Object and Attribute Merging
cs249b

Juan Batiz-Benet jbenet@cs.stanford.edu

March 18, 2011

Abstract

Object exchange makes up a considerable portion of the work that modern distributed applications and services do. On the one hand, these systems often grow considerably large, with many different server nodes needing to read and writ to the same objects. In addition, as personal computing increasingly turns to mobile devices, distributed systems must be able to cope with disconnected but operational nodes. DroneStore is a library that seeks to ameliorate the cost of developing such complex systems by abstracting away the object exchange layer at every node. This paper presents DroneStore's object model, explores attribute merging concepts, and evaluates whether object versioning can be successful.

1 Introduction

With the advent of smart phones and smarter browsers, application logic is getting pushed to richer clients at the edge. To support these massively distributed applications, many large systems exist with the sole purpose of serving object transactions to millions of clients. Recently, many new tools have been introduced in order to achieve high performance at particular layers. For example, Big Table [1] disrupted the SQL RDBMS ecosystem, inspiring new databases [2] that put high-performance before full relational models. Fast in-memory caches [3] and key-value stores [4] have dramatically improved the performance of serving popular objects. New servers [5, 6] handle requests orders of magnitude faster and more predictably than older technologies.

However, these tools do not interface well. kLOC are needed to glue all of these pieces together into a high-performance, reliable system to serve object exchanges. In addition, the piecing together often happens with ad-hoc protocols between two particular layers, and not across the entire system. A large amount of functionality is replicated (e.g. serialization, validation, authentication). This yields complex, convoluted systems that are not easy to change.

1.1 DroneStore

By establishing a common interface, a light-weight library designed explicitly to solve the problem end-to-end could significantly cut down development, performance, and maintenance costs. Instead of writing kLOC of application-specific code to glue layers of services together, the system could be built by simply connecting services that already interface correctly.

DroneStore is that library. It empowers hosts to exchange objects with other peer nodes. The system is designed to be entirely distributed: any two nodes can exchange objects without the need of a master. However, nodes are not meant to be equal; on the other hand, following the models described above, some nodes would run databases, others caches, request servers (load balancing

r40mm

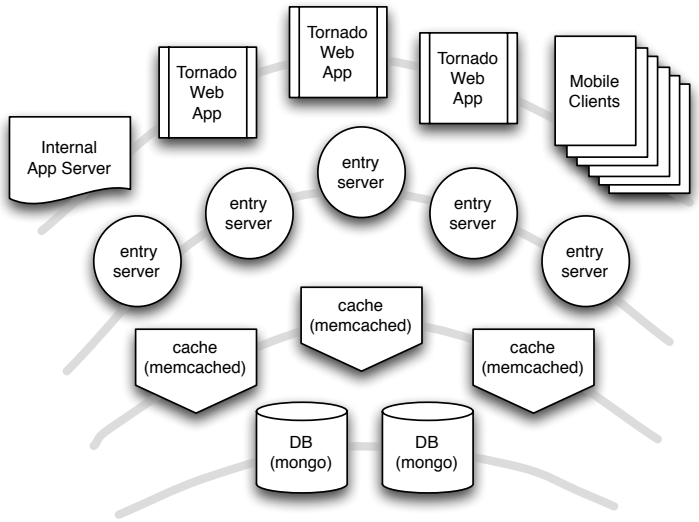

Figure 1: Hypothetical system. Functionality "Layers" are strongly divided. Adding nodes to a layer is straight forward. But, it is costly to (a) change inter-layer configuration; (b) replace a service entirely (e.g. migrate `memcached` to `redis`), along with all its glue code; (c) run implementation experiments (e.g. try running one node with `couchDB` to compare against `mongoDB`).

and authentication), web servers, and so on [1]. The mobile clients themselves are also DroneStore nodes.

DroneStore seeks to provide the common functionality needed by any node of the system to manipulate objects directly at any time, regardless of whether they are connected to the rest of the system or not. However, enabling disconnected nodes to make changes poses significant challenges. Particularly,

- **delayed writes:** disconnected nodes' changes must be stored locally and propagate to the rest of the system once the node re-establishes connection.
- **multiple writers:** as nodes may be disconnected (and should remain operational) for gargantuan periods of time (mobile phones), single-writer semantics (in a distributed sense) cannot realistically be imposed.
- **inconsistent view:** copies of a single, particular object residing in different nodes may exhibit different state. The system must be able to cope with nodes having an inconsistent view (from the omniscient perspective).

Beyond the implementation difficulties associated with the system, the object model and change semantics are crucial to the performance of the system. In order to be successful, DroneStore must implement the right approach to handling multiple objects changing in multiple places. The implementation must be lean and fast, as services utilizing the network should maximize efficiency. In addition, since it aims to be a library that simplifies programming distributed applications, it

[1] Presumably, a master node that indexes where sharded data is located would be easier than building an entirely distributed index.

must expose a simple interface. In other words, the system must be able to merge objects and their attributes correctly, efficiently, and easily (for a programming perspective).

2 Merging Changes

Merging data that has been changed by multiple writers is a general, pervasive problem with vastly different domain-specific solutions. While different fields have different needs, it is useful to look at how the problem has been solved in other applications. Particularly, let us take a quick look at Databases, File Systems, and Source Version Control, and attempt to gain some design insights.

2.1 Databases

Traditional full RDBMS databases and their ACID guarantees are structured to maximize crucial data integrity, trading off performance. They can be modeled as a monolithic bank vault where all data is stored and well organized. All clients line up and issue their requests in the form of query transactions that the database itself organizes and monitors. Though implemented to work in distributed environments, databases impose strong limitations on request expression, performance, and connectedness in order to satisfy the ACID requirements.

By atomizing changes to the entire data into a stream of transactions, databases solve the merging problem by ensuring all changes are sequential and current. That is, the changes are visible to the rest of the system as soon as the transaction completes. Databases keep a history of transactions and can thus play changes forward and back, protecting not only the current data, but multiple snapshots in time. All changes are merged perfectly (a transaction will fail otherwise), and kept for reference, giving full integrity and consistency across the entire system.

However, it does so at great costs in time, space, and usability. Databases tend to be the bottleneck in most applications, as their persistence requirement usually imposes disk access slowdowns. Full RDBMs databases perform so badly (in terms of speed) that performance-conscious companies have invested in large software projects to address the concerns (Google in `BigTable`, Facebook and Twitter in `Cassandra` [**?**], Zynga in `Membase` [**?**], among others). These efforts have shifted the direction of web service databases towards performance, giving up ground in terms of ACID that traditional RDMBSs did not (for instance, relaxing *Consistency* into *Eventually-Consistent*).

The key takeaways for our purposes are:

- ACID is very expensive in both time and space. It enforces single writer consistency and thus requires a fully connected system.
- The server-client paradigm takes away performance control from programmers. Abstractions are great when they perform well, but a system should allow behavior tuning in a straightforward way that places control in the developers.
- Performance critical no-SQL databases have proved Eventual Consistency is acceptable for many integrity-critical applications.

2.2 File Systems

In high contrast to databases, file systems do not come close to ACID. Multiple writers are able to write to the same file, potentially resulting in interleaved data. In addition, rather than hiding behind disk speed, file systems have highly optimized reading and writing performance, making extensive use of caching, memory mapping, read-ahead, delayed writes, and many other techniques.

By sacrificing databases' dogmatic ACID, File Systems maximize performance of computers' slowest bottlenecks (disk, network, etc). Though they place the onus of enforcing consistency in the application writer, file systems are generally very flexible and give the programmer many ways to tune behavior and performance.

In terms of changes, file systems make an obvious but important simplifying assumption: keep the most recent changes. Thus file systems do not suffer the high costs of strict data schemas or preserving large quantities of history to rollback potential bad writes. Yet most data-critical applications store their vital data in filesystems (including databases).

The key takeaways are:

- Keeping the most recent change is good enough.
- Simple, clear, open interfaces hand both control and responsibility to the programmer.
- Preserve critical data through replication on multiple devices or machines.

2.3 Version Control

Version Control systems approach our problem much more directly. Modern systems like `git` [**?**] and mercurial (`hg`) [**?**] allow entirely distributed changes to a set of files, merging changes at specific points in time. And they do so excellently. Files change drastically yet these systems are able to track and version changes accurately (excepting tough merges) in the raw schemaless medium that file systems give. The simplifying assumption here is to treat each line of the file as an attribute, keeping only "attribute" deltas as a representation of the change, drastically saving on space. Of course, this assumption has its limitations: two independent changes to the same line generate a merge conflict, trivial to users that understand the meaning of the lines. [2]

Just like file systems hand over consistency responsibility to the programmer, version control systems leave it up to the users to solve the "tough" merge conflicts. This kind of approach directly resembles *cs249* methodology regarding Exception handling. It works very well for version control, as users both know the correct way to resolve the merge conflict (hopefully) and are interacting directly with the program. This kind of interaction is close to impossible in our system, as it is expected to automatically serve thousands of objects per second. However, modifying the model slightly to put clients of our system as users, we could potentially hand over control during merges to a particular strategy written by the programmer.

Another crucial piece to the puzzle is history. Successful distributed version control systems keep very well tailored history of changes, dating back to the first initial commit. Just like databases, one can rollback to any point on history (or alternate branch of history). The use case strongly motives this implementation, as source code tends to experience multiple revisions that sometimes zig-zag in history or feature sets. In our case, objects are biased towards the latest representation, and the history is less important. It is hard to justify keeping history for millions of small little objects.

One last point to mention is `git`'s distributed work pattern. It is highly effective at minimizing merge conflicts and permitting independent work without imposing centralization contention. `git` emphasizes:

[2] If not done yet, it would be interesting to study whether actually lexing the code and generating parse trees could yield much better conflict resolution performance. Then again, these conflicts are critical and tend to be trivial to the programmer.

1. A working directory contains working copies of all the files in the system. The working directory is in itself a full, working repository, as is every other user's working directory, and any server (bare) copies.

2. Like any other source control system, one is free to modify as many lines (attributes) in files (objects) as they please, and they will appear listed as dirty, but separate from previous versions.

3. Dirty files are staged and `commit`ted to snapshot their versions. This is done *locally*, disconnected from the rest of the system.

4. Instead of using an incremental version number, the version is a cryptographic hash (taken over the change set and time), allowing multiple developers to commit code independently and at the same time.

5. At some point, the user can choose to `push` objects (files, changes, history) to some other (remote) repository, or `pull` objects from another repository to the local one. Any uncommitted changes are not taken into account, only previously committed files. This allows the objects to keep changing during any network bottle-necked tasks occur. It is akin to non-blocking synchronization.

6. When objects from two repositories are merged, their history trees are combined and history is replayed with the right changes interleaved at the right relative points.

7. These processes repeat ad-infinitum.

The key takeaways are:

- Merging whole objects is complex; the problem becomes much easier by merging attributes.
- The distributed model allows effective, non-blocking, disconnected changes to objects.
- Keeping the version history tree is really effective in creating database-like ACID.

3 The DroneStore Object Model

The following sections present the main data structures and the merge pattern.

3.1 Merge Building Blocks

After studying multiple different versioning systems, implementing a few different designs, and coalescing insights, a clear object model emerges to solve object merging in our domain. The building blocks of our model are:

- **Attributes** define particular types; are tuples of {a name, a representation type, a merge strategy}
- **Objects** are uniquely named (keyed) instances of a particular type; are the discrete unit of data.
- **Versions** represent a committed, serialized, snapshot in an Object's lifetime; are sent to remote nodes.

3.1.1 Attributes

An attribute is implemented as a templatized class that allows users to easily build Objects. The simplified interface of an attribute is:

```
1
2 template<class RefType, class MergeStrategy = Merge::LatestStrategy>
3 class Attribute {
4
5   Attribute(RefType _defaultValue, AttributeName _name);
6
7   virtual void valueIs(const RefType &_value);
8   virtual RefType value() const;
9
10  Status status() const; // { Clean, Dirty, ... }
11
12  virtual Merge::Result mergeVersionIs(const Version &_ver);
13
14  /* operators */
15 }
```

listings/attribute.cpp

The RefType is the underlying primitive type that the values of this attribute are stored in. It is meant to be a (or close to a) primitive type, though structs are supported. The MergeStrategy is a functor that will be user to merge versions, more on that later. The name represents the system-wide name for this attribute in this type. The serialized version expresses values in terms of the name expressed. The name should've been another template parameter, but the implementation in C++ is tricky and cluttered otherwise.

There is total freedom; attributes can be derived, RefTypes specified, names chosen (it does not need to match the C++ member name), MergeStrategies selected or derived, etc. The whole design is built to be extremely simple, and highly extensible.

3.1.2 Objects

Objects are the main class that users interact with. Taking lessons from other systems, this design maximizes simplicity of use. To define a merging type, all a user must do is derive from Object and define Attributes. For example:

```
1  class User : public DroneStore::Object {
2  protected:
3
4    User(const Key &_key, DroneStoreManager::Ptr _mgr) :
5      DroneStore::Object(_key, _mgr),
6      name_("", "name"),
7      phone_("", "phone"),
8      birthdate_(Time::now(), "birthdate"),
9      score_(0, "score"),
10     calls_(0, "calls") {
11   }
12
13   Attribute<String> name_;
14   Attribute<String> phone_;
15   Attribute<Time> birthdate_;
16   Attribute<U32, DroneStore::MaxStrategy> score_;
17   Attribute<U64, DroneStore::CounterStrategy> calls_;
18
```

```
19    /* ... */
20  };
```

listings/example.cpp

Everything else is handled for the users. They are free to assign values to the attributes as they please, and call `commit()`, which will snapshot the object at that particular moment and with those attribute values, and start propagating the changes in the background (configuring how to connect to the rest of the system is still in the drawing board).

3.1.3 Versions

Versions are snapshots of a particular object at a particular moment. They are created when users `commit()` changes to an object. Taking lessons from `git`, version numbers are cryptographic hashes of serialized object representations; thus changes can be made independently and merged at will.

Unlike `git` or databases, only one version is kept: the last one. Taking lessons from file systems and from average web service usage, one is most interested in the values of objects that reflect the object's current state. It is difficult to justify keeping all the history of quickly-changing objects around just to ease merging and be enable jumping to a previous snapshot of the data.

Versions allow access to attribute values at the point of saving, and include any meta data used by Merge Strategies. They are represented compactly, serialized via `BSON`, ready to ship over the network. The (slightly simplified) interface of Versions is:

```
1  class Version {
2  public:
3
4    Hash hash() const; // hash.hexDigest() for human readable version hash.
5    Hash parent() const;
6
7    string shortHex(int _length=6) const; // hash.hexDigest().substr(0, len)
8
9    ObjectRep *serialRep() const; // access to the underlying data.
10
11   Serial::Element attribute(AttributeName _name) const;
12
13   Time timeCommitted() const;
14
15   bool operator==(const Version& v) const;
16   bool operator!=(const Version& v) const;
17
18 };
```

listings/version.cpp

3.2 Merging

The merging pattern is also quite simple. Insights from both `git` and file systems helped refine what is a very simple but flexible mechanism. The key idea is that different attributes can be categorized by update pattern and what they represent. These categorizations reveal very distinct merge operations that do apply to that category but not the others. In other words, a particular attribute has some meaning associated with it that the programmer understands, and through which he picks (or defines his own) a particular MergeStrategy to do the merging.

In the above example, three attributes (name, phone, and birthdate) only define a RefType in the attribute template. The default MergeStrategy is MergeLatest, which, given two values for an

attribute, always picks the latest changed one. This is the most common case (so common that for file systems it is the only case). Merge Strategies are allowed to store a bit of meta data along with an attribute in the serialized version representation. And every time `valueIs(...)` is called on an attribute (called by `operator=(...)`), the attribute notifies its associated MergeStrategy. Thus, MergeLatest can record the time when the attribute is changed. When another version is merged, MergeLatest decides based on those times.

But sometimes, certain attributes do not belong in that category! Consider a monotonically increasing counter (such as packets received from a given address). If multiple machines update that counter independently, and we merge using MergeLatest, we will clearly get incorrect results as the latest value will be picked. Hence the parametrization of MergeStrategy. This abstraction is what makes the system flexible enough to actually be applicable. For the packet count case, we could use a CounterStrategy, that logs independent count tuples as {context, count} (where context could be machine, or instance, etc). Thus, whenever the attribute is assigned a new value, MergeCounter increments the counter for the current context (and the total sum). Whenever a merge is made with another node, the entries are interleaved, and the total sum recalculated (avoiding over and under counting).

Thus, MergeStrategies can be defined easily to fit the precise needs of a particular attribute. They are merely a functor (deriving from MergeStrategy) that can be called on a particular attribute and a version. The MergeStrategy functor decides whether the local or the remote version is best to keep, and updates the attribute accordingly. The current library contains a few MergeStrategies and more will be implemented.

Below you can see examples of the implemented strategies.

```
1  class Strategy ... {
2  public:
3    virtual void changedAttributeIs(Attribute *_attr) {}
4    virtual Result operator()(Attribute *_attr, const Version &_ver) = 0;
5  };
6
7  //-------------------------------------------------------------------------
8
9  class LatestStrategy : public Strategy {
10 public:
11   LatestStrategy() : Strategy(), updated_(0) {}
12
13   virtual void changedAttributeIs(Model::Attribute *_attr)
14     { updated_ = Time::now(); }
15
16   virtual Result operator()(Attribute *_attr, const Version &_ver)
17
18     Time _updated = _ver.attribute(_attr->name())["updated"];
19     if (_updated <= updated_)
20       return MergeSucceeded_;     // Time is not newer! get out.
21
22     _attr->valueIs(_ver.attribute(_attr->name())["value"]);
23     updated_ = _updated;        // override.
24     return MergeSucceeded_;     // Success!
25   }
26
27 protected:
28   Time updated_;
29 };
30
31
```

```
32 //-----------------------------------------------------------------------------
33
34 class MaxStrategy : public Strategy {
35 public:
36   MaxStrategy() : Strategy() {}
37
38   virtual Result operator()(Attribute *_attr, const Version &_ver) {
39     Serial::Element _value = _ver.attribute(_attr->name());
40     if (_value <= _attr)
41       return MergeSucceeded_;
42
43     _attr->valueIs(_value);
44     updated_ = _updated;       // override.
45     return MergeSucceeded_;    // Success!
46   }
47 };
```

listings/strategies.cpp

4 Discussion

So far, the project has succeeded in proving that object merging can be done in a relatively simple way. A simple interface is exposed to clients, with details abstracted away. However, control still remains in the programmers hands, through deriving defined types.

Also, strong guiding principles were distilled from studying other projects and by trial and error. Once identified and justified, these principles laid the foundation of the latest design and proved successful. Future work should thoroughly test these principles and design.

4.1 Code Complexity

After a few iterations on the design, and simplifying both the object model and merge pattern, templates (and template meta programming) provided for a long, but straight forward implementation of the concepts. The MergeStrategy abstraction relieves the main library code from worrying about edge conditions and particularly complex merge schemes.

The amount of code written for the entire project is:

Part	LOC
Tests	3584
Sources	2808
Framework	717
Example	223
Total	7332

7 kLOC is not a large amount of code, but is also not very small. Overall, the complexity was not in implementation, but in refining the design. It helped considerably to have worked on the problem for some time prior to this project, with lessons already learned and some paths crossed out. In particular, it helped to take a step back and evaluate other systems from an academic point of view, and draw correlations and clear distinctions in design and philosophy.

What surprised me was the particularly small code size of the example code. The objects were defined and interacting with just 223 lines of code! Though this is likely to grow once networking is implemented, it is impressive that the object definition got to be so small.

The code base is written with `cs249` principles in mind. The code so far exhibits lots smart pointers, activities, notifications, audit, named descriptions, etc. I wish I had gotten to test out MemPools, however I am not at a point to benchmark memory performance yet.

4.2 Alternative Approaches

Object merging as presented here has significant advantages over other alternative approaches in the space, however it may not perform as well or be suited for every case yet.

As discussed in class, the traditional approach to a distributed system suggests to have a master (with replicas) and a series of child processes that manipulate objects. In our case, they would be web servers serving out objects.

While the stateful-proxy pattern described is very similar to our approach, object merging differs significantly in that agent processes can be disconnected from the rest of the network. This is a serious advantage over distributed systems that do not allow this sort of behavior, as it allows a much broader application client base.

Object merging is certainly less well suited to critical ACID-requiring tasks, particularly within one internal network, as it provides a larger data format and lots of code built to deal with object merges.

4.3 Data Format Alternatives

When implementing services that will utilize the network (let alone be primarily dedicated to serving through it), it is crucial to analyze the schema and to compare the amount of data sent to the amount of information.

DroneStore is using `BSON`, a binary data encoding derivative of `JSON`. `JSON` has become extremely popular for its ease of use in web applications. It is now the standard way to send messages for many kinds of services, even though it is not particularly efficient. `BSON` sought to remedy this by providing a binary packed and efficiency oriented version. `BSON` still keeps the schema, and thus is a constant factor less efficient than `protocolbuffers` or `thrifty`. It depends on the exact schema of the data, but it ranges from 2x to 5x more bytes.

In `BSON`'s defense, however, the significant benefits come from easy interfacing with multiple other systems and APIs, simple description, and its characteristic schema-full fast prototyping and format independence. For a system that needs to send various user-defined types, it saves the hassle of forcing users to comply to a standard message format, have them define their own `protobuf` or `thrifty` format files, and overall increases the ease of use. The performance hit seems acceptable at this point, particularly while still writing the bulk of the system.

4.4 Insights Regarding Merging

Various important insights made themselves clear throughout the process. Some were mentioned above, but here they are summarized:

4.4.1 DOs

- Eventual Consistency is acceptable for many integrity-critical applications.
- Keeping the most recent change is good enough for many cases (but not all!)
- Simple, clear, open interfaces hand both control and responsibility to the programmer.

- Preserve critical data through replication on multiple devices or machines.
- Merging whole objects is complex; the problem becomes much easier by merging attributes.
- The distributed model allows effective, non-blocking, disconnected changes to objects.

4.4.2 DONTs

- ACID is very expensive in both time and space. It enforces single writer consistency and thus requires a fully connected system.
- The server-client paradigm takes away performance control from programmers. Abstractions are great when they perform well, but a system should allow behavior tuning in a straightforward way that places control in the developers.
- Keeping the version history tree is really effective in creating database-like ACID, however at the rate of change of in-memory objects, this seems out of the question.

4.5 Future Work

- Test more MergeStrategies. Some particular cases (like large collections) seem problematic off hand for deletions.
- Building out the network stack with event-driven programming.
- Finish writing the library features (persistence, etc).
- Explore simple security mechanisms using capabilities.
- Test of fire: run it in production for some time.

References

[1] Fay Chang, Jeffrey Dean, et al., "Bigtable: a distributed storage system for structured data," in *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, Berkeley, CA, USA, 2006, pp. 15–15, USENIX Association.

[2] "mongodb," `http://www.mongodb.org/`, Jan 2011.

[3] "memcached," `http://memcached.org`, Jan 2011.

[4] "redis," `http://redis.io`, Jan 2011.

[5] "Tornado," `http://www.tornadoweb.org/`, Jan 2011.

[6] "Nginx," `http://nginx.org/`, Jan 2011.