Project 4: Puddlestore

Due: 11:59 PM, May 10, 2022

Contents

1 Code Exchange

Remember, if you write code on a department machine, you must use go1.17 instead of just go (i.e. go1.17 install or go1.17 test). Add `alias go=go1.17` to your `~/.bashrc` for convenience.

```
git clone github.com/brown-csci1380-s22/puddlestore-<TeamName>
```

Use the above command to clone your repo from GitHub Classroom, and use `go1.17 install ./...` to fetch all dependencies.

You will also need to use your own or the TA implementation of Tapestry as part of your project. **If you choose to use the TA implementation, you and your partner will need to sign this form as soon as possible.**

2 Introduction

For your final project, you'll be implementing a simple distributed filesystem called PuddleStore. It's based on *OceanStore*, a project that was developed at UC Berkeley in the early 2000s [1].

> *OceanStore is a utility infrastructure designed to span the globe and provide continuous access to persistent information. Since this infrastructure is comprised of untrusted servers, data is protected through redundancy and cryptographic techniques. To improve performance, data is allowed to be cached anywhere, anytime. Additionally, monitoring of usage patterns allows adaptation to regional outages and denial of service attacks; monitoring also enhances performance through pro-active movement of data.*[2]

The specifications for PuddleStore are a bit more modest. We'd like you to implement the basic functionality for a distributed filesystem using some of the projects and labs you've already finished this semester as a basis for a few of the essential components. In addition to the basic functionalities, you are required to implement some additional features to receive full credit; see the Capstone & Extra Features section for more information.

3 A Quick Diversion: File Systems

We know what you're thinking. This isn't CS167. How am I supposed to know how to implement a file system? Don't worry — you're not expected to develop a sophisticated file system, and we will describe everything you need to know to about file systems in this document. In the following sections, we will suggest a way in which to implement the PuddleStore file system, but it is important to remember that you have the freedom to design your implementation any way you would like provided that it offers the basic functionality that is expected.

Usually, persistent storage utilities are backed by a disk. In this assignment, however, all data will reside in memory and will be distributed among the nodes of your system.

The two primitive file system objects that ought to be available to users of PuddleStore are **files** and **directories**. A file is a single collection of sequential bytes and directories provide a way to hierarchically organize files. Directories are really just special files that are interpreted to contain references to other files (and other directories, too).

3.1 Files & Directories

In order to actually implement a file system, we'll need to define a set of primitive objects to be used internally. An obvious abstraction is the **data block**, which simply represents a fixed length

[1] https://oceanstore.cs.berkeley.edu/

[2] https://oceanstore.cs.berkeley.edu/publications/papers/pdf/asplos00.pdf

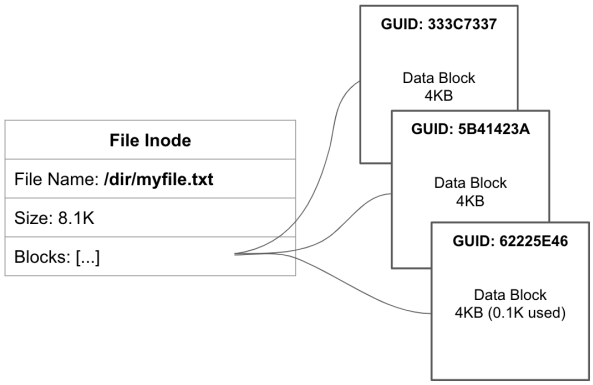

Figure 1: Inodes and data blocks

array of bytes. We can compose files of arbitrary length out of some number of data blocks. Some of the space available in the last data block will be wasted if the size of the file is not a multiple of the block size. The merits of storing files as collections of data blocks, rather than a single array of bytes, will become apparent when we discuss copy-on-write operations.

We also need an object to store a file's metadata. In file-system lingo, we call these objects **inodes**. An **inode** stores information such as the name and size of the file, as well as a map of data blocks associated with a file.

In a common linux system, directories are also represented by inode and have data blocks like files to represent its children. However, we have simplified the directory implementation this year, so that you don't need to worry about directories at all.

3.2 Reading & Writing

The simplest interface you can provide for reading from and writing to files will take a byte location at which to start and a buffer of bytes. When reading, the buffer gets filled with the bytes of the file, starting at the specified location. When writing, the buffer contains the data that should replace the current file data, starting at the specified location. If the user writes past the end of the file (or begins to write at a location beyond the end of the file), it is assumed that the length of the file ought to be extended. When new data blocks are added to a file, the inode for the file must be updated to reference those blocks.

Files are made up of many fixed length data blocks, so reading and writing with them isn't quite as simple as with an array of bytes. The block size, which will be provided as configurations, will determine how the data that makes up a file is distributed. Given the starting location and the number of bytes that you must read or write, you will need to figure out which blocks contain the data and where to begin and end within the first and last blocks.

4 PuddleStore Architecture

PuddleStore isn't just a file system; it's a *distributed* file system. The infrastructure is meant to allow data to reside anywhere on the network and be available everywhere. It also makes guarantees about the order in which updates appear at clients. The implementation details of these aspects of the system are described in this section.

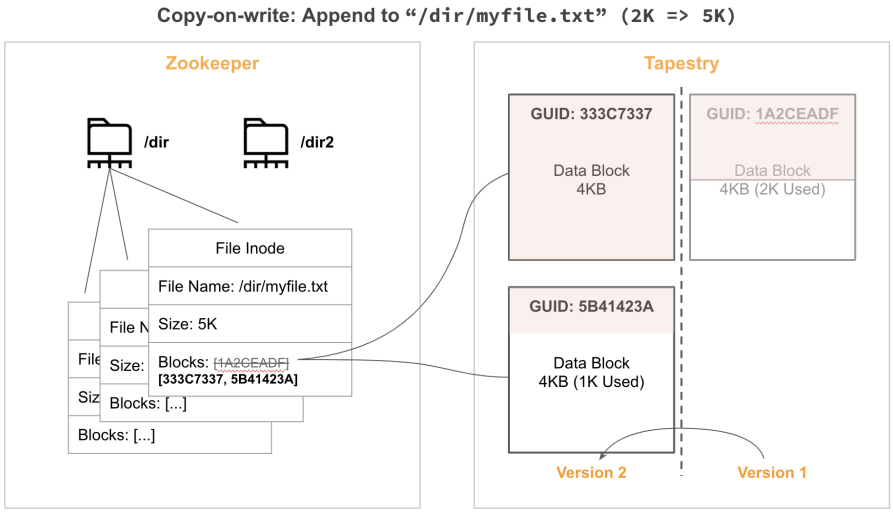

Figure 2: Copy-on-write: append 3K data to a file

4.1 DOLR

You will use a Distributed Object Location and Retrieval (DOLR) service, **Tapestry**, to store and replicate file data blocks. Each data block should be accessible through a unique ID at any Tapestry node. Your PuddleStore implementation will need to make choices about where to replicate objects, as Tapestry does not automatically replicate them for you.

4.2 File Hierarchy

We will use Zookeeper to simplify our file hierarchy implementation and maintain linearizability of file metadata. We put inodes (instead of the whole data blocks) in Zookeeper, because it is good at handling chunks of KBs efficiently.

When operating on files, your client will first consult Zookeeper for inode and block IDs, and then query Tapestry for the actual data blocks.

4.3 Updates

In order to have a useful file system, clients must be provided with a consistent view of files and directories. In a traditional, non-distributed file system, standard locking primitives are used to ensure that multiple threads do not leave the internal representation of a file in an inconsistent state. For PuddleStore, this is a more difficult problem because the objects stored by the DOLR are replicated to provide fault tolerance and therefore coordinating updates on them would require designating a master Tapestry node for each object and in addition to locking. Some distributed file-systems, like GFS, do this quite successfully but the sophistication required is beyond the scope of the basic specifications of this project. For this reason, PuddleStore uses a **copy-on-write** approach to maintain data consistency and a synchronous update system to update pointers to data.

Copy-on-write means that there is no modification of data blocks in the DOLR. When modifying a file, the client retrieves the data blocks it will update from the DOLR. It then makes changes to those blocks locally, then stores those blocks as new objects in the DOLR. As a result, the client is also responsible for modifying the file's inode whenever it makes any changes to it. This is because the file's data block mapping must be updated to point to the new data block IDs. (See Figure 2)

This approach maintains data consistency as it provides **atomicity** for file modifications. Without

copy-on-write, we could run into a situation where two clients' modifications to the same file both partially succeed. For example, imagine Client A and Client B both try to modify Block 1 and 2 of the same file at the same time. Since we are *not* using copy-on-write in this example, we directly modify the original data blocks and thus, should not change the inode of the file. Since our DOLR doesn't offer multi-object transactions, it is very possible that Client A writes to Block 1 after Client B, but Client B writes to Block 2 after Client A. The end result is a file where Block 1 has Client A's modifications but Block 2 has Client B's modifications. This will make both Client A and B very confused when they subsequently read from the file.

Thus, the key aspect of **copy-on-write** in our implementation is the **atomic** modification of the file's inode. Every time a file is updated, its contents will represent the exact data block mapping that the modifying client had.

To facilitate copy-on-write, each data block in the file-system should be assigned a unique ID, which we will call a **GUID (Globally Unique ID)**. Unfortunately, in a distributed system, there is no way to guarantee unique IDs without some global sequencer. The additional overhead to implement a global sequencer just to provide unique GUIDs does not add much value to this project so it is fine to just use some UUID library to generate probabilistic unique IDs.

In a file system it is also necessary to uniquely identify the files and directory themselves. A typical file system uses unique inode numbers to represent files and directories. I know what you're thinking, why not just use the file path as the unique ID for each file and directory? While it's true that two different files **cannot** have the same path in a file system, two different paths **can** point to the same file via file-system links. Luckily for us, PuddleStore is not sophisticated enough to support file-system links so using file paths as unique IDs for files and directories is sufficient.

4.4 The Membership Server

Just as the underlying Tapestry system allows some nodes to be added and removed on the fly, PuddleStore should be able to scale. The unified system should have a well defined mechanism for manipulating the membership of the DOLR network. Any systems that require coordination not provided by Tapestry should have a way to scale as well. Additionally, the API needs a mechanism for balancing client requests across the many servers that will be available.

The simplest way to handle this is to maintain a well-known membership server. It can facilitate scaling of the DOLR network by providing gateway servers for new Tapestry nodes to use to join. Clients can also query the membership server to get access to existing Tapestry nodes when performing operations. The membership server is not a focus of this project. You are only required to provide enough functionality to add nodes to the system and try to ensure that, on average, each Tapestry node is being used by some clients. A simple implementation might just return a random Tapestry node at the request of a client and provide a way to add and remove nodes from its member lists.

Zookeeper provides out-of-box membership service. Checkout this link.

> ... Another function directly provided by ZooKeeper is group membership. The group is represented by a node. Members of the group create ephemeral nodes under the group node. Nodes of the members that fail abnormally will be removed automatically when ZooKeeper detects the failure.

4.5 File Locking

As we have described it, PuddleStore handles conflicting updates inadequately. For instance, if two clients simultaneously attempt to modify different byte ranges of the same file, it's likely that the final version of the file will only contain the changes of one client. To prevent this from

happening, you need a file locking protocol that will ensure that each file is updated by a single client at a time.

For this, you can adapt the distributed lock that you implemented in the Zookeeper lab. Since we are not providing a stencil file for this, you can do this however you'd like. Keep in mind that your lock implementation likely won't work without modifications. One important issue to consider here is the existence of other znodes in Zookeeper. For example, if you want to lock the directory `\a`, which has a file `\a\b`, calling `Children` on `\a` will return not only the ephemeral lock znodes created for locking, but also `\b`.

Note: You are not allowed to use the built-in locking functions in go-zookeeper library; if you do, the autograder will flag your submission.

5 The Assignment

5.1 Requirements

You have two tasks for this assignment:

1. **Setup the PuddleStore cluster**: Fill out the structs and methods in `cluster.go` and `tapestry.go`. This simply entails creating a Tapestry cluster and 'registering' each node to Zookeeper. You will also need to use the `Tapestry` wrapper struct in `tapestry.go`.

   You can assume that the centralized Zookeeper server in our filesystem is 100% reliable for the scope of this project. Thus, all PuddleStore clients only need to connect to the address provided in the single cluster's `config.ZkAddr`.

2. **Implement the client interface**: We provide an interface in `client.go` that PuddleStore clients must implement (Open/Close/Read/Write/Mkdir/Remove/List). Your client will operate on the provided Zookeeper server and tapestry nodes and provide the complete functionality of a file system.

   The Client interface has documents on the expected behavior of each operation. **Make sure you follow the document.**

   Since one of the main goals of this assignment is to implement an API spec on top of fully-implemented components (including the Tapestry project), Puddlestore clients should **not be able to directly access tapestry nodes** in your implementation. Rather, they should consult tapestry clients and make appropriate API calls on them.

In general, all of the functions, structs, and interfaces you will be implementing are decorated with copious comments. You should adhere to the specs in these comments as our grading relies on these specs. Also, take a look at `config.go` for necessary cluster configuration.

5.2 Implementing the File Hierarchy

We've mentioned the necessity of inodes in file systems and using file paths as a way to identify inodes, but we haven't mentioned where to actually store this information. You shouldn't store this information in Tapestry since we require linearizable operations of file metadata. Linearizability of file metadata is crucial for maintaining consistency across our distributed file system. You can imagine how disastrous it would be if say, the deletion of a file was eventually consistent.

You previously worked on a system that accomplishes just this: Raft. However, we're not going to use Raft for this project. Instead, we'll be using Zookeeper to implement the file system hierarchy, store inodes, and maintain cluster membership. The decision to use Zookeeper is due to the fact that it provides a file-system API and an easy way to keep track of cluster membership changes.

5.3 Encoding and Decoding Messages

Since you will be storing inode information in Zookeeper, you will need to write functions that can encode your inode struct into bytes and decode your inode struct from bytes. This is because Zookeeper is a distributed store that maps file paths to byte data.

To do this, we recommend you use `encodeMsgPack` and `decodeMsgPack` in `utils.go`. These are also used in Raft project, so you can see how we used them to encode and decode LogEntries to store into Bolt. Note that these functions will be able to encode/decode only the public fields, so make sure that you capitalize first letters of all the field names of your inode struct.

5.4 Capstone & Extra Features

Every group is required to implement at least one of these features. Students taking CS1380 as a capstone or groups of 3 are required to implement two of these features. You are welcome to implement more for extra credit.

5.4.1 Read-Write locks & Pre-fetching

Assuming a read-heavy access pattern, there are several approaches to optimize performance:

1. Separate read-write locks to allow multiple clients to read.
    - Write Lock: For each file, at most one client can acquire a write lock. With a write lock, the client can freely, read/write to a file.
    - Read Lock: For each file, many clients can simultaneously hold a read lock. Each client with a read lock can simultaneously read the file.
    - If a client holds a write lock, no one can hold a read lock.
    - The Zookeeper locking recipe provides detailed instructions on how to implement read and write locks.
2. Monitor the behavior of clients to determine the average number of blocks that a client reads from a file. Based on this, you can pre-fetch these blocks and cache the locally at the client. For example, your implementation notices that on average clients read 4 blocks from a file. When a client reads a block, you can pre-fetch the next 3 blocks.

5.4.2 Access Control List

The default implementation provides no security primitives. One option is to extend Puddlestore with features for access control list. This will require additional meta-data and additional calls.

In particular, you will need to add the following calls:

- `Chmod(file, ACL)` which sets the permission on the file. You only need to support two permission levels: Owner (i.e., UserID of the client which created the file) and Everyone (i.e., permissions for all other `UserID`s). The ACL options are:
    1. "44" – owner can read, everyone can read.
    2. "55" – owner can read/write, everyone can read/write.
    3. "54" – owner can read/write, everyone can read
    4. "45" – owner can read, everyone can read/write.
- `Login(UserID)` which specifies the userID for a client.

5.4.3 Web Client

Web application to control and interface with PuddleStore (using gRPC). (Refer to this repo for an example on node.js and grpc-web.). This web application should support similar operations and commands as the PuddleStore client.

5.4.4 Tapestry Hotspot Caching and Reliability

There are many techniques you may use to improve the Tapestry's performance and reliability. Because objects may be replicated anywhere in a Tapestry network, it makes sense to move copies of an object to the nodes that request that object often (this is "hotspot caching"). The speed of surrogate node lookups can also be increased by caching object location information at the nodes along the publishing path from the object replica to the surrogate node. To protect against node failures, you can salt the hash so that the information is sent to multiple nodes. Finally, objects may be re-replicated as Tapestry nodes leave the network or fail, ensuring that the system remains reliable over longer periods of time. If you do this, you may also want to implement erasure codes.

Note: You should only implement this if you did not implement hotspot caching for Tapestry already.

6 Testing

You are expected to thoroughly test your code. For this project, your testing will be a large portion of your grade. You should provide exhaustive tests that demonstrate edge cases and specific behaviour within your PuddleStore implementation. As with previous projects, you might find it useful to check your test coverage by using Go's coverage tool.

You are expected to reach 80% test coverage.

Warning: When you run multiple tests while running a single instance of Zookeeper, previous tests may influence later tests since non-emphemeral znodes will persist throughout different tests. You may want to implement a util function that cleans up everything so that it can be called at the beginning or end of every test.

7 Style

You should use Go's formatting tool `gofmt` to format your code before handing in.

You can format your code by running:

```
gofmt -w=true */*.go
```

This will overwrite your code with a formatted version of it for all go files in the current directory. You will be graded for this!

8 Handing in

You need to write a README documenting any bugs in your code, any extra features you added, and anything else you think the TAs should know about your project. Document the test cases you created and briefly describe the scenarios you covered.

When you are done, submit your project to Gradescope. Each group should submit only one copy. Make sure to add your partner name in the submission page using "add group member".

Feedback

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS138 document by filling out the anonymous feedback form.