Eventual Consistency

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 second way is from the server side: how updates flow through the system and what guarantees systems can give with respect to updates.

Client-side Consistency

The client side has these components:

- **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.
- **Processes 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:

Client-side Consistency cont.

- **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 eventual consistency models

- **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 it has updated a data item, always accesses the updated value and will never see an older value. This is a special case of the causal consistency model.

The eventual consistency models con.

- **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 needs to 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 hard to program.

Comments

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.

Server-side Consistency

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

Strong Consistency

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 asynchronous replication 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 the system cannot write to **W** nodes because of failures, 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.

Different cases

In distributed-storage systems that need to 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 need to 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.

Weak/Eventual Consistency

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.

Weak/Eventual Consistency con.

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 **$W+R \leq N$**, then the system is vulnerable to reading from nodes that have not yet received the updates.

Monotonic Consistency

Whether or not read-your-writes, 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 harder 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.

End