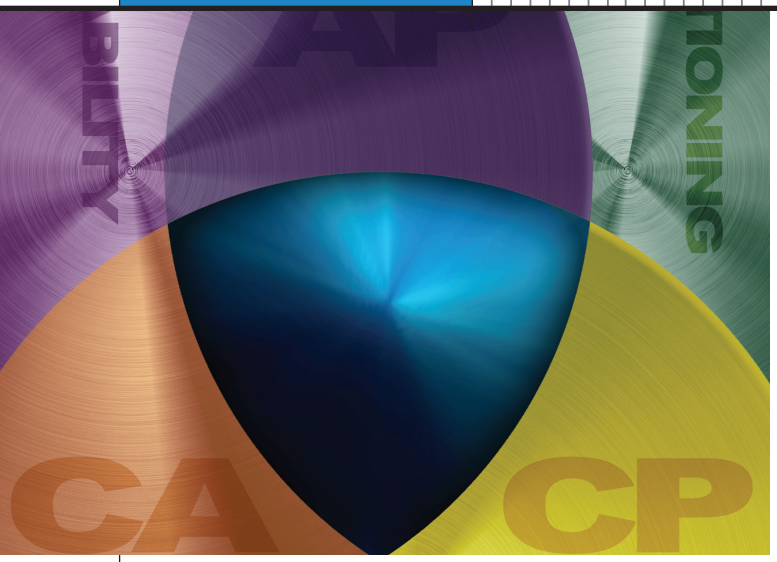

Perspectives on the CAP Theorem

Seth Gilbert, *National University of Singapore*
Nancy A. Lynch, *Massachusetts Institute of Technology*

The CAP theorem is one example of a more general tradeoff between safety and liveness in unreliable systems. Viewing CAP in this context provides insight into the inherent tradeoffs and the manner in which they can be circumvented in practice.

Almost 12 years ago, Eric Brewer introduced the idea that there is a fundamental tradeoff between *consistency*, *availability*, and network *partition tolerance*. This tradeoff, known as the CAP theorem, has been widely discussed ever since.

Some of the interest in CAP derives from the fact that it illustrates a general tradeoff in distributed computing: the impossibility of guaranteeing both safety and liveness in an unreliable distributed system.[1-5] Informally, an algorithm is *safe* if nothing bad ever happens. Consistency as defined in CAP is a classic safety property: every response sent to a client is correct. By contrast, an algorithm is *live* if something good eventually happens. Availability is a classic liveness property: every request eventually receives a response. Finally, a system can be *unreliable* in many ways, experiencing crash failures, message loss, malicious attacks, Byzantine failures, and so on. CAP is simply one example of this tradeoff between consistency and availability in unreliable systems.

Because of this inherent tradeoff, it is necessary to sacrifice one of these properties. Accordingly, some systems guarantee strong consistency and provide best-effort availability; others guarantee availability and provide best-effort consistency. Perhaps surprisingly, some systems sacrifice both consistency and availability, yielding better results than could be achieved otherwise. Viewing CAP in the broader context of safety/liveness tradeoffs provides insight into the feasible design space for distributed systems, shedding some light on the manner in which algorithm designers and software engineers have circumvented the theorem.

CAP THEOREM

Brewer first presented CAP in the context of a Web service[6] implemented by a set of servers distributed over a set of geographically diverse datacenters. Clients issue requests to the service, which sends back responses. This notion of a Web service is intentionally abstract and can embrace a wide variety of applications including search engines, e-commerce, online music services, and cloud-based data storage.

CAP terminology

CAP implies that a Web service must trade off consistency, availability, and partition tolerance. To prove the theorem, it is necessary to carefully understand each of these three terms.

Consistency. The first part of CAP refers to consistency, which is, informally, the property that each server returns the right response to each request, that is, a response that is appropriate to the desired service specification. The exact meaning of consistency depends on the type of service.

Trivial services do not require any coordination among the servers—for example, a service that returns the value

 Published by the IEEE Computer Society 0018-9162/12/$31.00 © 2012 IEEE

of the constant π to 100 decimal places. Trivial services do not fall within the scope of the theorem.

Weakly consistent services involve some distributed coordination, but each server can make progress on its own. As a result, they too do not fall within the scope of CAP and hence can avoid sacrificing availability while still providing some weak consistency guarantees. A distributed Web cache is an example of such a service.

Atomic services are defined in terms of atomic operations, which are described by a sequential specification. A sequential specification describes a service in terms of its execution on a single, centralized server: the server maintains some state, and it processes each request in order, updating the state and generating a response. A distributed service is atomic if, for every operation, there is a single instant between the request and response at which the operation appears to occur. From the clients' perspective, it is as if a single, centralized server executes all the operations according to the sequential specification.

Complicated services either cannot be specified sequentially or require more intricate coordination, transactional semantics, and so on.

Our formulation of CAP focuses on Web services that implement a read/write atomic shared memory: the service provides its clients with a single (emulated) register, and each client can read or write from that register. (Commonly used consistency conditions such as sequential or causal consistency typically yield similar CAP-like results.)

Availability. The second part of CAP refers to availability, which is, informally, the property that each request eventually receives a response. A fast response is clearly preferable to a slow response, but in the context of the theorem, even requiring an eventual response is sufficient to create problems. (In most real systems, of course, a response that is too late is just as bad as no response at all.)

Partition tolerance. The third part of CAP refers to partition tolerance. Unlike the other two requirements, partition tolerance is really a statement about the underlying system rather than the service itself: communication among the servers is unreliable, and the servers can be partitioned into multiple groups that cannot communicate with one another. We model a partition-prone system as one that is subject to faulty communication: messages can be delayed and sometimes lost forever. (Again, in practical terms, a long-delayed message might as well be lost.)

Proving CAP

In sum, CAP can be stated as follows: in a network subject to communication failures, it is impossible for any Web service to implement an atomic read/write shared memory that guarantees a response to every request.

There is a relatively simple proof of the theorem.[6] Assume the service consists of servers p_1, p_2, ..., p_n, along with an arbitrary set of clients. Consider an execution in which the servers are partitioned into two disjoint sets: $\{p_1\}$ and $\{p_2, ..., p_n\}$. Some client sends a read request to server p_2. Because p_1 is in a different component of the partition from p_2, the system loses every message from p_1 to p_2. Now consider the following two cases:

1. A previous write of value v_1 has been requested of p_1, and p_1 has sent an *ok* response.
2. A previous write of value v_2 has been requested of p_1, and p_1 has sent an *ok* response.

No matter how long p_2 waits, it cannot distinguish these two cases, hence it cannot determine whether to return response v_1 or v_2. Its choice is to either eventually return a (possibly wrong) response or to never return a response.

In fact, if communication is asynchronous—if processes have no a priori bound on how long it takes to deliver a message—then the same situation can occur even in executions in which there are no permanent partitions and no messages are lost. In the above scenario, server p_2 eventually must return a response, even if the system is partitioned; if the message delay from p_1 to p_2 is sufficiently large that p_2 believes the system to be partitioned, then it may return an incorrect response despite the lack of partitions. Thus it is even impossible to provide good responses when there are no partitions and bad responses only when there are partitions.

CAP states that any protocol implementing an atomic read/write register cannot guarantee both safety and liveness in a system prone to partitions.

THEORETICAL CONTEXT

The tradeoff between consistency and availability in a partition-prone system is an example of the general tradeoff between safety and liveness in an unreliable system. This notion that it is impossible for a system to achieve both properties has played a key role in distributed computing.

Consistency, availability, and partition tolerance

A safety property requires that at every point in every execution, the property holds. Consistency requirements are typically safety properties. For example, to say that an algorithm guarantees atomic consistency is to claim that in every execution, every response is correct with respect to the "prior" operations.

A liveness property, by contrast, says nothing about the state at any specific instant; it requires only that if an execution continues for long enough, then eventually

something desirable happens. In this context, CAP states that any protocol implementing an atomic read/write register cannot guarantee both safety and liveness in a system prone to partitions.

Agreement is impossible

Understanding the relationship between safety and liveness properties has been a long-standing challenge in distributed computing. The issue achieved widespread prominence in 1985 when Michael Fischer, Nancy Lynch, and Michael Paterson showed that fault-tolerant agreement is impossible in an asynchronous system.[3]

Their research focused on the problem of *consensus*, in which each process p_i begins with an initial value v_i, and the processes all must agree on one of those values. There are three requirements for consensus:

- *agreement*—every process must output the same value;
- *validity*—every value output must have been provided as the input for some process; and
- *termination*—every process must eventually output a value.

In the case of consensus, safety and liveness are impossible if the system is even potentially slightly faulty.

Agreement and validity are safety properties, while termination is a liveness property. The impossibility of consensus is thus an important example of the inherent tradeoff between safety and liveness.

The safety requirements of consensus are more difficult to meet than those we have considered with respect to CAP: achieving agreement is (provably) harder than simply implementing an atomic read/write register. Thus, CAP also implies that it is impossible to achieve consensus in a system subject to partitions.

Fischer, Lynch, and Paterson did not consider a system with partitions; they were concerned with the more benign issue of crash failures. Specifically, they assumed that one unknown process can cease operation while most of the processes in the system continue to communicate reliably. Their surprising conclusion was that consensus is impossible in such a system. In fact, for every purported consensus protocol that guarantees agreement and validity, there is some execution in which there are no failures and yet the algorithm never terminates. In the case of consensus, safety and liveness are impossible if the system is even potentially slightly faulty.

This result has had significant implications. For example, the problem of consensus is at the heart of the replicated state machine paradigm, one of the most common approaches for building reliable distributed services.[7-10] This paradigm achieves availability by replicating the service across a set of servers. The servers then agree on every operation performed by the service. The impossibility of fault-tolerant consensus implies that services built according to the replicated state machine paradigm cannot achieve both availability and correctness in an asynchronous network.

Safety/liveness tradeoff for consensus

Distributed computing researchers soon began exploring safety/liveness tradeoffs in more depth. Their efforts provide context for analogous questions raised by CAP.

Synchrony. Given that safety and liveness are impossible in sufficiently unreliable systems, the first natural question to arise was under what conditions it is possible to achieve both. Much research has concentrated on network synchrony: What level of synchrony is necessary to avoid the inherent tradeoff? What level of network reliability is needed to achieve both consistency and availability, subject to a given level of faulty behavior?

A network is said to be synchronous if it satisfies the following properties:

- every process has a clock, and all the clocks are synchronized;
- every message is delivered within a fixed and known amount of time; and
- every process takes steps at a fixed and known rate.

We can think of such systems as progressing in rounds; within each round, each process sends some messages, receives all the messages that were sent to it in that round, and performs some local computation.

If a system is wholly synchronous, consensus can be achieved—that is, the tradeoff between safety and liveness can be wholly avoided. Notably, consensus requires $f + 1$ rounds, if up to f servers can crash.[11,12] As consensus is impossible in an asynchronous system if there is even one crash failure, how much synchrony is needed to achieve consensus?

Cynthia Dwork and coauthors attempted to answer this question by exploring various partial synchrony models.[13] Most notably, they introduced the idea of *eventual synchrony*: a system can experience periods of both synchrony and asynchrony, but as long as it eventually stabilizes and maintains synchrony for a sufficiently long time, it is possible to achieve consensus. Subsequent research has demonstrated that $f + 2$ rounds of synchrony are adequate.[14,15]

There is also a connection between a system's level of synchrony and its crash tolerance. A synchronous system can achieve consensus for any number of failures, while in an asynchronous system it is impossible to achieve consensus for even one failure. In an eventually synchronous system, however, it is possible to achieve consensus only if there are $< n/2$ crash failures, where n is the number of servers. If there are $\geq n/2$ crash failures, consensus is again impossible (due to partitioning, much like in CAP). Most practical consensus implementations today are explicitly or implicitly designed for eventually synchronous systems.[7,9,16]

Another line of research has pursued a different approach to the question of how much synchrony is needed to achieve consensus. Tushar Chandra and colleagues introduced the idea of a *failure detector* that provides sufficient information for processes to achieve consensus in an asynchronous crash-prone system.[2,17] One example of a failure detector is a *hello protocol* that monitors which neighbors in a network are active. Another example is a *leader election service*, which identifies a reliable server to act as the leader.

Marcos Aguilera and colleagues pursued a third line of research, exploring a specific minimal set of link reliability and synchrony assumptions for achieving consensus.[18,19]

Consistency. A second natural question arising from the inherent safety/liveness tradeoff relates to consistency: given that the network is unreliable, what is the maximum level of consistency that can be achieved?

In response to this question, Soma Chaudhuri introduced the concept of *set agreement*.[20] Much like consensus, each process proposes a value and eventually outputs a value that was initially proposed by some process—that is, the validity and termination conditions are identical. Unlike consensus, however, output values can disagree in a limited way. Specifically, for k-set agreement, there can be up to k different output values.

This weaker consistency guarantee leads to a sequence of problems: 1-set agreement, 2-set agreement, 3-set agreement, ..., n-set agreement. Note that 1-set agreement is identical to consensus, and n-set agreement is trivial—that is, each process simply outputs its own initial value. Consequently, 1-set agreement is impossible if there is even one crash failure, while n-set agreement can tolerate an arbitrary number of crash failures.

A seminal series of studies demonstrated that k-set agreement can be achieved if and only if there are at most $k - 1$ crash failures.[1,4,5] Thus, k-set agreement is the "most" agreement achievable in a system with $k - 1$ failures that ensures availability. Chaudhuri and colleagues related the degree of consistency to the running time: in a synchronous system with t failures, at least $\lfloor t / k \rfloor + 1$ rounds are necessary and sufficient to achieve k-set agreement.[21]

PRACTICAL IMPLICATIONS

To overcome CAP's negative implications, practitioners building and deploying distributed services over unreliable networks have traditionally chosen to sacrifice either availability or consistency. However, alternative approaches have emerged in practice that sacrifice both. Some such systems balance the required level of availability and consistency. Others segment a larger system into different components, each of which can choose a different tradeoff. The resulting design often yields a system that responds well to most user requests, even under bad network conditions, and also provides high levels of consistency when and where needed.

Practitioners building and deploying distributed services over unreliable networks have traditionally chosen to sacrifice either availability or consistency.

Best-effort availability

Perhaps the most common approach to dealing with unreliable networks is to design a service that guarantees consistency—that is, correct operation—regardless of network behavior. Software architects then optimize the service to provide best-effort availability, that is, to be as responsive as possible given current network conditions. This design makes sense when communication is typically reliable and timely—for example, when all the servers running a service are in the same datacenter—and partitions or other network anomalies occur only rarely.

A recent popular example of this approach is Chubby, a coarse-grained lock service and metadata service for distributed networks that supports the Google File System, BigTable, and other key Google infrastructure elements.[7,22] Google also uses Chubby as a naming service to replace the Domain Name System (DNS).

At the heart of Chubby is a distributed database with a primary-backup design. The system ensures strong consistency among the servers using a replicated state machine protocol, Paxos,[9] to maintain synchronized logs. Chubby operates effectively as long as no more than half the servers fail and the network is reliable.

Most of the time, there are no partitions in the system. Each Chubby "cell" is deployed in a single datacenter, and communication within a cell is typically fast and reliable. The primary replica rarely fails, and when it does, changing to another backup results in only occasional delays. Overall, Chubby provides very high availability most of the time.

Best-effort consistency

For some applications, sacrificing availability is not an option: users require that the application be responsive in all situations. Moreover, when the application is deployed over a wide area rather than within a datacenter, availability can degrade rapidly. In such situations, designers sacrifice consistency to guarantee a response—preferably a fast one—at all times. Consequently, the response might not always be correct. The system provides only best-effort consistency.

The classic example of this approach is seen in Web caches, services that store content such as images and video on servers in globally distributed datacenters. Whenever a user requests a webpage, the system delivers content from a nearby Web cache, if it is available; otherwise, it retrieves the data from the Web server on which the content was originally hosted.

CAP has played an important role in defining the design space, and has implications for the changing world we inhabit today.

Such a system guarantees very high availability: the proximity of the cache servers to end users ensures that responses are rapid, and network connectivity issues rarely prevent a response. On the other hand, the consistency guarantees of Web caching are weaker. When the system updates a webpage, it might take some time for the new content to propagate to all the cache servers. The caching service does its best to provide up-to-date content, but there is no assurance that all users accessing a webpage at any given time receive the exact same content.

For Akamai and other content delivery service providers, this tradeoff makes sense. Users viewing Web content do not necessarily require strong consistency: there is usually little harm if users in different locations view slightly different versions of a webpage or if the content is a little out of date. On the other hand, Web users have little patience—a fast response is critical. In addition, the widespread geographical distribution of Web users implies the need to sacrifice consistency to achieve sufficient availability and performance.

Balancing consistency and availability

Some systems allow for an adjustable tradeoff between consistency and availability. For example, the owner of a website may specify how out-of-date the content may become: it might be acceptable for some content to be one hour out of date, but not one day out of date. A website owner who specifies this weaker level of consistency is provided in return with a higher level of availability. By setting a threshold for out-of-date data, system operators can precisely specify the desired CAP tradeoff.

Haifeng Yu and Amin Vahdat explored this approach with the TACT (Tunable Availability and Consistency Tradeoffs) toolkit, which enables replicated applications to specify the desired level of *continuous consistency*. A notable aspect of their system is the ability to update this value dynamically, as the application executes.[23,24] Consider, for example, an airline reservation system. When most of the seats on an airplane are available, the system can rely on somewhat out-of-date data without overbooking the plane. As the plane fills, however, the system requires increasingly accurate data to prevent overbooking. Using TACT, the system could request increasing levels of consistency as the number of available seats diminishes.

Such a tunable system guarantees neither strong consistency nor continual availability: data can be inconsistent, and a major network partition can still render the service unavailable. Nevertheless, this type of tradeoff can significantly increase the system's robustness to network disruption, before it must compromise availability.

Segmenting consistency and availability

Many systems do not have a single uniform requirement: some aspects require strong consistency and others high availability. For such systems, a natural approach to circumventing CAP is to segment the system into components that provide different types of guarantees. Doing so might again produce a service that overall guarantees neither consistency nor availability, yet ultimately each part of the service provides exactly what is needed.

The system can be partitioned along various dimensions. The precise guarantees that segmentation provides are not always clear and are specific to the given application and the particular partitioning scheme.

Data partitioning. Different types of data can require various levels of consistency and availability. For example, an online shopping cart might be highly available, responding rapidly to user requests, yet occasionally inconsistent, losing a recent update in anomalous circumstances. An e-commerce site's product information likewise could be somewhat inconsistent: users will tolerate slightly out-of-date inventory data. However, checkout, billing, and shipping records must be strongly consistent: users will be unhappy if a final order does not reflect their intended purchase. Different data might require different tradeoffs.

Operational partitioning. Different operations may require different levels of consistency and availability. Consider, for example, a system that guarantees high availability for read-only operations, while operations that modify the database need not respond during network partitions. For an e-commerce site, a purchase

operation must be consistent, while a query operation can return out-of-date data. To achieve a good user experience, Yahoo's PNUTS (Platform for Nimble Universal Table Storage) system provides such tradeoffs for different read and write operations.[25]

Functional partitioning. Many services can be divided into subservices with their own requirements. For example, an application might use a service such as Chubby for coarse-grained locks and distributed coordination (strong consistency), a service such as DNS to handle naming (relatively weak consistency but high availability), and a third subservice with a different consistency/availability tradeoff for content distribution.

User partitioning. Network partitions, and poor network performance in general, typically correlate with geographic distance: users far away from servers are more likely to experience poor performance. Thus, a service like Craigslist might elect to divide its servers among two different datacenters in the US—one on the east coast and one on the west coast. The service would, for example, rely on the west coast datacenter to provide high availability to users from California cities; moreover, as it stores and maintains the California data within a single datacenter, the system can more easily achieve consistency among the west coast servers. The same holds true for east coast users that rely on the east coast datacenter. (On the other hand, New York users inquiring about Craigslist ads in San Francisco might experience weaker performance under this design.) A social networking site might similarly try to partition its users to ensure high availability among groups of friends.

Hierarchical partitioning. Some applications are organized hierarchically, with partitioning occurring multiple times along different dimensions. At the top level, an application encompasses the whole world or the entire database; subsequent levels of the hierarchy partition the world or database into smaller parts. At each level of the hierarchy, the system might provide a different level of performance: better availability toward the leaves, less consistency toward the roots. For example, in descending a geographically organized hierarchy, CAP's limitations become less onerous as the relevant servers become better connected.

Since the CAP theorem was first formulated over a decade ago, the networked world has changed significantly, creating new tradeoffs to explore and new challenges to overcome. The theorem has played an important role in defining the design space, and has implications for the changing world we inhabit today. The same safety/liveness tradeoffs raised by CAP arise when considering issues of scalability and network security, and when developing mobile and wireless networks.

Increasingly, systems must be scalable to accommodate future growth as well as today's needs. Intuitively, a system is scalable if it can use new resources efficiently to handle more load. CAP hints at a tradeoff between scalability and consistency (and latency): maintaining consistency among more resources requires more communication, which is subject to safety/liveness tradeoffs. This might explain why even within a datacenter, where partitions rarely occur, it is difficult to efficiently scale strongly consistent protocols like Paxos.

Severe attacks on networks are on the rise. For example, denial-of-service attacks are becoming a near-continuous threat to everyday network operations. These sorts of attacks create safety and liveness issues. CAP, however, addresses network partitions, and a DoS attack cannot simply be modeled as a network partition. Similarly, malicious users are increasingly hacking servers and otherwise disrupting major Internet services. Tolerating such attacks requires a new understanding of consistency/availability tradeoffs.

CAP was initially intended to address wide-area Internet services. Today, however, mobile devices initiate a significant and growing percentage of Internet traffic. Many of the same CAP tradeoffs also apply to mobile networks and are even harder to overcome: wireless communication is notoriously unreliable, and message latencies can vary considerably.

In addition, many of the applications deployed in wireless networks have different properties than traditional Internet services like search engines and e-commerce websites. For example, they are heavily influenced by geography and proximity; they are organized around social interactions; and privacy considerations are more immediate. By reexamining CAP in the context of wireless networks, we might better understand the unique tradeoffs that occur in these types of scenarios. C

References

1. E. Borowsky and E. Gafni, "Generalized FLP Impossibility Result for t-Resilient Asynchronous Computations," *Proc. 25th Ann. ACM Symp. Theory of Computing* (STOC 93), ACM, 1993, pp. 91-100.
2. T.D. Chandra and S. Toueg, "Unreliable Failure Detectors for Reliable Distributed Systems," *JACM*, Mar. 1996, pp. 225-267.
3. M.J. Fischer, N.A. Lynch, and M.S. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," *JACM*, Apr. 1985, pp. 374-382.
4. M. Herlihy and N. Shavit, "The Topological Structure of Asynchronous Computability," *JACM*, Nov. 1999, pp. 858-923.
5. M. Saks and F. Zaharoglou, "Wait-Free *k*-Set Agreement Is Impossible: The Topology of Public Knowledge," *SIAM J. Computing*, Mar. 2000, pp. 1449-1483.

6. S. Gilbert and N. Lynch, "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services," *ACM SIGACT News*, June 2002, pp. 51-59.
7. T.D. Chandra, R. Griesemer, and J. Redstone, "Paxos Made Live: An Engineering Perspective," *Proc. 26th Ann. ACM Symp. Principles of Distributed Computing* (PODC 07), ACM, 2007, pp. 398-407.
8. L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Comm. ACM*, July 1978, pp. 558-565.
9. L. Lamport, "The Part-Time Parliament," *ACM Trans. Computer Systems*, May 1998, pp. 133-169.
10. B.W. Lampson, "How to Build a Highly Available System Using Consensus," *Proc. 10th Int'l Workshop Distributed Algorithms* (WDAG 96), Springer, 1996, pp. 1-17.
11. L. Lamport and M. Fischer, "Byzantine Generals and Transaction Commit Protocols," opus 62, unpublished paper, rev. 25 Apr. 1984; http://research.microsoft.com/en-us/um/people/lamport/pubs/trans.pdf.
12. N.A. Lynch, *Distributed Algorithms*, Morgan Kaufmann, 1996.
13. C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the Presence of Partial Synchrony," *JACM*, Apr. 1988, pp. 288-323.
14. P. Dutta and R. Guerraoui, "The Inherent Price of Indulgence," *Proc. 21st Ann. ACM Symp. Principles of Distributed Computing* (PODC 02), ACM, 2002, pp. 88-97.
15. D. Alistarh et al., "How to Solve Consensus in the Smallest Window of Synchrony," *Proc. 22nd Int'l Symp. Distributed Computing* (DISC 08), Springer, 2008, pp. 32-46.
16. M. Castro and B. Liskov, "Practical Byzantine Fault Tolerance and Proactive Recovery," *ACM Trans. Computer Systems*, Nov. 2002, pp. 398-461.
17. T.D. Chandra, V. Hadzilacos, and S. Toueg, "The Weakest Failure Detector for Solving Consensus," *JACM*, July 1996, pp. 685-722.
18. M.K. Aguilera et al., "Communication-Efficient Leader Election and Consensus with Limited Link Synchrony," *Proc. 23rd Ann. ACM Symp. Principles of Distributed Computing* (PODC 04), ACM, 2004, pp. 328-337.
19. M.K. Aguilera et al., "On Implementing Omega in Systems with Weak Reliability and Synchrony Assumptions," *Distributed Computing*, Oct. 2008, pp. 285-314.
20. S. Chaudhuri, "More Choices Allow More Faults: Set Consensus Problems in Totally Asynchronous Systems," *Information and Computation*, July 1993, pp. 132-158.
21. S. Chaudhuri et al., "A Tight Lower Bound for *k*-Set Agreement," *Proc. IEEE 34th Ann. Symp. Foundations of Computer Science* (SFCS 93), IEEE CS, 1993, pp. 206-215.
22. M. Burrows, "The Chubby Lock Service for Loosely-Coupled Distributed Systems," *Proc. 7th Symp. Operating Systems Design and Implementation* (OSDI 06), Usenix, 2006, pp. 335-350.
23. H. Yu and A. Vahdat, "Design and Evaluation of a Conit-Based Continuous Consistency Model for Replicated Services," *ACM Trans. Computer Systems*, Aug. 2002, pp. 239-282.
24. H. Yu and A. Vahdat, "The Costs and Limits of Availability for Replicated Services," *ACM Trans. Computer Systems*, Feb. 2006, pp. 70-113.
25. B.F. Cooper et al., "PNUTS: Yahoo!'s Hosted Data Serving Platform," *Proc. VLDB Endowment* (VLDB 08), ACM, 2008, pp. 1277-1288.

Seth Gilbert *is an assistant professor in the Department of Computer Science at the National University of Singapore. His research focuses on fault tolerance and scalability in large-scale, highly dynamic distributed systems. Gilbert received a PhD in computer science from Massachusetts Institute of Technology. Contact him at seth.gilbert@comp.nus.edu.sg.*

Nancy A. Lynch *is the NEC Professor of Software Science and Engineering in the Department of Electrical Engineering and Computer Science at Massachusetts Institute of Technology, where she also heads the Theory of Distributed Systems (TDS) research group at the Computer Science and Artificial Intelligence Lab (CSAIL). Her research focuses on distributed algorithms and impossibility results and on the formal modeling and verification of distributed systems. Lynch received a PhD in mathematics from MIT. She is an ACM Fellow and a member of the National Academy of Engineering. Contact her at lynch@theory.csail.mit.edu.*