Paper Trail

Computer Systems, Distributed Algorithms and Databases

Consistency and availability in Amazon's Dynamo

There is a continuing and welcome trend amongst large, modern technology companies like Google, Yahoo and Amazon to publish details of their systems at academic conferences. One of the problems that researchers at universities have is making a convincing case that their ideas would work well in the real world, since no matter how many assumptions are made there really is no substitute for field testing, and the infrastructure, workloads and data just aren't available to do that effectively. However, companies have infrastructure to burn and a genuine use-case with genuine users. Using their experience and data to discover what does and doesn't work, and what is and is not really important provides an invaluable feedback loop to researchers.

More than that, large systems are built from a set of independent ideas. Most academic papers leave the construction of a practical real-world system as an exercise for the reader. Synthesising a set of disparate techniques often throws up lots of gotchas which no papers directly address. Companies with businesses to run have a much greater incentive to build a robust system that works.

At 2007's Symposium on Operating Systems Principles (SOSP), Amazon presented a paper about one of their real-world systems: "Dynamo: Amazon's Highly Available Key-value Store". It wound up winning, I think, the audience prize for best paper. In this post, I was planning to describe Dynamo 'inside-out', based on a reading group mandated close reading of the paper. However, trying to lucidly explain a dense 12 page paper leads to many more than 12 pages of explanation. So instead, I want to focus on one particular aspect of Dynamo which I think is the most interesting.

Dynamo

Dynamo is a service that provides persistent data storage to other services running on Amazon's distributed computing infrastructure. There are lots of ways to build a service like this: the set of design decisions and tradeoffs that Amazon have made are what makes this an interesting paper.

Dynamo stores completely opaque data objects with accompanying keys which identify them. This is a very simple data model, and completely different to, say, a relational database which stores data with a large amount of structure. The structure in a RDBMS allows complex queries to be made quickly, but at the cost of a very rigid data model. Dynamo instead just treats objects as blobs of bits, and has no idea about any of the semantic content of the object. This disallows it from querying the database to find objects that satisfy some set of conditions, but makes it extremely flexible for storing any kind of data the client might need – without needing to specify a schema and a bunch of constraints.

Dynamo supports just two operations: *get* and *put* that retrieve the most recent value of a key (with some complications that I'll describe later) and update a value respectively. From an interface perspective it could not be more simple. Note that there's not even an operation for deleting data, which is something we'll discuss later.

So Dynamo is essentially a big dictionary service, designed for fault-tolerance which implies running it on many machines, replicating data items on several machines so that if any one fails there's still a working copy of the data in the system. This really is the canonical distributed system. Designing a distributed dictionary involves making decisions about the three big requirements of any distributed data store: *reliability*, *availability* and *consistency*. A system is reliable if it continues to operate correctly in the presence of failures, it is available if it is always usable and it is consistent if the values of any object at any replica don't contradict each other.

Highly Available Systems

Unfortunately, you can't have your cake and eat it. To make a reliable system, the very best way to do it is to have hundreds or thousands of replicas so that you can tolerate many failures. However, to keep all those replicas consistent you have to contact each of them every time someone does a write so that every replica sees the same value. Doing this means that during the time that all the replicas are being contacted, the system has to prevent anyone else from initiating another write to avoid a race condition on the replicas. This impacts availability, as the system is unavailable for writes while consistency is being maintained.

This reflects a basic truth in distributed computing: you have to pick at most two from reliability, availability and consistency. If you compromise reliability (by only having one replica, say) then you can achieve availability and consistency very easily. If you decide that consistency isn't so important, reliability and availability are easy to accomplish.

Most often, it's availability that takes the fall. Availability is easy to recover from programmatically – you simply retry your operation until it succeeds. The cost is poor performance. However, failure of reliability can be catastrophic as your system simply stops working, and failure of consistency means that many assumptions that clients might be making about the data they read aren't true (imagine a banking system where you didn't always get the most recent balance when you asked for it…).

Interestingly, Amazon took a different view with Dynamo. They decided that availability was massively important, as latency had a significant impact on the success of their business applications. Amazon's web applications are built from a chain of services – the web page that you see at amazon.com is built by a large number of separate services, all executing in parallel. The speed with which those services can render your page directly affects your happiness with the web-site, and consequentially the likelihood of you buying that toaster oven. Therefore each service is committed to a Service-Level Agreement (SLA) that stipulates a performance contract which each service guarantees adherence to. These SLAs at Amazon are expressed as 99.9th percentile guarantees, that is that 99.9% of the calls to that service will meet the SLA requirements. Interestingly enough, this explains why some of the myriad widgets that now litter any product page at Amazon sometimes simply do not appear – if they break the SLA the web server goes ahead and sends the page without them in order not to compromise latency to the user. Clearly, at Amazon, responsiveness is king.

So if Amazon are championing availability, what did they compromise? It's always a big gamble risking reliability. In the worst case you end up with a system that simply doesn't work, and making the probability of that non-trivial is the sort of decision that gets people fired. So, instead, the Dynamo team weakened the consistency requirements.

Consistency and Vector Clocks

Consistency of data actually encompasses a wide spectrum of possible models, all of which have been thoroughly studied by researchers in parallel computing. For the purposes of this post, we can restrict ourselves to thinking about only three consistency models.

- *Strong* consistency – every replica sees every update in the same order. Updates are made atomically, so that no two replicas may have different values at the same time.
- *Eventual* consistency – every replica will eventually see every update (i.e. there is a point in time after which every replica has seen a given update), and will eventually agree on all values. Updates are therefore not atomic.
- *Weak* consistency – every replica will see every update, but possibly in different orders.

Strong consistency is clearly the easiest to work with, if you're building an application on top of it. Strong consistency guarantees that a distributed store will act just like a database stored on one machine where it doesn't matter which replica is asked for a copy of an object, the response will be the same. However, it is also the most expensive to achieve, and as we have seen usually comes at the cost of availability.

Weak consistency provides few guarantees to the client. In fact, the only guarantee a read in a weakly consistent system gives you is that the value that is returned was once the value of an object at some time previously. Although weak, this model is not completely useless, but it does allow for the possibility that later updates may be overwritten by earlier ones.

In between the two lies eventual consistency. This model does not rule out being able to read inconsistent values (two successive reads might give two different answers), but guarantees that eventually the system will stabilise and that all replicas will have the same value. Replicas may see updates in different orders, but will be able to infer an ordering on the updates such that the eventual value they achieve is consistent with every other replica. For example, if every update had a timestamp from a global clock, each replica could apply updates in the order they were issued, rather than the order in which they were received. Unfortunately, global clocks don't really exist.

Amazon chose a variant of eventual consistency for Dynamo. The problem with achieving eventual consistency is ensuring that all replicas apply updates in a consistent order. For updates in a strongly consistent system, what is often done is to force every replica to not accept any other writes until every replica has agreed to commit the current one. Because every write is committed only with the agreement of every other replica, each replica sees exactly the same order of writes as every other.

This approach is too heavy-handed for a highly-available system. The problem lies with concurrent writes – those that haven't finished executing before a new one is started. There is, in general, no way to tell which of two writes was issued first without a global clock. Instead, Amazon make a really interesting design decision here. They allow inconsistent values from concurrent writes to enter the system, stored side-by-side, and force the client which reads the values to reconcile them and update them back to Dynamo. So each key now maps onto a *set* of values which are all returned with a read.

The rationale for this (see section 2.3) is that any standard policy for resolving conflicting writes will be too simplistic for any application, especially because Dynamo has no knowledge of the internal structure of object values. Therefore instead the responsibiity is passed back to the application, which has an idea of what the semantics of the conflicting updates are.

The paper describes the shopping cart service as one application that uses Dynamo. Here, from a business

perspective, it's very important that additions to the cart don't get lost, as they represent lost sales. However, it's less important that deletions from the cart aren't lost, as users are typically happy to correct that error themselves. Therefore the shopping cart application can reconcile two conflicting states of a shopping cart by taking the union of the contents of both, which guarantees that no additions are lost.

However, it's not good if deletions are lost as a matter of course – customers might suspect that Amazon is trying to cheat them! Therefore Dynamo does a best-effort job of resolving update conflicts, and only in the case where no automatic reconciliation can be made is the problem pushed back onto the calling application. The mechanism that Dynamo uses is very old and well known. Every update is stamped by the replica that receives it with a *vector clock*. There's not enough room here to fully describe vector clocks, but the basic idea is very simple. Every replica keeps a list of the number of updates it has seen from every other replica. Whenever an update is made, the replica that handles the update increases its update counter in the vector clock, and sends the new clock value along with the update to the other replicas. If a replica receives concurrent updates from two other replicas, it can compare the vector clocks to see – in some cases – which came first, according to a simple rule:

If all update counters in a vector clock V_1 are smaller than or equal to all the update counters in a vector clock V_2, then the update that was stamped with V_1 precedes V_2.

That is, if the set of updates seen at the point that V_1 was issued is a subset of those seen when V_2 was issued, we can conclude that V_2 came after V_1. However, this doesn't work for all possible vector clocks: it could be that the set of updates previously at V_1 only overlap with V_2, and then the system cannot tell which was issued first. It is this case that Dynamo forces the application to deal with. The precise sequence of events required to trigger inconsistent writes is relatively unlikely: two or more clients have to update an old version of an object and have their updates handled by different replicas. However, if failures occur – such as a network partition – then there is a greatly increased chance of inconsistency.

One thing that Dynamo does do is ensure that all versions of an object that are committed to the system are returned when read. This is done through the time-honoured technique of *quorum assembly*. Enough replicas are written to such that the number of replicas subsequently read from are guaranteed to contain at least one instance of every value that has been committed to the system. Conflicting values are stored side-by-side at each replica, therefore they cost more space in the system. One could imagine a denial-of-service attack based on this property, but it would extremely difficult to execute in practice.

(What isn't clear from the paper is exactly how writes are committed to replicas. The paper says that, of N replicas, the system waits for $W - 1$ replicas, where W is the size of the write quorum, to respond, which is a bit better for availability than two-phase commit. However, it's not clear what happens if not enough replicas respond: I think that the assumption is that enough replicas will be live (due to the sloppy quorums mentioned later) that a write will essentially *never* fail except in extreme failure modes, but it's not completely obvious then why the client has to wait for a write to be committed.)

Discussion, Conclusion

There are advantages and disadvantages to forcing applications to resolve update conflicts. As discussed earlier, it might make more sense for the application to take responsibility, having more domain knowledge. Similarly, by ultimately having the application as a catch-all, the effect of bugs can be masked (although the benefits of this are

arguable). On the downside, the applications for which Dynamo is suitable become more limited, as Dynamo cannot always provide a total ordering on updates. You wouldn't want Dynamo to handle your bank account, for example.

The most significant advantage for Amazon is the total availability of Dynamo for writes. Applications can't be prevented from making progress with Dynamo due to heavy load (except in extreme cases), and therefore strong guarantees about latency can be made.

However, application code is made more complex by having to deal with read-time reconciliation. It's hard to decide what the right policy is to take with inconsistent reads, because very often there is no right choice. Writing application logic to cope with the possibility is therefore going to be tricky and potentially error prone. On the other hand, failure modes are made explicit to the application which means they can be dealt with appropriately.

Are there alternatives for maintaining availability? Asynchronous writes are one possibility (asynchronous in the sense that the client does not block) which basically fakes availability to the client at the expense of complicated clean-up if the write is rejected – although this may simply translate into a retry loop. This solution is no good if you expect to be able to read the value that you have written immediately after the write – this is not guaranteed by Dynamo either but only happens when there are almost malicious failure cases.

The paper suggests that inconsistent values are present only in 0.06% of cases. Six out of every ten-thousand doesn't sound like a lot until you realise that Amazon, at peak, is doing millions of transactions a day, each of which must involve multiple reads and writes to Dynamo. So reconciliation is an important bit of code to get right; it's certainly not enough to fail out and throw some kind of exception.

Dynamo, therefore, represents an interesting point in the design space of distributed storage systems. Unstructured data is certainly popular, and you only have to look at one of Jeff Atwood's posts to realise that availability and consistency are biting people designing even relatively simple systems.

Published: August 26, 2008
Filed Under: computer science, Distributed systems, Paper Walkthrough
Tags: Distributed systems : dynamo : paper review

12 Responses to "Consistency and availability in Amazon's Dynamo"

1.  *Gene t* says:
   August 26, 2008 at 1:56 pm

   link to SOSP paper is 404

2.  *Henry* says:
   August 26, 2008 at 2:38 pm

   Should be fixed now, thanks.