Project 4: Puddlestore

Due: 11:59 PM, May 6, 2019

Contents

1 Introduction

For the final CS1380 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 you've already finished this semester as a basis for a few of the essential components. You will have a high degree of freedom in this assignment to design your own APIs and system infrastructure. In order to receive full credit on this assignment you will need to research and implement an additional component of your choosing.

2 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.

Narrowly defined, a file system is the storage utility, usually provided by the operating system, where applications can read and write persistent data (information that is still available after the program has ended). 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 (at least from the perspective of applications) 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).

2.1 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 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

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

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

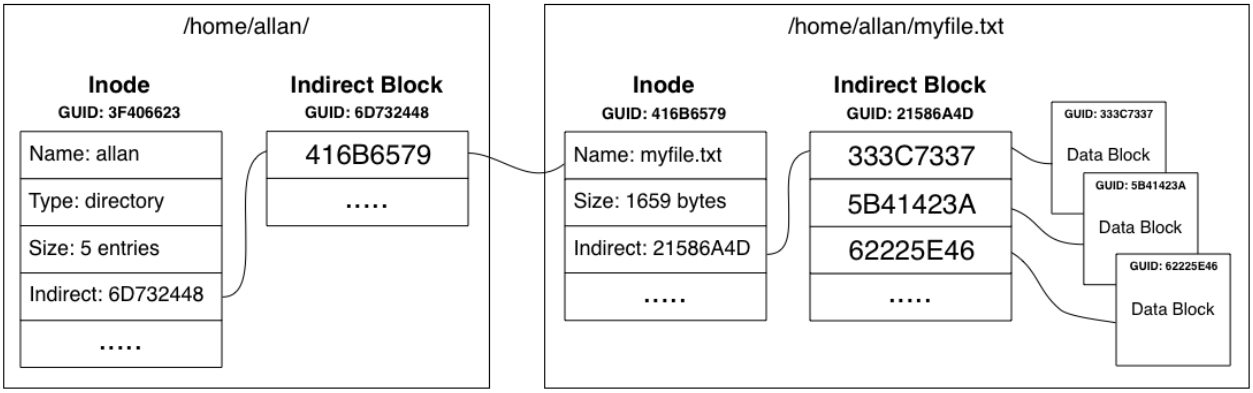

Figure 1: Inodes, indirect blocks, and data blocks

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.

Somewhere, we need to store references to an ordered list of data blocks that make up a file. In common filesystem parlance, this is called an **indirect block**. In a standard disk-based file system, data blocks are accessible via integer indexes, and indirect blocks are just data blocks that have been allocated to hold a list of integers that refer to the blocks that make up a file. In our system, we'll refer to data blocks and other fundamental file system objects with GUIDs (Globally Unique Identifiers), which are integers or strings that refer to a certain object that may be located on any node in the distributed system. An easy way to obtain a GUID is to take the hash value of some human readable string that identifies the object.

Finally, a third type of object called an **inode** maintains the metadata associated with a file. Information stored in an inode could include the name and size of the file, as well as a pointer to the indirect block.

Directories are represented internally in a similar way to files. Each directory also has its own inode object, but this inode is marked as representing a directory. It also has an indirect block like a file inode, but in this case the indirect block is interpreted as containing pointers to the files and other directories linked to by the directory.

2.2 Programming Interface

Applications that would like to store data in a file system need a way to access it. Basic operations that must be available to applications include the following:

- `open`: obtains a reference to a particular file
- `read & write`: for examining and changing the data contained in a file
- `create`: creates a new file or directory within an existing directory
- `list`: obtains a list of the contents of a particular directory

- `remove`: deletes an entry from a directory

There are many more operations that would be useful to have, such as *appending* and *locking*, but you are not required to implement them. You should think about how you want to interact with the file system; depending on how you think or like to write your programs, it might make sense to have an object-oriented interface that returns references to files and directories as objects with relevant methods, or instead a POSIX-like interface that relies on file descriptors (integers which identify files in the kernel). Be creative with your interface. The methods you make available do not have to be named `open`, `create`, or `list`. Create an interface that makes sense to you.

2.3 Naming & Lookup

All file systems allow users to identify files and directories with a string (called a path) that adheres to some naming conventions. In a Unix based operating system, one might construct the path to a file like `/home/allan/myfile`, whereas on Windows an equivalent path would look like `C:\Documents and Settings\allan\myfile.txt`. Both paths start at the root of the filesystem (the root of the `C:` drive in the Windows example), and from left to right they list the directories that should be traversed to get to the specified file. Subdirectories are separated by a delimiter character (for Unix it is '`/`' and Windows it is '`\`'). To simplify naming, Puddlestore will use Unix-style paths.

So, how should we find the file corresponding to some path when implementing `open`? Given the kinds of objects we have to work with, it should be obvious that the first step is to obtain the root inode of the file system, from which we can get the indirect block. We then iterate over the contents of the indirect block, obtaining each inode referenced and checking whether its name matches the first directory listed in the path. Once the first directory is found, the same search is performed with its indirect block. This process is repeated until the last file or directory in the path is found. Notice that it is necessary to have a well-known way to obtain the root inode, as all file lookups begin with it. It is imperative to perform lookup iteratively. This can be for several reasons such as checking file system permissions and ensuring that the path to the file is valid (i.e. no directory has been deleted in the path).

2.4 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 blocks are added to a file, the indirect block 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 you choose (4096 bytes is a pretty standard block size, but you'll want to make them smaller for testing) 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.

3 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.

3.1 DOLR

You will use a Distributed Object Location and Retrieval (DOLR) service, Tapestry, to store and replicate inodes, indirect blocks, and data blocks. Internal file system objects should be accessible through their GUIDs 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.

3.2 Updates

In order to have a useful filesystem, 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 filesystems, 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.

To facilitate copy-on-write, file system objects may be given a constant **Active GUID** (AGUID) which maps to a **Version GUID** (VGUID). The VGUID references the "current" version of an object which can be obtained through the DOLR. An **inner-ring** of servers replicates the AGUID to VGUID mappings (this is where Raft comes in). By using Raft as the synchronous update mechanism maintaining these mappings, we ensure that modifications to the filesystem occur in a causal and total order from the perspective of client applications.

At the very least, each file and directory inode must be given an AGUID. To modify file metadata (such as the name of a file), the current version of the inode is obtained and copied. The modifications are performed on the copied data and then the new inode is replicated via Tapestry. Finally, the AGUID for the file is updated synchronously to point to the new VGUID. Mutations on file data are handled similarly. The current version of a file's inode, indirect block, and relevant data blocks are obtained first. The data blocks that require modification are copied and mutated (and new data blocks are created when writing past the end of a file). A copy of the indirect block is made and modified to point to the new data blocks. To finish it off, the inode is copied and modified to point to the new indirect block and the AGUID for the file is updated to point to the VGUID of the new inode. Note that the indirect blocks of a directory must contain the AGUIDs of the files and subdirectories that it contains, not the VGUIDs.

If you choose, you may assign every filesystem object an AGUID, instead of just inodes. With this configuration, it won't be necessary to copy inodes and indirect blocks as often, but the inner-ring will be required to handle more updates. The decision of whether to have AGUIDs for every object

or not is a design choice that you must make.

3.3 The Membership Server

Just as the underlying Tapestry and Raft systems allow 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 and the inner-ring. Any systems that require coordination not provided by Tapestry or Raft nodes 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. (Unless you are implementing membership changes for Raft, your membership server only needs to support helping the Raft nodes initially join.)

The simplest way to handle this is to maintain a well-known membership server. It can facilitate scaling of the DOLR network and inner-ring by providing gateway servers for new Tapestry or Raft nodes to use to join. Clients can also query the membership server to get access to existing Tapestry and Raft 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 and Raft node is being used by some clients. A simple implementation might just return a random Tapestry node or Raft node at the request of a client and provide a way to add and remove nodes from its member lists.

4 The Assignment

To receive a **B grade** on this project, you need to implement PuddleStore as described in this handout and write an interactive test program that exercises all of the functionality of the system. An intuitive test program is important because you will be demoing your project to a TA during an interactive grading session. You also need to turn in a design document that describes your plans for completing this project.

4.1 Getting an A

PuddleStore lacks many features that would make it truly useful. To get full credit on this assignment, you need to go beyond the basic specification and implement additional functionality. **Your extra functionality should solve a distributed systems problem.** You are welcome to improve aspects of the system that do not relate directly to its distributed nature for usability purposes, but you cannot obtain an A unless you implement a feature worthy of an A. Below are a few suggested areas of improvement, but feel free to propose your own. No matter what you choose, you should consult the TAs (at the very least, via your design document) about implementation details and the scope of your project. We want you to try something doable and interesting.

Note: Three person groups are required to implement one A-level feature to get a B and two A-level features to get full credit in addition to implementing the B-level Puddlestore as described above.

4.1.1 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. Optionally, you could expose your file locking procedures to provide entry consistency that is compatible read operations as well. Basic file locking could be very easy to implement, so your solution will need to incorporate some level of fault tolerance.

4.1.2 Raft Membership Changes

For your Raft implementation, you assumed that the size of the cluster was fixed, and everyone knew everyone else's addresses from the start. This prevents you from dynamically adding new members when others permanently fail and need to be replaced, or when you want to increase the number of nodes permitted to fail before consensus cannot be reached. Raft allows for membership changes by implementing *joint consensus*, which allows for two configurations (the old and the new) to be live during the transitional period, and requires that a majority of nodes in both configurations agree before a log entry can be committed. The details for implementing this feature can be found in the Raft paper.

4.1.3 Raft Log Compaction

To prevent Raft's log from growing inordinately large, it is necessary to occasionally compact it to something smaller. Raft does this through *snapshotting*, the simplest approach to compaction. In snapshotting, the entire state of the state machine and all other relevant system state is written out to stable storage, and the log up to that point is discarded. The Raft paper discusses how to implement this, and also alludes to two other techniques: log cleaning and log-structured merge trees. If you decide to implement log compaction, but want to go with one of these two methods instead of snapshotting, please let us know in your design document.

4.1.4 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. You can implement several of these five features together to receive an A on the project. For the design document, please propose three of these features you would like to implement. If you choose three more challenging features, the TA reviewing your design document may inform you that you only have to complete two of them.

4.2 Zookeeper

Apache Zookeeper is a centralized server for maintaining distributed configuration service, sychronization service, and naming registry for large distributed systems. Instead of using a membership server, Puddlestore clients can use Zookeeper to find Raft or Tapestry nodes to interact with the network, and Raft and Tapestry nodes can connect to the clusters using Zookeeper. If you do this, you will need to propose two additional features from the Tapestry Hotspot Caching and Reliability section above to implement as well.

4.3 Design Document

We want you to start thinking about your project design early. This project will be involved, and you have about 3 weeks to complete it for a reason. In your design document you should answer the following questions.

- Which features would you like to implement to receive an A? At a high level, what is required to implement these features? See 4.1 for a description of this part of the project.
- How do you plan to structure the API that you will expose to client applications?
- What sort of test application will you write to exercise your API?
- What is your strategy for testing aspects of the system that may not be exercised fully by an interactive application? The internal guarantees that your system makes may not be easy to test using only your API.

This document counts for **5% of your project grade**.

4.4 GRPC and Protobuf

Before starting to implement the puddlestore please think about what RPC calls would be required in order for different nodes and clients to communicate with membership server and with each other. You can find more about how to writing and generating Protobuf file using the GRPC lab guide.

4.5 Collaboration

You are encouraged to collaborate more than usual on this assignment. You may discuss the implementation details of your A or B level features with other students (although you cannot share the code itself). Also, groups of students may agree to implement the same API and share test programs with each other as long as each turns in their own, unique test program.

5 Testing

Since you will be implementing your own APIs and demoing your PuddleStore to use (see Section 7: Grading), you will need 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[3].

Additionally, you should ensure your interactive test program provides all of the functionality required for the assignment, as well as the ability to demonstrate your A level feature. The TAs will be using this to test your program in interactive grading. (See the section on "Grading".) You can also refer to the testing guide for tips on how to write test cases and test the distributed systems in general. [4]

6 Code Exchange

Remember, if you write code on a department machine, you must use go1.11 instead of just go (ie go1.11 install or go1.11 test)

There is no support code for this assignment! You should use your implementations of Tapestry and Raft as part of the project, however. Additionally, you should complete the project in Go.

If you are seriously concerned about the quality of your Tapestry or Raft assignments, please contact the TAs, and we can provide you with the TA implementations. You and your partner will have to sign an agreement that you will not share the code with anyone.

7 Grading

You and your partner will be graded interactively by the TAs. This means that you will need to provide and be able to demonstrate tests that make us confident in your implementation and show off all of the features that you implement. Your tests and examples will be the majority of your PuddleStore grade. You should be able to talk about the edge cases that your tests cover, and convince us that your testing is exhaustive. Your tests should also be specific: setting up a PuddleStore cluster, running through some common PuddleStore operations, etc., is not enough.

You should implement an interactive command-line interface, so that we can play around with your PuddleStore implementation during the demo, and have you demonstrate things we want to see, but you may not have tested out. (Having a CLI is also useful for you own testing purposes during development.)

Aside from tests, you may also find it useful to provide additional materials to us, such as graphs or tables of data you collect from running your PuddleStore implementation. For example, if you implement hotspot caching, you may want to show us the performance improvements.

8 Handing in

One partner from your group should handin a PDF of your design document by **16:00 PM, Apr 18, 2018** on Gradescope.

[3] `http://blog.golang.org/cover`

[4] `https://docs.google.com/a/brown.edu/document/d/12Siqqv03favnfjowf7g5TRP0bHkw9a-2NSc-1AHXcQA`

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. Once you have completed your README and project, you should hand in your `puddlestore` by running

```
/course/cs1380/bin/cs1380_handin puddlestore
```

to deliver us a copy of your code.

Please look for an email regarding interactive grading closer to the assignment due date.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS1380 document by filling out the anonymous feedback form:

`http://cs.brown.edu/courses/cs138/s19/feedback.html`.