416 Distributed Systems: Assignment 2

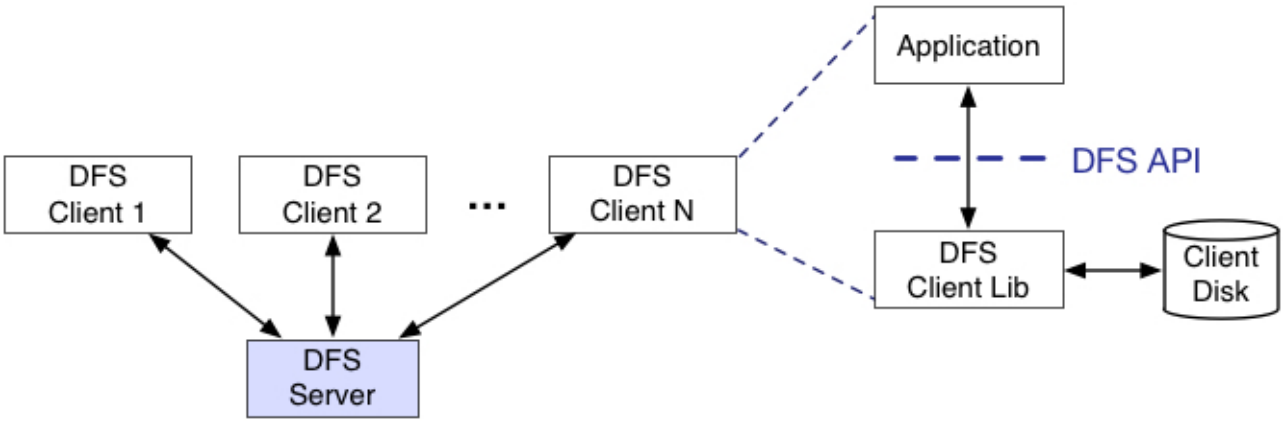
Due: Jan 29th at 11:59PM

Winter 2018

In this assignment you will learn about distributed file systems by building one. Your DFS will support a narrow API through a client-side library and your DFS design will be based around two key features: client-side storage and client-side caching. That is, in your DFS the clients will store all of the files. And, your system will implement client-side caching to improve application read/write performance. Finally, your DFS will have to be robust to certain failures.

System overview

There are two kinds of nodes in the system: clients and a single server. The clients are connected in a star topology to the server: none of the clients interact directly with each other and instead communicate indirectly through the central server. Each client is composed of a DFS client library and an application that imports that library and uses it to interact with the DFS using the DFS API. Each client also has access to local persistent disk storage.



In DFS the server is used for coordination between clients and does not store any application data. However, the server may store metadata, such as which client has which files, which client has opened which files, etc. All application data must live on the clients, specifically on their local disks (to survive client failures). Although clients communicate indirectly through the server, a client should make no assumptions about other clients in the system -- about other clients' identities, how many clients there are at any point in time, which client stores what files, etc. Such client meta-data should, if necessary, be stored at the server.

Your DFS system must provide serializable file access semantics and gracefully handle (1) joining clients, (2) failing clients, and (3) clients that access file contents while they are disconnected from the server.

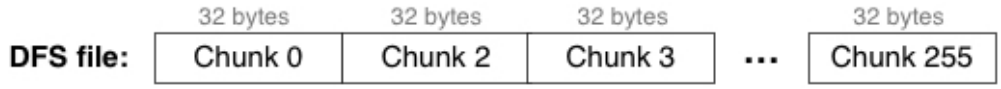
DFS API

A client application uses the dfslib library to interact with DFS. This library exposes an interface that is detailed in the dfslib.go file. You are **not** allowed to change this client interface. However, you have design freedom to implement this interface however you want. Your library does not need to be thread safe. That is, you can assume that a single application thread is interacting with your library.

You can use a stub implementation of the library API and a client application that uses it as starters for your system:

- dfslib.go (dfslib)
- app.go (application)

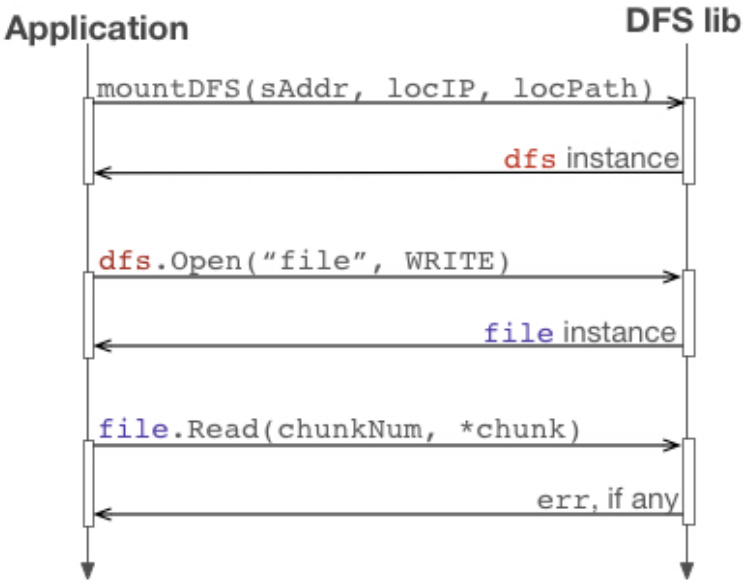
DFS files are accessed in units called **chunks**. Each chunk is a fixed size byte array of 32 bytes. In the API a chunk is identified by a `chunkNum` of type `uint8`, so a file could contain at a maximum 256 chunks:



DFS files must have alpha-numeric names that are 1-16 characters long. DFS filenames must use lower-case letters (this prevents local FS differences from cropping up in DFS; e.g., HFS+ is case-insensitive, while ext3 is case sensitive). Filenames should be validated during `Open`; return a `BadFilenameError` on violation. DFS files cannot be deleted (once opened), exist in a single shared namespace, and are readable/writable by any client (no access control). All files must be stored as local OS files at `localPath` passed to `MountDFS`. A file with DFS name `foo` should be stored as `foo.dfs` on disk. A file could be opened by clients in three different modes:

- (READ) mode for reading. In READ mode, the application observes the latest file contents using `Read` calls. In this mode `Write` calls return an error.
- (WRITE) mode for reading/writing. In WRITE mode, the application observes the latest file contents using `Read` and is also able to make modifications using `Write` calls.
- (DREAD) mode for reading while being potentially disconnected (from the server). In DREAD mode, the application observes potentially stale file contents using `Read` calls. In this mode `Write` calls return an error.

Here is an illustration of an example exchange between an application and the DFS lib:



File access semantics

Your system must provide serializability to all READ and WRITE mode operations on the same file . That is, regardless of the order in which DFS API calls were issued by the set of all the applications in the system, the order of these completed operations for the same file should be observable to all applications as (1) a single serial order, and (2) this order should be identical across all applications. In other words, to applications your DFS system must behave as if it was a local file system. However, you do not need to provide serializability *across* files and you do not need to provide these semantics for DREAD operations.

Chunks. Read and Write operations operate in discrete chunk units. Chunks have default values of an initialized byte array (with byte values of 0). In other words, DFS files are constant size (of 256 chunks) and are initialized to a default value. This means that it is legal to read a chunk number 8 before chunk number 8 was written; such a read should return an empty byte array.

Durability. Once a write of a chunk returns to the application, the written chunk must be guaranteed to be durable on disk at the client that issued the write and (optionally) at other clients in the system.

Disconnected operation. Operating on a file opened in READ or WRITE modes while disconnected should return an error (`DisconnectedError`), and all future operations on **this** file should also fail (even if the client re-connected). Operations on other files may continue to succeed (e.g., if the disconnection was transitory).

A client that has opened the file in READ or WRITE modes and has accessed chunks c_1,...,c_N should be able to later access these same chunks in DREAD mode (while disconnected). Furthermore, the contents of a chunk c_i that this client accesses in DREAD mode should be no more stale than the last read/write that the client observed/performed on c_i.

A client that has successfully opened the file must have fetched the latest version of the file to the client's local disk (before returning from Open). That is, once a client has opened a file, a version of the file must be available to the client during disconnected operation in DREAD mode

An application may want to open a file in DREAD mode while it is disconnected. This requires that `MountDFS` call succeeds even if the client is disconnected.

Concurrency control. The system must allow a **single writer** and **multiple readers** to co-exist. That is, if two applications attempt to open the same file for writing, only one of them should succeed. However, there could be any number of applications that open a file for reading.

In practice, the above semantics imply the following properties (among others):

- A `Read` call in READ or WRITE modes always returns the latest version of the chunk (i.e., the version that was generated by the last `Write` call that returned).
- A `Write` call in WRITE mode does not return to the application until the system can guarantee that (1) any future read at the same chunk position (from any client) will return the written chunk content, or (2) any future read at the same chunk position (from any client) will return an error because this (latest) chunk cannot be accessed due to client failures.
- Once the client opens a file, the client must become a replica for the file. Furthermore, if the client opened the file for the first time, but no other client is online that has the file contents, then the open should fail (with the `FileUnavailableError`). That is, the client cannot open a file unless the client can completely download some version of the file to its local storage. More precisely, the client must attempt to download the latest version of each file chunk. If it cannot do that, then it must grab the most up-to-date version of each chunk that it can find available. However, the client cannot grab the trivial version of the file (the version that has been opened/exists but has not been written --- each chunk has a default value and initial version); in this case open should a `FileUnavailableError`.
- A file that was successfully opened in READ or WRITE modes must later successfully open in DREAD mode (since some version of the file is guaranteed to be available to the client locally offline). Note, however, that the reverse does not have to be true: a file open in DREAD mode could succeed by fetching the file (if there is connectivity when open is called), without relying on a previously fetched version of the file. However, the exact semantics of open in DREAD mode are up to you to finalize, as long as you are consistent.

Handling failures

Your system must gracefully handle fail-stop client failures. That is, in your system clients may fail by halting. When this happens, the server should shield other clients from the failure and the rest of the system should (eventually) resume normal operation with minimal disruption to other clients. For example, if a client A is writing a file, and client B fails, then A should not observe B's failure (e.g., A's `Write` call should not fail).

If client A has file `f` open in WRITE mode and fails, then another client B should eventually be able to open `f` in WRITE mode assuming that B has a copy of the file or a copy exists on some other connected client.

Note that the mode of an opened file cannot change during failures. You must provide the semantics for the mode the file was opened with (e.g., a `Read` cannot return stale data if the file was opened in READ or WRITE modes).

Unobservable transitory disconnections. A client could disconnect and then reconnect without the application making any DFS library calls. In this case the transitory disconnection is unobservable to the application and the library should not reveal the disconnection to the application in calls after the re-connection. There is one major exception to this rule:

- Consider a client A that has a file F open for writing. Client A disconnects and does not write to F while it is disconnected. If another connected client B decides to open F while A is disconnected, the server may decide that A has been disconnected for a long enough period that it can time-out A and revoke its writer status. In this case, when A reconnects and makes a `Write` call, the library should return `WriteModeTimeoutError`.

Assumptions you can make

- The server is always online, does not fail, and does not misbehave.
- A timeout of 2s can be used to detect failed clients.
- A client that has failed will not re-connect to the server for at least 2s.
- A client does not fail until after the `MountDFS` call has returned.
- A maximum of 16 clients will ever be connected to the server.
- Clients are not behind a NAT or Firewall that makes them unreachable from the server. In particular, the server can open a UDP/TCP connection to the client.
- You are free to manage `localPath` however you want (e.g., you can create additional files, directories, etc).

Assumptions you cannot make

- There is a known ordering on when clients fail or connect to the server.
- There is a known number of clients that will join the system or that will fail.
- A client that has failed will later re-join the system.
- Clients and the server have synchronized clocks (e.g., running on the same physical host).
- Reliable network (e.g., if you use UDP for communication, then expect loss and reordering)
- You are running on a specific type or version of an OS.

Solution sketch

The above specification describes a system that can be implemented in many different ways. For example, you can design the communication protocol between the client library and server to use RPC/HTTP/TCP/etc. And, different solutions will make different trade-offs between performance/availability/consistency while satisfying the above specification. This section sketches out an implementation direction that we recommend. You may follow this sketch, or deviate from it. That is your choice.

(Show solution sketch)

Implementation requirements

- The client code must be runnable on CS ugrad machines and be compatible with Go version 1.9.2.
- You must support the API given out in the initial code.
- The `server.go` file should live at the top level of your repository in the *main* package.
- The dfslib implementation should live inside the `dfslib/` directory at the top level of your repository (it should be possible to run `go run app.go` at the top level).
- Your solution can only use standard library Go packages.

- Your solution code must be Gofmt'd using gofmt.

Solution spec

Write a go program called `server.go` and a Go library called `dfslib.go` that behave according to the description above.

Server's command line usage:

```
go run server.go [client-incoming ip:port]
```

- [client-incoming ip:port] : the IP:port address that clients instances of dfslib use to connect to the server

Starter code

You can use a stub implementation of the library API and a client application that uses it as starters for your system:

- dfslib.go (dfslib)
- app.go (application)

Rough grading scheme

- 60% : Connected operation
  - 20% : One client; example scenarios:
    - Client opens/reads/writes/closes a file, quits.
    - Another client starts and uses a different local path; check that prior client's files exist globally, but file chunks are unobservable
  - 20% : One reader/writer client and one writer client; example scenarios:
    - Both clients attempt to open same file for writing
    - Client A creates file, client B checks file with `GlobalFileExists`
    - Client A writes file, client B reads same file and observes A's write
    - Client A writes file, client B writes same file, client A observers B's write
  - 20% : Multiple reader clients and one writer client; example scenarios:
    - Client A writes file, client B,C,D checks file with GlobalFileExists
    - Client A writes file, client B,C,D reads same file and observe A's write
    - Client A writes file, client B writes same file, clients C,D observe A-B writes in same order
- 40% : Disconnected operation
  - 20% : One client; example scenarios:
    - Client writes file(s), disconnects, and can use DREAD and `LocalFileExists` while disconnected
    - Single client unobservable transitory disconnections
  - 10% : One reader/writer client and one writer client; example scenarios:
    - Client A opens file F for writing, disconnects. Client B connects and opens F for writing, succeeds.
    - Client A writes/closes file, disconnects. Client B connects, writes file. Client A re-connects, reads, observes B changes.
  - 10% : Multiple reader clients and one writer client; example scenario:
    - Client A writes file, client B writes same file, clients C,D open file. A,B disconnect. Check C,D have identical file copies.

Make sure to follow the course collaboration policy and refer to the assignments instructions that detail how to submit your solution.

Last updated: January 21, 2018