Replication, and a Cloud Client

Due: November 22, 2015, 11:59:59 pm

v0.1

1 Introduction

In this project you will extend your file system to a replicated system. You will be able to start up multiple replicas, have them talk to each other, and have changes made to a shared file system reflected on the other. Further, these replicas can even live on the same machine, having their own mount points and leveldb repositories.

2 Setup

For this project, you will structure your code as a single replica. To run a system with two replicas, manually start each with command-line arguments that allow the replica to configure itself. The following is the complete set of command-line flags you **need** to respect:

- `-c` specifies an empty (clean) DB for the local replica. This may have been `-n` on prior projects.
- `-d` turns debugging flags ON (default should be OFF)
- `-f <secs>` specifies the flusher's period in seconds. For example, `-f 60` tells your replica to only flush changes (both to disk and to other replicas) only every 60 seconds. Default should be five seconds.
- `-m <mode>` The mode can be either "none", "eventual", or "strong". Default is "none"
- `-r <rtag,rtag,....>` specifies at least a single replica tag. The first tag will be used as the replica's own, any additional tags will be assumed to be other active replicas.

The config file will consist of a series of lines like the following in a file named `config.txt` in the same directory as `main.go`:

```
# lines beginning with '#' are ignored
hyper,1,/tmp/kel1,/tmp/dbkel1,128.8.126.55,3000
hyper2,2,/tmp/kel2,/tmp/dbkel2,128.8.126.55,3010
hub,3,/tmp/kel1,/tmp/dbkel1,216.164.48.37,3000
triffid,4,/tmp/kel1,/tmp/dbkel1,128.8.126.119,3000
```

The lines specify a network endpoint's `name`, `pid`, mount point, where the DB files are located, a network address, and a port. Points:

- Your replicas should create the mount point if it does not already exist.
- As implied above, your code should work with multiple replicas on the same machine, as long as their mount points and DB directories are different.

2.1 Save Pete's Sanity

I am mandating a new project layout. Assume your code consists of a file `main.go` and a subdirectory named `dfs`.

- I will untar your directory underneath `go/src/dss/p4/<blah>`, where `blah` is your directoryid.
- You MUST reference your subdirectory with a absolute pathname: like:

```
import (
     "dss/p4/blah/dfs"
)
```

- I will run a replica as `go run main.go -c -r hyper,hyper2`, for example, to denote a clean DB and a replica that will listen to port 3000 on `hyperion.cs.umd.edu` and talk to another replica on the same machine, but at port 3010.

3 Communicating Replicas

Your replicas should cooperatively present a shared namespace across all of their mountpoints. Replicas communicate via *flushing*, *demand fetching*, and *sending heartbeats*. Remote communication is through either `zeromq`/protobufs or RPC/gobs.

3.1 Flushing

Replicas flush to other replicas at the same time changes are flushed to the key-value store. In fact, both flushes can be accomplished in the same routine.

A flush is accomplished by sending a `snapshot`, consisting of file recipes for any new or modified file and directories (and ancestors all the way to the root). The recipes should be arranged postorder, so that the new root is the last recipe.

Once changes get to a remote replica, they need to be unmarshalled and then incorporated into that replica's view of the file system.

3.2 Demand Fetches

Only new versions of directories/files should be published. Non-local data chunks should be fetched on demand from the remote replica that created them.

Demand fetches have to have a destination. For this project, each saved version should record the replica `pid` (derived from the config file and the replicas name) of the replica that created that version. This replica is guaranteed to have a copies of the needed chunks.

3.3 Heartbeats

Replicas should occasionally send *heartbeat* messages to all other replicas. The heartbeat is essentially a degenerate flush, consisting mainly of the most recent root signature seen by the sender. This is useful both in bringing new replicas up to date, and in implementing eventual consistency.

4 File System Consistency

You will build three different versions of the file system, w/ differing levels of consistency:

1. *best effort* - incoming snapshots overwrite the root, potentially obscuring local changes not seen at the source of the incoming flush.

2. *eventually consistent* - incoming snapshots are *merged* w/ the local snapshot, w/ conflicting versions resolved by their creation timestamps. Only directories are merged, not files.

3. *strongly consistent* - token-based serialization: only one replica has the ability to access data (write *OR* read) at any given time, and the i'th holder of the token must see all updates made by previous holders.

For example, assume a file read comes from fuse to your code. The local replica must obtain the token **before** the read, which may be *owned* by a remote replica. Retrieval of the token from a remote replica should also bring in the most recent root signature, which is then used to update the local root.

Local updates must be committed (flushed) before releasing the token. Assume a replica has modified/dirty data for file `foo`, but receives a token request before `foo` is flushed. The replica must implement a complete flush before releasing the token, and include the latest root signature with the token release message.

5 Node Aliasing

Among many other issues, node aliasing is one of the most important. Bazil interacts with your code through the `fs.Node` interface. In class, we've discussed defining your own types (DNodes, or perhaps multiple types such as Dirs and Files), and returning these as the result of callbacks from Bazil (the `root()` callback from the file system you pass `fs.Serve()`, and `fs.Node.Lookup()`).

This works fine for the first three projects, but there exists an aliasing problem that must be handled in this project. Assume that you use a pointer to some type named `DNode` to implement Bazil's `fs.Node` interface. So your `Lookup()` implementation will return `*DNode`.

The problem is that Bazil will cache that pointer value and use it to make calls to `GetAttr()`, `Readdir()`, etc. No worries, unless you actually start using a different instance of `DNode` (different address, etc.) to implement that same object. Now Bazil is making calls to the wrong version of your object.

Why might you do this? As an example, Bazil caches your `*DNode` for root. When you apply new changes from another replica, this invariably involves a new version of root, which will be allocated at a different location in the heap. Bazil is now caching a pointer to the wrong root object. Similar issues exist with any directory.

Solution approach 1: Write incoming changes to the existing node structures. This is somewhat complex, but can be done.

Solution approach 2: Use a layer of indirection between Bazil and your objects. For example, assume:

```
type Path string

func (Path) Attr() fuse.Attr {
  .....
```

If you return a `Path` object to `Lookup()` calls, all your functions (`GetAttr(), Readall()` etc.) will get called on `Path` objects. The first line in each such method would then be a simple `map` lookup from the `Path` to a `*DNode`.

The main virtue of this approach is that `Path`'s do not change, even as the underlying `DNode` objects do. While conceptually simple, you must be careful to update the `Path` map when creating/deleting files and directories, when renaming, and probably when incorporating a new root node from a remote replica.

```
package main

import (
        "fmt"
        "launchpad.net/goamz/aws"
        "launchpad.net/goamz/s3"
        "log"
)

func main() {
        auth := aws.Auth{
                AccessKey: "ASDFASDFASDFASDK",
                SecretKey: "DSFSDFDWESDADSFASDFADFDSFASDF",
        }

        connection := s3.New(auth, aws.USEast)
        mybucket := connection.Bucket("motefs2")
        res, err := mybucket.List("", "", "", 1000)
        if err != nil {
                log.Fatal(err)
        }
        for _, v := range res.Contents {
                fmt.Println(v.Key)
        }
}
```

Figure 1: Simple Go program to list all keys in bucket "motefs2".

6 Amazon S3

Optional! Folk say: "Use the cloud! The cloud solves all ills!" Fine, we will use the cloud.

The use of the `leveldb` key-value store makes it easy to swap in use of Amazon's Simple Storage Service (S3) instead. The following is an example Go program that talks to Amazon and lists all key names in a specific "bucket".

For this to work, however, you must:

- Have an account on the S3 service. Amazon gives out free accounts allowing reasonable bandwidth for the first year. After that, the rates are still quite low (I've so far rung up a charge of $.06).
- Create your own access and secret keys (possibly through Amazon's console).
- Create a bucket (as above).

Substitute the keys and bucket name into the file in Figure 1.

Many docs of a forked version of `goamz` here.

7 Testing

Given the above, I will test your code as follows:

1. by attempting to start two replicas on `hyperion` through:

   ```
   go run main.go -c -hyper,hyper2
   ```

   from `hyperion`, and

```
go run main.go -c -hyper2,hyper
```

The replicas should behave in all respects as if they are distinct machines.

2. copy `redis` (C proj code posted to piazza) into the first mount
3. compile it from the other
4. go back to the first mount, remove a `.o` file, and re-make

Careful: OSXFuse (or the mac port of Bazil) caches both data and metadata aggressively, meaning that your code might not be given the opportunity to show a version changed by a remote replica because you never get a `GetAttr()` or `ReadAll()` call. **The Ubuntu version seems to cache much less aggressively and should be where you test the second version of this project**.

8 Testing and Grading

I will test and assign points to your program as follows.

- **[25 points]** I will bring up two clean (empty DB via `-c`) replicas. Copy `redis` into one mount point, see that it appears in, and is compilable, in the other. [*best effort*]
- **[25 points]** Bring up a third clean replica. See that it soon populates and can be used like the first two. [*best effort*]
- **[25 points]** Bring up two clean replicas, w/ 60-second flush intervals (`-f 60`). Create `foo` in one, and `bar` in the other. Verify that we end up a directory with both. Note that the first replica should be broadcasting a new root w/ only `foo`. The second will be broadcasting a new root that contains only `bar`. [*eventual consistency*]
- **[25 points]** Bring up two clean replicas w/ long timeout (`-f 60`), and then:
  1. create `foo` in the first
  2. read and append to `foo` from the second
  3. read and append to `foo` from the first

  all within the 60 second window. [*strong consistency*]
- **[25 points extra credit]** Implement the S3 client.

The maximum point total is 100.

9 Submit

As described above, your code should run from `go/src/dss/p4/<blah>`, where `blah` is your directory ID. Create a tarball as follows:

tar cvfz keleher.tgz keleher

with YOUR name instead of mine. Upload "<dirid>.tgz" to the submit server.

Your directory should be set up to run in GOPATH. Your directory should consist of a single file named "main.go", the config file (named "config.txt"), and the bulk of your code implementing a different package below that. For example, my code is set up in `go/src/dss/p4`. Underneath I have:

```
keleher  - main.go
         \ config.txt      / fs.go
           dfs/   ---------  fuse.go
                             \ flusher.go
```

where `fs.go`, `fuse.go`, and `flusher.go` implement the package "dss/p4/keleher/dfs". "main.go", of course, implements package "main" and imports package "dss/p4/keleher/dfs".