Paper Trail

Computer Systems, Distributed Algorithms and Databases

Yahoo's PNUTS

In these politically charged times, it's important for written media to give equal coverage to all major parties so as not to appear biased or to be endorsing one particular group. With that in mind, we at Paper Trail are happy to devote significant programming time to all the major distributed systems players.

This, therefore, is a party political broadcast on behalf of the Yahoo Party.

PNUTS: Yahoo!'s Hosted Data Serving Platform

(Please note, that's the first and last time in this article that I'll be using the exclamation mark in Yahoo's name, it looks funny.)

As you might expect from the company that runs Flickr, Yahoo have need for a large scale distributed data store. In particular, they need a system that runs in many geographical locations in order to optimise response times for users from any region, while at the same time coordinating data across the entire system. As ever, the system must exhibit high availability and fault tolerance, scalability and good latency properties.

These, of course, are not new or unique requirements. We've seen already that Amazon's Dynamo, and Google's BigTable/GFS stack offer similar services. Any business that has a web-based product that requires storing and updating data for thousands of users has a need for a system like Dynamo. Many can't afford the engineering time required to develop their own tuned solution, so settle for well-understood RDBMS-based stacks. However, as readers of this blog will know, RDBMSs can be almost too strict in terms of how data are managed, sacrificing responsiveness and throughput for correctness. This is a tradeoff that many systems are willing to explore.

PNUTS is Yahoo's entry into this space. As usual, it occupies the grey areas somewhere between a straight-forward distributed hash-table and a fully-featured relational database. They published details in the conference on Very Large DataBases (VLDB) in 2008. Read on to find out what design decisions they made…

(The paper is here, and playing along at home is as ever encouraged).

PNUTS allows key-value based lookup as Dynamo does. Key-value pairs are called *records*. However, values may be structured as columns which may be typed. In particular, columns may be 'blobs' which are arbitrary structures. However not all records must have the same columns, and the schema may be updates on the fly at any time. Any column may be the primary key.

PNUTS has a more expressive query language than Dynamo. Users may control the consistency of the data they wish to read by either requiring the latest version or settling for potentially stale records. Range queries are allowed on certain types of record. The consistency model is stronger than Dynamo's, which allowed conflicting records to exist side by aide. PNUTS totally orders all updates to records, with the result that the only record inconsistency that may be observed is staleness.

Consistency

As with Dynamo, PNUTS is happy to sacrifice a little consistency to gain a lot of availability. However, they take the view that Dynamo-style *eventual consistency* is too weak for their purposes. To demonstrate this, they give an example of a user who updates their record, or profile, twice in quick succession. The first time, he removes his mother from the list of people who can access his photos. The second update has him upload his spring break photos, presumably involving some debauchery that mother-dearest would not appreciate seeing.

If the update model was only weakly eventually consistent, the second update could be seen at some replicas before the first. Eventual consistency doesn't guarantee anything about the order that updates are seen in, only that the resulting record will eventually be the same at every replica. There may be intermediate states during which the second update has been applied at some replicas, but not the first. Then if mother decides she wants to check up on son, she might accidentally see his spring break photos. Yahoo are then liable for any ensuing medical bills.

Couldn't this be solved by bundling together updates into single atomic operations? Yes, but it's impossible to predict the updates that a user might want to make to their record. The system has no way of knowing that the change in permissions is going to be followed by a photo upload which is predicated on the success of the previous operation. What is needed instead is a way of totally ordering updates, so that the system can guarantee that an update is only received or applied once all previous updates have been made.

Yahoo does this by relying on an in-house message brokering service (YMB) which guarantees reliable, totally ordered delivery of messages. This gives a nice and clean solution to the problem: there is no way that any replica can receive the second update before the first.

However, this property is only guaranteed at geographical locations – it seems that one logical instance of YBMS runs for each location. As a result of this and other design limitations, PNUTS only guarantees total ordering of updates for each record in the database individually. This has several important consequences. The first is that no advanced database operations like atomic updates to multiple rows are possible, therefore foreign key relationships are unavailable. Updates to two separate records might arrive in different orders at different replicas. The result of a join between them will therefore potentially be inconsistent.

Perhaps the most significant effect of this decision is on the programming model. If you have any user data that you want to update in-order, you have to keep it in the same record. It's easy to see this leading to bloated records, massive de-normalisation and some really annoying situations where you have to re-engineer your data if you didn't think of this in advance. Still, these are no worse than the data engineering problems you get with RDBMSs. It's also important to remember that this isn't a database with a rigorously normalised data model – in fact this is another sense in which designers are prepared to weaken their systems in order to support better performance.

Every record is versioned, which allows clients to compare different versions of records that they might receive. This is exposed directly through the API, so that clients can ask to read *any* record, *any record more recent than version V* and *the latest record*. Clients can also do atomic test-and-set operations with version numbers. This is a great deal more expressive than Dynamo. It would be interesting to know how much coverage each

API call gets – i.e. whether developers automatically use the read call that gives the strongest guarantees because distributed systems are hard to reason about.

Data Storage

Now that we've covered the semantics for accessing records, we can move on to see how the API is implemented. Records are stored in tables, which are split up into *tablets* (much like GFS's chunks) which are the unit of storage for any data centre. The paper refers to the nodes that physically store tablets as *storage units*. Storage units respond to a simple API of *get*, *set* and *scan* requests. The storage is backed by MySQL for tables that are ordered and therefore may be scanned. PNUTS also offers a hash-table table type that doesn't support efficient range scans, but allows (I think) for more effective load balancing. This table type is backed on storage units by a custom-built hash-table filesystem that Yahoo originally designed for their user database.

Each storage unit manages a tablet that contains an interval either of the ordered table key space or the hash table value space. The mapping from intervals to storage units is held permanently by the *tablet controller* which acts as a master for a PNUTS instance. The tablet controller is responsible for mapping tablets to storage units, and then remapping them when load-balancing is required, or recovery from failure is needed. It's not mentioned how the tablet controller recovers from failure – presumably it either logs its behaviour to the YMB (which is able to persistently log messages at each routing step) or to its own local storage and replicates that log, a la GFS.

However, lookups initially go through a router, which maintains a soft cache of tablet mappings. If the mappings should become invalidated, the router can detect that through an error returned by the queried storage unit, and retrieves the new mapping from the tablet controller.

Each record is replicated in several different PNUTS clusters around the world – there doesn't appear to be any local replication which means that if a storage unit should fail its tablets have to be retrieved from geographically diverse locations, which has a high cost. Because YMB doesn't guarantee in order delivery between clusters, only internally to them, each record is *mastered* at a particular cluster which receives the update messages from YMB, commits them and then publishes them to replica clusters. This makes sure that every update gets replayed in a canonical order at each replica.

Persistence of the update log for records is also a 'for-free' benefit of using YMB, which logs the update messages as they travel through the YMB network. Once they are persisted inside YMB they are committed, and will be delivered in order. Reading between the lines, it seems that every record must be an addressable entity in YMB, since each record is individually mastered. Presumably YMB scales to handle this, but it would be interesting to find out how.

Record-level mastering is justified by showing that a high proportion of reads and writes occur from one cluster for each record, so latency is not compromised in the average case if the master is in that cluster. That said, the quoted value is 85%, which still leaves 15% of updates from other clusters needing to go around the world. The system keeps track of the locations of the three most recent updates (by piggybacking them on the updates themselves), so that if updates have changed source the master will be moved to another cluster. The paper says that the most recent 3 updates are used to detect this situation – this number seems low and as though it could result in masters changing location frequently. Most of the change is logical, however, as it's all done by publishing a message to YMB, presumably followed by any updates that are still in-flight and are received at the old master before the new one takes over.

If a storage unit should fail, it's copied from a replica by the tablet controller which publishes a 'checkpoint' message to the replica to ensure that all pending updates are applied. Since the data copy is very expensive, it's suggest that a dedicated 'backup' cluster be located near to the live clusters to mitigate the cost of the copy.

Supporting Expensive Scans

PNUTS supports predicate queries, which is something that Dynamo doesn't offer. The router, through which every query to a cluster is sent, implements *scatter-gather querying*, which simply sends the query to every relevant storage tablet at once, and gathers the data back to send to the client. In effect, it's a bit like a mini map-reduce engine. It has the nice optimisation that only K records are returned with each query, with K chosen to saturate the throughput potential of the client – this makes sure that each query doesn't wait for the entire range scan to complete. The router also returns a continuation query that the client can use to get the next K records if needed (probably transparently to the application itself).

One nice property that this has (also used by Dynamo) is that queries of that type only require the fastest few storage units to respond. In the case of range queries, the router will then scan only one tablet at a time, but then will stop again after K records.

Performance and Conclusions

Some reasonably extensive evaluation was performed. In general the latency numbers are pretty good, but not amazing. Surprisingly, the hash table solution is consistently worse than the ordered table – this is explained as a highly unoptimised custom filesystem layer in which at least 40ms per query is spent. This is bad enough to mask any wins from the better load-balancing – if that is the point of the hash tables. I can't really understand why they exist.

Range scans are expensive, and scale fairly badly as the number of clients increases. Any serious range scans seem to incur at least 500ms latency which renders them useful only for rare queries. This could maybe be mitigated by finer granularity of tablets, better indices or more replication for better availability within a cluster.

So this is Yahoo's take on distributed data stores. The main criticism I have with this paper is that it relies on YMB which doesn't seem to be a published system, yet is responsible for the totally ordered message delivery which allows PNUTS to build all its guarantees upon. It's well known that given totally ordered multicast, building distributed systems is more tractable. It would be good to know how that has been achieved scalably and reliably. For example, can YMB fail?

Another criticism is that failure is expensive, due to cross-cluster tablet copying. There are no performance numbers on that. Yet if we subscribe to the GFS philosophy that failure is now the norm rather than the exception, how can we reconcile that with making failure the expensive case?

Future work is targeted towards making the data model richer through user-accessible indexes, and providing atomic but not isolated access to multiple records. The latter seems to be rather challenging with the guarantees provided by YMB, so it will be interesting to see how they manage it!