Serialization, Persistence, and Immutability

Due: October 4, 2015, 11:59:59 pm

v0.2

This project builds on the previous to add persistence. You should end with a project that saves files and can retrieve them the next time it starts. Though your system won't yet have a way to retrieve old versions of file (a la `Elephant`), all versions of all nodes and directories are stored forever.

We'll reach persistence by dividing each written file into chunks, and writing those chunks to a key-value store. The high points of this project include the following:

- Rabin-Karp fingerprints: use content-defined chunk boundaries to split large files, as in `lbfs`.
- Data will be made persistent by writing to a key-value store, in this case `leveldb`.
- Nodes will be serialized using Go's native JSON encodings.
- *All objects written to `leveldb` are immutable.*

1 Setup

You will need to install both `leveldb`:

https://code.google.com/p/leveldb

and go bindings:

https://github.com/syndtr/goleveldb

The code can be installed via `go get github.com/syndtr/goleveldb/leveldb`. Be sure to use `goleveldb`, not `leveldb-go`.

Also, I'm now using a version of `getopt` to parse command lines, you might want to use that as well. See the project `P1` source for an example.

As this project builds on the previous, you need to have a working base. You can download my code from http://dss.kelehers.me/projects/p1.go. This code is rough, but does fulfill the requirements of the project. Bug-fixes and improvements are welcome.

1.1 Terminology

We name blocks of data through cryptographic hashes. In this project you should use `SHA1` to create a hash of an item, and `base64` to create an ASCII representation of that hash value. Both are easily created by calling packages in Go's standard library. Go objects are first serialized in a JSON representation (see Section **??**), and then hashed as above.

- A file's metadata is contained in a structure called a `DNode` (instead of DFSNode, which just seemed verbose).
- The *signature* of a data block is the `base64` version of a `SHA1` hash of that block.

- The *signature* of a `DNode` is the `base64` version of a `SHA1` hash of a JSON version of that `DNode`.
- A *key* refers to a string used to store an item in the key-value object store. For all items except the "Head" (see below), the key is the item's signature.

2 Strawman Implementation

I am first going to describe a straightforward implementation for this project. The next section will describe a number of optimizations.

Consider creating and writing a small file `foo.c` under an otherwise-empty top-level directory exported by your filesystem. We need to save three pieces of information: the file's data, file's `DNode`, and the `DNode` of the containing directory, which is the root.

There are three different types of objects that are written to the object store: head pointers, `DNode`s, and raw file data. `DNode`s and file data are written to the object store using SHA1 hashes of their content, hereafter referred to as "signatures", as keys[1].

A `Head` object is an instantiation of the following type:

```
type Head struct {
  Root    string
  NextInd uint64
  Replica uint64
}
```

"Root" is the current root's signature. 'NextInd' is the next available "inode number", or "inumber", saved here as the field `INode` in the attributes. All versions of the same file have the same inode number, and this number must be unique among all files. Ignore `Replica`. The current "Head" variable is written to the object store by writing a `base64` version of the JSON'd representation of that variable, with "head" as the key.

2.1 JSON serialization

Head pointers and `DNode`s must first be serialized through `json.Marshal` and `json.Unmarshal` from the standard library. There are two tricky aspects. First, the package will only marshal *exported* fields of a struct, i.e. capitalized fields. This can be convenient, as fields that need to be saved should be capitalized and those that can be tossed should start with a lowercase letter. For example, the following is a possible definition for a `DNode`:

```
type DNode struct {
   Name       string
   Attrs      fuse.Attr
   ParentSig  string
   Version    int
   PrevSig    string
   ChildSigs  map[string]int
   DataBlocks []string

   sig        string
   dirty      bool
   metaDirty  bool
   expanded   bool
```

[1]SHA1 is insecure. You could use SHA256 instead, it just makes all the hashes larger.

```
    parent     *DNode
    kids       map[string]*DNode
    data        []byte
}
```

The first seven fields are included in the marshalled version; the others are not.

Every distinct `DNode` (one per version of a file) is uniquely identified by its signature, which is the `base64` version of the hash of its JSON-marshalled value. Each time a `DNode` is modified and written to the object store, its hash (signature) is different and it is therefore stored under a different key. The signature allows a saved directory, for example, to precisely specify the appropriate snapshot of the `DNode`'s state.

"kids" is not capitalized and will not be saved. This is appropriate, because it is not needed to persist a directory. However, the saved version of a directory *does* need to be able to name the files/versions it contains. These are named by "ChildSigs". "kids" is used only during runtime, and "ChildSigs" is used only in the persistant representation of the node.

Likewise, the saved version of the node should not contain the file's data. Instead, it should contain signatures naming or more blocks of data that contain the file's data.

2.2 End-to-end

The following, then, is the list of tasks your code will need to accomplish for this simplified version. Note that this "straw man" does not include any concept of the separate `Flusher()` thread discussed below.

I will describe here the tasks to be accomplished when a 32-byte file, `foo.c`, is created in a new file system. I am assuming that you already have a root `DNode` in memory, that this `DNode` has version 1, that this version has already been written to the object store, and that the signature from that version has been written to the store with a key of "head".

You will receive three important FUSE calls. Your tasks will be something like the following:

1. create("foo")
   (a) Create the `DNode`, with Name "foo", Version "1", default attributes, and length zero.
   (b) Serialize `foo`'s new `DNode`, write to object store with new signature as key.
   (c) Modify the root `DNode` to include the above signature in ChildSigs, set the version to 2.
   (d) Serialize the new root and write to object store.
   (e) Update a global `Head` variable with the new root signature, and write to the object store as the data associated with key "head".
2. write("foo", 32 bytes of data at offset 0)
   (a) Modify data and attributes of the file in-memory.
3. flush("foo")
   (a) Update `foo`'s version to 2.
   (b) Create new signature of `foo`'s data, write data to object store with that signature as key. Add that signature to `foo`'s `DataBlocks` field.
   (c) Serialize `foo`'s `DNode`, write to object store with the modified `DNode`'s signature as key. Insert this signature into the root's `ChildSigs` array.
   (d) Update the root `DNode` to version 3, serialize, and write to object store.

(e) Update global `Head` variable, and write root version 3's signature to the object store as data, with "head" as the key.

Note that last item. Since we write it last, it serves to make the entire change atomic, including modification of all `DNode`s between the affect file and the root. Think about it.

Note that we are potentially building up a great deal of garbage in the key-value store: each new version of a `DNode` makes the previous versions obsolete. However, we ignore this now because nodes take little space, and we will leverage those obsolete node versions to provide user-visible versioning in a later project.

To restart the system, we request key "head" from the object store, and get back data that is the signature for the most recently saved version of the root directory.

3 Optimizations

The previous section described a relatively simplistic approach to persistence. This section will augment that approach with a number of techniques that will improve the system's performance.

3.1 Chunking

So far we have assumed that the entire content of a file is saved as one object in the object store. However, we can reduce storage and communication requirements by taking a chunking approach instead. As described in the LBFS paper, we will use Rabin-Karp fingerprints to divide files into content-defined "chunks" before writing file data to the store.

See Wikipedia for a detailed explanation of Rabin-Karp chunking. Your chunking approach should use 31 as your prime, 32 bytes as the length of the hash, and min/target/max chunk sizes of 2048/4096/8192 bytes.

To make things a bit easier, you can find a C version of the algorithm in the starter files for this project. You just need to translate this to Go. The translation is quite straightforward, though there are NO implicit type conversions between different types of integers: you need to use type assertions at each conversion.

3.2 Versioning and Writing to the Object Store Asynchronously

The example of new file `foo.c` in the previous section synchronously wrote `DNode`s to the object store immediately after a flush. The example showed how this can cause a file copied into the new file system to have two distinct version, where only one is logically necessary.

Rather than have two versions of `foo.c`'s `DNode` (one newly created, a second after data has been added), a lazy writing approach could delay and end up writing only a single version to the object store. Likewise, if we copy in a directory that contains 100 files, we'd like to not create 100 new versions of the directory's `DNode`.

Making the versioning and writing to the object store asynchronous and periodic is the key to limiting version explosion. We mutate objects in-memory for some period of time, then briefly "stop the world" and write changes to the object store.

The following is one approach:

1. Add a new `metaDirty` boolean to nodes.

2. When a file or directory is created or modified, that node, *together with all the parents up to the root*, are marked "metaDirty". While modified file data *is* written to the object store during any `flush()` call, the modified `DNodes` are not.

3. A second “flusher” thread is created to do the periodic writes. The flusher runs periodically, maybe every five seconds or so, starts at the root, and writes all “metaDirty” nodes.

   Note that this really needs to be done post-order in order to avoid referencing data that is not yet in the object store...

Threading in Go is accomplished through “goroutines”. Functions are run in distinct goroutines by writing “go func(...)”instead of “func(...)”. Goroutines (including the “main” thread, which can also be thought of as a goroutine) communicate with each other through *channels.*

Two last details. Running the flusher every five seconds or so could be accomplished with signals, or just `sleep` calls in an infinite loop. However, the “idiomatic” (preferred for Go) approach is to use a channel to connect the flusher to a `Timer`.

Second, the flusher should really be synchronized with respect to all the function calls from `bazil/fuse` into your code. A reasonable approach for this project is to define a single global mutex, lock it while in any bazil-called function, and lock it in the flusher when executing (not while sleeping). Again, however, the idiomatic approach is to use a channel as a synchronizer between the flusher on the one hand, and all the FUSE calls on the other hand.

4 Testing

There are many moving parts here, and you will have to write quite a bit of code. *Test parts separately* when possible. Go makes it easy and fast to take small sections of code and run them by themselves. For example, I used `marshal.go` (below) to help me understand JSON marshalling. You can do similar modular testing for the key-value store, for the fingerprint chunking, etc.

For the more ambitious, Go has a lightweight testing framework.

5 Grading

Your code should support the following command-line arguments:

- **-d** - toggles printing of debugging output. Default: `false`
- **-m <mountdir>** - Default: `dss` in current directory.
- **-n** - “new file system”: reset the file system to initial state.
- **-s <leveldb path>** - Path to pass to `leveldb` open. Default: `db` in current directory.
- **-t <time string>** - Time travel: specifies that the file system should reflect a specific time, AND that the resulting file system is read-only. Example: `-t "2015-09-15T12:40:28"`

I will assign grades vaguely as follow:

- 50 pts - persistence works to the leveldb key-value store
- 15 pts - rabin-karp finger-print-defined blocks
  - 10 pts - if you have static 8k blocks rather than Rabin (e.g. a 16,564-byte file will consist of two 8192-byte blocks, and a 200-byte block).
- 20 pts - asynchronous writing of data via goroutine(s)
- 15 pts - time travel

6 Files

The archive http://dss.kelehers.me/projects/p2.tgz contains the following files:

- `main.go` - Scaffolding for P2, which you don't have to use. References files in `dfs` in same directory. You run, for example, like `go run main.go -d -m dss2`.
- `marshal.go` - simple, running example of marshaling
- `sget.go` - retrieves data corresponding to specific key, prints. You should use this on your JSON'd `DNode`s, not file data. Can also be used to print out all keys in the object store.
- `rk.c` and `rktest.c` - C version of rabin-karp chunking. Compile ("gcc -o rk rk.c rktest.c") and run on `testfile`. Insert two spaces at the beginning of the first line and try again. The results should differ only in the length of the very first chunk. similar. Your `go` version of finger-printing should produce the same results.
- `testfile` - File to test chunking on.

7 Submit

To the submit server as before.

8 Addendum

Project #4 will augment this project to build a replicated log via distributed consensus with Raft [**?**]. Making your flusher create and store a "log segment" containing all modified nodes, rather than storing each node individually, gets you closer to where we will be ending up.