Acknowledgments

I thank Mike Dahlin, Hank Korth, Marc Shapiro, Justin Sheehy, Amin Vahdat, Ben Zhao, and the IEEE Computer Society volunteers for their helpful feedback on this work.

## About the Author

**Eric Brewer** is a professor of computer science at the University of California, Berkeley, and vice president of infrastructure at Google. His research interests include cloud computing, scalable servers, sensor networks, and technology for developing regions. He also helped create USA.gov, the official portal of the federal government. Brewer received a PhD in electrical engineering and computer science from MIT. He is a member of the National Academy of Engineering. Contact him at brewer@cs.berkeley.edu


*Computer, the flagship publication of the IEEE Computer Society, publishes highly acclaimed peer-reviewed articles written for and by professionals representing the full spectrum of computing technology from hardware to software and from current research to new applications. Providing more technical substance than trade magazines and more practical ideas than research journals. Computer delivers useful information that is applicable to everyday work environments.*

## References

1. E. Brewer, "Lessons from Giant-Scale Services," *IEEE Internet Computing*, July/Aug. 2001, pp. 46-55.
2. A. Fox et al., "Cluster-Based Scalable Network Services," Proc. 16th ACM Symp. *Operating Systems Principles* (SOSP 97), ACM, 1997, pp. 78-91.
3. A. Fox and E.A. Brewer, "Harvest, Yield and Scalable Tolerant

Systems," *Proc. 7th Workshop Hot Topics in Operating Systems* (HotOS 99), IEEE CS, 1999, pp. 174-178.
4. E. Brewer, "Towards Robust Distributed Systems," *Proc. 19th Ann. ACM Symp.Principles of Distributed Computing* (PODC 00), ACM, 2000, pp. 7-10; on-line resource.
5. B. Cooper et al., "PNUTS: Yahoo!'s Hosted Data Serving Platform," *Proc. VLDB Endowment* (VLDB 08), ACM, 2008, pp. 1277-1288.
6. J. Sobel, "Scaling Out," *Facebook Engineering Notes*, 20 Aug. 2008; on-line resource.
7. J. Kistler and M. Satyanarayanan, "Disconnected Operation in the Coda File System" *ACM Trans. Computer Systems*, Feb. 1992, pp. 3-25.
8. K. Birman, Q. Huang, and D. Freedman, "Overcoming the 'D' in CAP: Using Isis2 to Build Locally Responsive Cloud Services," *Computer*, Feb. 2011, pp. 50-58.
9. M. Burrows, "The Chubby Lock Service for Loosely-Coupled Distributed Systems," *Proc. Symp. Operating Systems Design and Implementation* (OSDI 06), Usenix, 2006, pp. 335-350.
10. J. Baker et al., "Megastore: Providing Scalable, Highly Available Storage for Interactive Services," *Proc. 5th Biennial Conf. Innovative Data Systems Research* (CIDR 11), ACM, 2011, pp. 223-234.
11. D. Abadi, "Problems with CAP, and Yahoo's Little Known NoSQL System," *DBMS Musings*, blog, 23 Apr. 2010; on-line resource.
12. C. Hale, "You Can't Sacrifice Partition Tolerance," 7 Oct. 2010; on-line resource.
13. W. K. Edwards et al., "Designing and Implementing Asynchronous Collaborative Applications with Bayou," *Proc. 10th Ann. ACM Symp. User Interface Software and Technology* (UIST 97), ACM, 1999, pp. 119-128.
14. P. Mahajan, L. Alvisi, and M. Dahlin, *Consistency, Availability,*

*and Convergence*, tech. report UTCS TR-11-22, Univ. of Texas at Austin, 2011.
15. D.B. Terry et al., "Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System," *Proc. 15th ACM Symp. Operating Systems Principles* (SOSP 95), ACM, 1995, pp. 172-182.
16. B. Du and E.A. Brewer, "DTWiki: A Disconnection and Intermittency Tolerant Wiki," Proc. 17th Int'l Conf. World Wide Web (WWW 08), ACM, 2008, pp. 945-952.
17. "What's Different about the New Google Docs: Conflict Resolution" blog.
18. M. Shapiro et al., "Conflict-Free Replicated Data Types," *Proc. 13th Int'l Conf. Stabilization, Safety, and Security of Distributed Systems* (SSS 11), ACM, 2011, pp. 386-400.
19. M. Shapiro et al., "Convergent and Commutative Replicated Data Types," *Bulletin of the EATCS*, no. 104, June 2011, pp. 67-88.
20. G. DeCandia et al., "Dynamo: Amazon's Highly Available Key-Value Store," *Proc. 21st ACM SIGOPS Symp. Operating Systems Principles* (SOSP 07), ACM, 2007, pp. 205-220.
21. H. Garcia-Molina and K. Salem, "SAGAS," *Proc. ACM SIGMOD Int'l Conf. Management of Data* (SIGMOD 87), ACM, 1987, pp. 249-259.
22. H. Korth, E. Levy, and A. Silberschatz, "A Formal Approach to Recovery by Compensating Transactions," *Proc. VLDB Endowment* (VLDB 90), ACM, 1990, pp. 95-106