# De-mystifying "eventual consistency" in distributed systems

Recently, there's been a lot of talk about the notion of eventual consistency, mostly in the context of NoSQL databases and "Big Data". This short article explains the notion of consistency, and also how it is relevant for building NoSQL applications.

A distributed system maintains copies of its data on multiple machines in order to provide high availability and scalability. When an application makes a change to a data item on one machine, that change has to be propagated to the other replicas. Since the change propagation is not instantaneous, there's an interval of time during which some of the copies will have the most recent change, but others won't. In other words, the copies will be mutually inconsistent. However, the change will eventually be propagated to all the copies, and hence the term "eventual consistency". The term *eventual consistency* is simply an acknowledgement that there is an unbounded delay in propagating a change made on one machine to all the other copies. Eventual consistency is not meaningful or relevant in centralized (single copy) systems since there's no need for propagation.

Various distributed systems address consistency in different ways because there's a tradeoff between speed, availability, and consistency. In some systems, the machine where the change originates will simply send asynchronous (and possibly unreliable) messages to the other machines and declare the operation as successful. This is fast, but at the cost of potential data loss if the originating machine fails before the replica(s) have received the update. Other systems send synchronous (blocking) messages to all other machines, receive acknowledgements, and only then, declare the operation as successful. These systems favor consistency and availability at the cost of performance. Finally, a system might implement some variant of these two extremes (e.g. wait for acknowledgements from a majority of the replicas).

The notion of *eventual consistency* has significant implications for the application programmer, which is easily explained using a simple analogy from everyday life.

Imagine that a person (sender) needs to send a package to another person (recipient) using a package delivery service like the postal service or FedEx. Various options are provided by the delivery service, each with a different cost. For example, the cheapest delivery method might be surface mail, with some uncertainty about when the package will be delivered (e.g. three to five business days). At the other extreme, for an additional cost, the delivery service also provides the guaranteed overnight delivery with delivery confirmation. Depending on the nature of the package contents and the urgency of the situation, the sender can choose the appropriate delivery option, being aware of the cost of each option and also the associated delays and guarantees of delivery. If the sender doesn't care when (or if) the package is delivered to the recipient, then s/he will choose the cheapest option. On the other hand, if the package delivery is critical to the sender, then s/he will choose the more reliable (and usually most expensive) delivery option. In today's world, it is hard to imagine a scenario where the delivery service only provides a single option (either the expensive overnight and guaranteed delivery


Published June 2012

option or the inexpensive, but slow option). The flexibility of delivery options and clear and simple explanation of the costs and delivery guarantees allow the sender and recipient to communicate effectively and efficiently.

Unfortunately, in the NoSQL software world the semantics of keeping multiple copies consistent are not as clear as described in the analogy. Some systems make tradeoffs that favor performance, potentially at the cost of reading stale data (weak consistency). In some cases, the system may need to access data from more than one replica in order to determine whether the value is out-of-date.

Other systems favor updating the replicas (or a majority of the replicas) synchronously, at the cost of performance (strong consistency). Data may be consistent and highly available, but at a higher cost.

A system that lets the application designer choose how changes are propagated is simpler to understand and use. For example, if the application designer knows which updates are critical, he/she can choose the "guaranteed overnight delivery" semantics, in exchange for the associated cost of requiring these guarantees. In particular, if the application is likely to read a data item shortly after it has been changed and wants to ensure that it reads the most recent value (read-my-writes consistency), then fast propagation is important. In other cases, it may be okay to relax the constraints on change propagation in order to gain performance benefits. For example, if the application is simply capturing data, then it is acceptable to relax the propagation guarantees in favor of higher performance.

When choosing a NoSQL system, it is important to understand whether a choice of consistency policy is available. If the database system only supports eventual consistency, then the application will need to handle the possibility of reading stale (inconsistent) data. This is not as easy as it sounds since the responsibility of avoiding the "stale read" problem associated with eventual consistency is left to the application developer.

Oracle NoSQL Database allows the application designer to choose the consistency level required, on a per-operation basis. Per-operation choice of consistency is the most flexible and the most "application-friendly" option since the application designer has a clear understanding and control on the performance as well as the consistency guarantees, without additional complexity in the application program.

To summarize, a per-operation consistency policy similar to that provided by Oracle NoSQL Database provides the most flexibility and best performance while keeping applications simple and easy to understand and maintain.


Published June 2012