DOI:10.1145/1435417.1435432

Building reliable distributed systems at a worldwide scale demands trade-offs between consistency and availability.

BY WERNER VOGELS

Eventually Consistent

AT THE FOUNDATION of Amazon's cloud computing are infrastructure services such as Amazon's S3 (Simple Storage Service), SimpleDB, and EC2 (Elastic Compute Cloud) that provide the resources for constructing Internet-scale computing platforms and a great variety of applications. The requirements placed on these infrastructure services are very strict; they need to score high marks in the areas of security, scalability, availability, performance, and cost-effectiveness, and they need to meet these requirements while serving millions of customers around the globe, continuously.

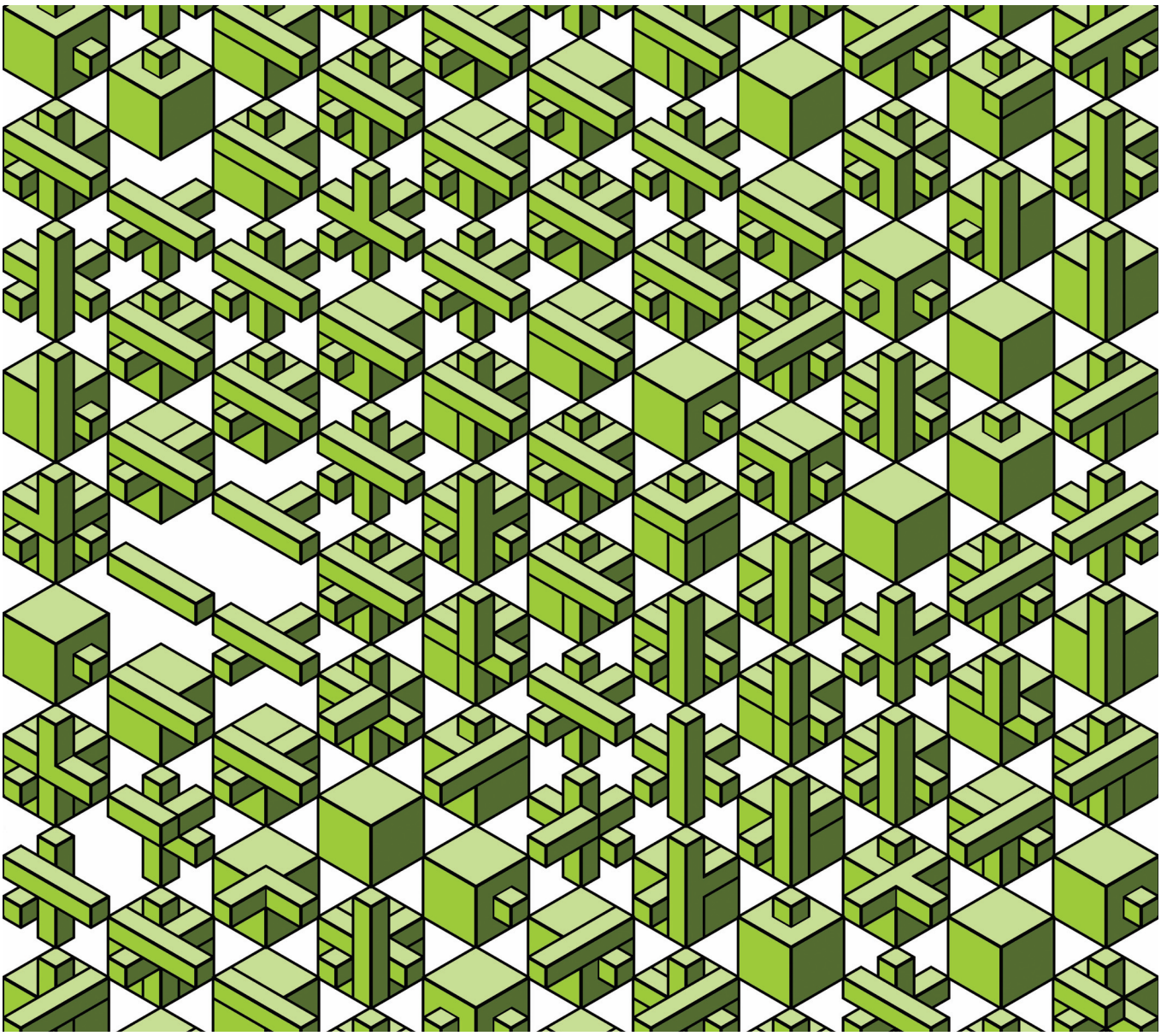
Under the covers these services are massive distributed systems that operate on a worldwide scale. This scale creates additional challenges, because when a system processes trillions and trillions of requests, events that normally have a low probability of occurrence are now guaranteed to happen and must be accounted for upfront in the design and architecture of the system. Given the worldwide scope of these systems, we use replication techniques ubiquitously to guarantee consistent performance and high availability. Although replication brings us closer to our goals, it cannot achieve them in a perfectly transparent manner; under a number of conditions the customers of these services will be confronted with the consequences of using replication techniques inside the services.

One of the ways in which this manifests itself is in the type of data consistency that is provided, particularly when many widespread distributed systems provide an *eventual consistency* model in the context of data replication. When designing these large-scale systems at Amazon, we use a set of guiding principles and abstractions related to large-scale data replication and focus on the trade-offs between high availability and data consistency. Here, I present some of the relevant background that has informed our approach to delivering reliable distributed systems that must operate on a global scale. (An earlier version of this article appeared as a posting on the "All Things Distributed" Weblog and was greatly improved with the help of its readers.)

Historical Perspective

In an ideal world there would be only one consistency model: when an update is made all observers would see that update. The first time this surfaced as difficult to achieve was in the database systems of the late 1970s. The best "period piece" on this topic is "Notes on Distributed Databases" by Bruce Lindsay et al.[5] It lays out the fundamental principles for database replication and discusses a number of techniques that deal with achieving consistency. Many of these techniques try to achieve *distribution transparency*—that is, to the user of the system it appears as if there is only one system instead of a number of collaborating systems. Many systems during this time took the approach that it was better to fail the complete system than to break this transparency.[2]

In the mid-1990s, with the rise of larger Internet systems, these practices were revisited. At that time people began to consider the idea that availability was perhaps the most impor-



tant property of these systems, but they were struggling with what it should be traded off against. Eric Brewer, systems professor at the University of California, Berkeley, and at that time head of Inktomi, brought the different tradeoffs together in a keynote address to the Principles of Distributed Computing (PODC) conference in 2000.[1] He presented the *CAP theorem*, which states that of three properties of shared-data systems—data consistency, system availability, and tolerance to network partition—only two can be achieved at any given time. A more formal confirmation can be found in a 2002 paper by Seth Gilbert and Nancy Lynch.[4]

A system that is not tolerant to network partitions can achieve data consistency and availability, and often does so by using transaction protocols. To make this work, client and storage systems must be part of the same environment; they fail as a whole under certain scenarios and as such clients cannot observe partitions. An important observation is that in larger distributed-scale systems, network partitions are a given; therefore, consistency and availability cannot be achieved at the same time. This means there are two choices on what to drop: relaxing consistency will allow the system to remain highly available under the partitionable conditions; making consistency a priority means that under certain conditions the system will not be available.

Both options require the client developer to be aware of what the system is offering. If the system emphasizes consistency, the developer has to deal with the fact that the system may not be available to take, for example, a write. If this write fails because of system unavailability, then the developer will have to deal with what to do with the data to be written. If the system emphasizes availability, it may always accept the write, but under certain conditions a read will not reflect the result of a recently completed write. The developer then has to decide whether the client requires access to the absolute latest update all the time. There is a range of applications that can handle slightly stale data, and they are served well under this model.

In principle the consistency property of transaction systems as defined in the ACID properties (atomicity,

ILLUSTRATIONS BY DAVE BOLLINGER

consistency, isolation, durability) is a different kind of consistency guarantee. In ACID, consistency relates to the guarantee that when a transaction is finished the database is in a consistent state; for example, when transferring money from one account to another the total amount held in both accounts should not change. In ACID-based systems, this kind of consistency is often the responsibility of the developer writing the transaction but can be assisted by the database managing integrity constraints.

Consistency—Client and Server

There are two ways of looking at consistency. One is from the developer/client point of view: how they observe data updates. The other is from the server side: how updates flow through the system and what guarantees systems can give with respect to updates.

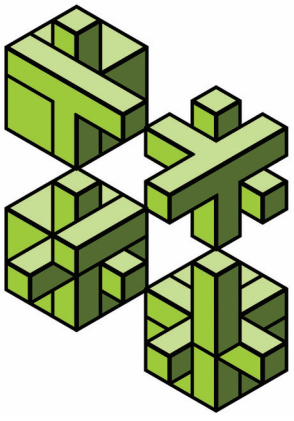
The components for the client side include:

▸ *A storage system.* For the moment we'll treat it as a black box, but one should assume that under the covers it is something of large scale and highly distributed, and that it is built to guarantee durability and availability.

▸ *Process A.* This is a process that writes to and reads from the storage system.

▸ *Process B and C.* These two processes are independent of process A and write to and read from the storage system. It is irrelevant whether these are really processes or threads within the same process; what is important is that they are independent and need to communicate to share information.

Client-side consistency has to do with how and when observers (in this case the processes A, B, or C) see updates made to a data object in the storage systems. In the following examples illustrating the different types of consistency, process A has made an update to a data object:



▸ *Strong consistency.* After the update completes, any subsequent access (by A, B, or C) will return the updated value.

▸ *Weak consistency.* The system does not guarantee that subsequent accesses will return the updated value. A number of conditions need to be met before the value will be returned. The period between the update and the moment when it is guaranteed that any observer will always see the updated value is dubbed the *inconsistency window*.

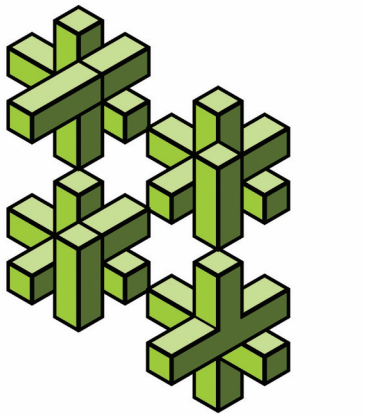
▸ *Eventual consistency.* This is a specific form of weak consistency; the storage system guarantees that if no new updates are made to the object, eventually all accesses will return the last updated value. If no failures occur, the maximum size of the inconsistency window can be determined based on factors such as communication delays, the load on the system, and the number of replicas involved in the replication scheme. The most popular system that implements eventual consistency is the domain name system (DNS). Updates to a name are distributed according to a configured pattern and in combination with time-controlled caches; eventually, all clients will see the update.

The eventual consistency model has a number of variations that are important to consider:

▸ *Causal consistency.* If process A has communicated to process B that it has updated a data item, a subsequent access by process B will return the updated value, and a write is guaranteed to supersede the earlier write. Access by process C that has no causal relationship to process A is subject to the normal eventual consistency rules.

▸ *Read-your-writes consistency.* This is an important model where process A, after having updated a data item, always accesses the updated value and never sees an older value. This is a special case of the causal consistency model.

▸ *Session consistency.* This is a practical version of the previous model, where a process accesses the storage system in the context of a session. As long as the session exists, the system guarantees read-your-writes consistency. If the session terminates because of a certain failure scenario, a new session must be created and the guarantees do not overlap the sessions.



▸ *Monotonic read consistency.* If a process has seen a particular value for the object, any subsequent accesses will never return any previous values.

▸ *Monotonic write consistency.* In this case, the system guarantees to serialize the writes by the same process. Systems that do not guarantee this level of consistency are notoriously difficult to program.

A number of these properties can be combined. For example, one can get monotonic reads combined with session-level consistency. From a practical point of view these two properties (monotonic reads and read-your-writes) are most desirable in an eventual consistency system, but not always required. These two properties make it simpler for developers to build applications, while allowing the storage system to relax consistency and provide high availability.

As you can see from these variations, quite a few different scenarios are possible. It depends on the particular applications whether or not one can deal with the consequences.

Eventual consistency is not some esoteric property of extreme distributed systems. Many modern RDBMSs (relational database management systems) that provide primary-backup reliability implement their replication techniques in both synchronous and asynchronous modes. In synchronous mode the replica update is part of the transaction. In asynchronous mode the updates arrive at the backup in a delayed manner, often through log shipping. In the latter mode if the primary fails before the logs are shipped,

reading from the promoted backup will produce old, inconsistent values. Also to support better scalable read performance, RDBMSs have started to provide the ability to read from the backup, which is a classical case of providing eventual consistency guarantees in which the inconsistency windows depend on the periodicity of the log shipping.

On the server side we need to take a deeper look at how updates flow through the system to understand what drives the different modes that the developer who uses the system can experience. Let's establish a few definitions before getting started:

N = The number of nodes that store replicas of the data.

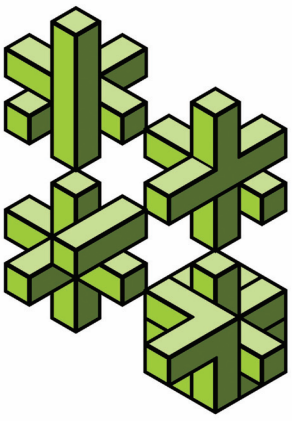
W = The number of replicas that need to acknowledge the receipt of the update before the update completes.

R = The number of replicas that are contacted when a data object is accessed through a read operation.

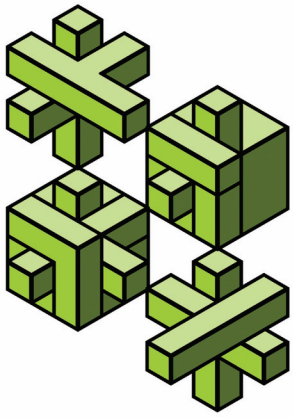
If W+R > N, then the write set and the read set always overlap and one can guarantee strong consistency. In the primary-backup RDBMS scenario, which implements synchronous replication, N=2, W=2, and R=1. No matter from which replica the client reads, it will always get a consistent answer. In the asynchronous replication case with reading from the backup enabled, N=2, W=1, and R=1. In this case R+W=N, and consistency cannot be guaranteed.

The problems with these configurations, which are basic quorum protocols, is that when because of failures the system cannot write to W nodes, the write operation has to fail, marking the unavailability of the system. With N=3 and W=3 and only two nodes available, the system will have to fail the write.

In distributed storage systems that



When a system processes trillions and trillions of requests, events that normally have a low probability of occurrence are now guaranteed to happen and must be accounted for upfront in the design and architecture of the system.



provide high performance and high availability the number of replicas is in general higher than two. Systems that focus solely on fault tolerance often use N=3 (with W=2 and R=2 configurations). Systems that must serve very high read loads often replicate their data beyond what is required for fault tolerance; N can be tens or even hundreds of nodes, with R configured to 1 such that a single read will return a result. Systems that are concerned with consistency are set to W=N for updates, which may decrease the probability of the write succeeding. A common configuration for these systems that are concerned about fault tolerance but not consistency is to run with W=1 to get minimal durability of the update and then rely on a lazy (epidemic) technique to update the other replicas.

How to configure N, W, and R depends on what the common case is and which performance path needs to be optimized. In R=1 and N=W we optimize for the read case, and in W=1 and R=N we optimize for a very fast write. Of course in the latter case, durability is not guaranteed in the presence of failures, and if W < (N+1)/2, there is the possibility of conflicting writes when the write sets do not overlap.

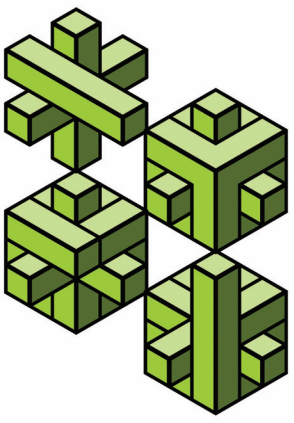
Weak/eventual consistency arises when W+R <= N, meaning that there is a possibility that the read and write set will not overlap. If this is a deliberate configuration and not based on a failure case, then it hardly makes sense to set R to anything but 1. This happens in two very common cases: the first is the massive replication for read scaling mentioned earlier; the second is where data access is more complicated. In a simple key-value model it is easy to compare versions to determine the latest value written to the system, but in

systems that return sets of objects it is more difficult to determine what the correct latest set should be. In most of these systems where the write set is smaller than the replica set, a mechanism is in place that applies the updates in a lazy manner to the remaining nodes in the replica's set. The period until all replicas have been updated is the inconsistency window discussed before. If W+R <= N, then the system is vulnerable to reading from nodes that have not yet received the updates.

Whether or not read-your-write, session, and monotonic consistency can be achieved depends in general on the "stickiness" of clients to the server that executes the distributed protocol for them. If this is the same server every time, then it is relatively easy to guarantee read-your-writes and monotonic reads. This makes it slightly more difficult to manage load balancing and fault tolerance, but it is a simple solution. Using sessions, which are sticky, makes this explicit and provides an exposure level that clients can reason about.

Sometimes the client implements read-your-writes and monotonic reads. By adding versions on writes, the client discards reads of values with versions that precede the last-seen version.

Partitions happen when some nodes in the system cannot reach other nodes, but both sets are reachable by groups of clients. If you use a classical majority quorum approach, then the partition that has W nodes of the replica set can continue to take updates while the other partition becomes unavailable. The same is true for the read set. Given that these two sets overlap, by definition the minority set becomes unavailable. Partitions don't happen frequently, but they do occur between data centers, as well as inside data centers.



In some applications the unavailability of any of the partitions is unacceptable, and it is important that the clients that can reach that partition make progress. In that case both sides assign a new set of storage nodes to receive the data, and a merge operation is executed when the partition heals. For example, within Amazon the shopping cart uses such a write-always system; in the case of partition, a customer can continue to put items in the cart even if the original cart lives on the other partitions. The cart application assists the storage system with merging the carts once the partition has healed.

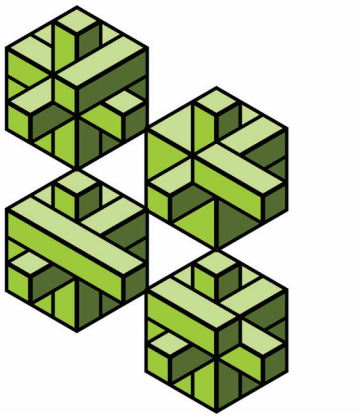
Amazon's Dynamo

A system that has brought all of these properties under explicit control of the application architecture is Amazon's Dynamo, a key-value storage system that is used internally in many services that make up the Amazon e-commerce platform, as well as Amazon's Web Services. One of the design goals of Dynamo is to allow the application service owner who creates an instance of the Dynamo storage system—which commonly spans multiple data centers—to make the trade-offs between consistency, durability, availability, and performance at a certain cost point.[3]

Summary

Data inconsistency in large-scale reliable distributed systems must be tolerated for two reasons: improving read and write performance under highly concurrent conditions; and handling partition cases where a majority model would render part of the system unavailable even though the nodes are up and running.

Whether or not inconsistencies are acceptable depends on the client application. In all cases the developer must be aware that consistency guarantees are provided by the storage systems and must be taken into account when developing applications. There are a number of practical improvements to the eventual consistency model, such as session-level consistency and monotonic reads, which provide better tools for the developer to work with. Many times the application is capable of handling the eventual consistency guarantees of the storage system without any problem. A specific popular case is a Web site in which we can have the notion of user-perceived consistency. In this scenario the inconsistency window must be smaller than the time expected for the customer to return for the next page load. This allows for updates to propagate through the system before the next read is expected.



The goal of this article is to raise awareness about the complexity of engineering systems that need to operate at a global scale and that require careful tuning to ensure that they can deliver the durability, availability, and performance that their applications require. One of the tools the system designer has is the length of the consistency window, during which the clients of the systems are possibly exposed to the realities of large-scale systems engineering. C

References

1. Brewer, E.A. Towards robust distributed systems (abstract). In *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing* (July 16–19, 2000, Portland, OR), 7.
2. Conversation with Bruce Lindsay. *ACM Queue 2*, 8 (2004), 22–33.
3. DeCandia, G., et. al. Dynamo: Amazon's highly available key-value store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles* (Stevenson, WA, Oct. 2007).
4. Gilbert, S. and Lynch, N. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant Web services. *ACM SIGACT News 33*, 2 (2002).
5. Lindsay, B.G. et al. Notes on distributed databases. *Distributed Data Bases*. I.W. Draffan and F. Poole, Eds. Cambridge University Press, Cambridge, MA, 1980, 247–284. Also available as IBM Research Report RJ2517, San Jose, CA (July 1979).

Werner Vogels is vice president and chief technology officer at Amazon.com, where he is responsible for driving the company's technology vision of continuously enhancing innovation on behalf of Amazon's customers at a global scale.

A previous version of this article appeared in the October 2008 issue of *ACM Queue*.