Distributed Systems

Group Project 2: Distributed File System

Team monadfs

Pure functional distributed file system build on Haskell.

Team members:

- Mihail Kuskov m.kuskov@innopolis.university
- Alfiya Mussabekova a.mussabekova@innopolis.university
- Nikita Aleshenko n.aleschenko@innopolis.university

Link to Github repository

Contents:

Description of the task

According to project description, the task is to implement a simple *Distributed File System*, which will be able to support basic operations like file reading, deleting, writing, creating and etc. The main components of DFS are: *Name server*, *Storage servers*, *Client*. Clients access storage servers in order to read and write files. Storage servers must respond to certain commands from the naming server.

The goal of the assignment

1. Understand the roles of namenode, storages and client, distribute functionallity
2. Go deep into haskell language libraries
3. Write compiled and working web server
4. Deploy servers on AWS using docker

Prerequisites

This project relies on the Haskell Stack tool.

It is recommended to get Stack with batteries included by installing Haskell Platform.

Build

To build this project simply run

```
stack build
```

This will install all dependencies, including a proper version of GHC

Run

This app consist of multiple executable. You can run each one independently.

To start the client, run the following command:

```
stack exec monadfs-client
```

To start the name server, run the following command:

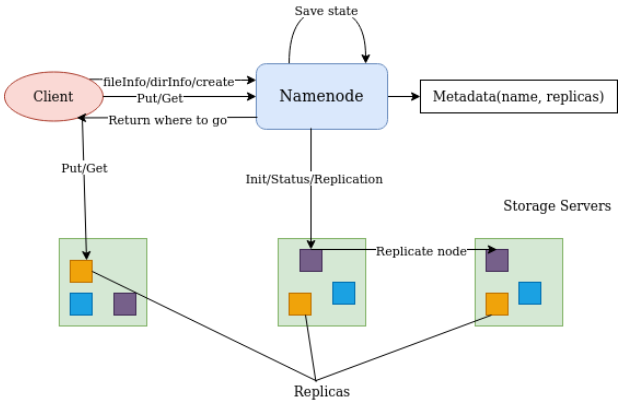
```
stack exec monadfs-name-server
```

To start the storage server, run the following command:

```
stack exec monadfs-storage-server
```

Implementation details

Project has the following structure:



Pipeline

We have one *namenode* which keeps metadata and controls storage services. In *docker-compose.yml* we can define the number of *storage services*, which is 3 by default. *Storage server* is the data store which provides *client* with access to data files. Whenever a *client* executes some command, it connects to *name server*, which either executes this command (*dirInfo*, *fileInfo*, *create*) or finds where needed data resides and returns address to *client*, which in its turn connects to *storage server* and executes the command (*get*, *put*).

API for *name server*.

```
type NameServerAPI = "init" :> Get '[JSON] SystemStatus
                   :<|> "file" :> FileAPI
                   :<|> "dir"  :> DirAPI

type FileAPI = "create" :> ReqBody '[JSON] FilePath   :> Post '[JSON] (FileStatus ())
          :<|> "read"   :> ReqBody '[JSON] FilePath   :> Post '[JSON] (FileStatus ServerAddr)
          :<|> "write"  :> ReqBody '[JSON] NewFile    :> Post '[JSON] (FileStatus [ServerAddr])
          :<|> "delete" :> ReqBody '[JSON] FilePath   :> Post '[JSON] (FileStatus ())
          :<|> "info"   :> QueryParam' '[Required] "file" FilePath
               :> Get '[JSON] (FileStatus FileInfo)
          :<|> "copy"   :> ReqBody '[JSON] SourceDest :> Post '[JSON] (FileStatus ())
          :<|> "move"   :> ReqBody '[JSON] SourceDest :> Post '[JSON] (FileStatus ())

type DirAPI = "create" :> ReqBody '[JSON] DirPath  :> Post '[JSON] (DirStatus ())
         :<|> "delete" :> ReqBody '[JSON] DirPath  :> Post '[JSON] (DirStatus ())
         :<|> "info"   :> QueryParam' '[Required] "dir" DirPath :> Get '[JSON] (DirStatus DirInfo)
         :<|> "exists" :> QueryParam' '[Required] "dir" DirPath :> Get '[JSON] (DirStatus ())
```

API for *storage server*.

```
type StorageServerAPI =
  "init" :> Get '[JSON] StorageServerStatus
    :<|> "tree" :> Get '[JSON] StorageTree
    :<|> "status" :> Get '[JSON] StorageServerStatus
    :<|> "file" :> FileAPI
    :<|> "dir" :> DirAPI

type FileAPI =
  "create" :> ReqBody '[JSON] FilePath :> Post '[JSON] (FileStatus ())
    :<|> "read" :> Raw
    :<|> "write" :> MultipartForm Tmp (MultipartData Tmp) :> Post '[JSON] (FileStatus ())
    :<|> "delete" :> ReqBody '[JSON] FilePath :> Post '[JSON] (FileStatus ())
    :<|> "copy" :> ReqBody '[JSON] SourceDest :> Post '[JSON] (FileStatus ())
    :<|> "move" :> ReqBody '[JSON] SourceDest :> Post '[JSON] (FileStatus ())
    :<|> "load" :> ReqBody '[JSON] LoadFile :> Post '[JSON] (FileStatus ())

type DirAPI =
  "create" :> ReqBody '[JSON] DirPath :> Post '[JSON] (DirStatus ())
    :<|> "delete" :> ReqBody '[JSON] DirPath :> Post '[JSON] (DirStatus ())
```

File Structure

Here you can see simplified file structure of a project:

```
.
├── client
│   └── Main.hs
├── name-server
│   └── Main.hs
├── shared
│   ├── API
```

```
│   │   ├── NameServer.hs
│   │   └── StorageServer.hs
│   ├── API.hs
│   └── Lib.hs
├── storage-server
│   └── Main.hs
└── test
    └── Spec.hs
```

- Folders `client`, `name-server`, `storage-server` contain code required only for the client, the name server and for the storage server respectively.
- Folder `shared` contains code which can be imported to every executable.
- Folder `test` constains tests for a code.

Member contribution

Mihail Kuskov * Client server * Storage server * Report

Alfiya Mussabekova * Report * Project deployment on AWS * Docker containeranization * Client server

Nikita Aleschenko * Name server * Project deployment on AWS * Report

Conclusion

The stated goals were achieved, one of the difficulties that we met during the project is parsing of *relative* and *absolute* file *paths* in console client commands.

Another one is that sometimes because of luck of time we made design decisions which allowed us to code faster, but now they are harder to understand and maintain. For example, error massages in our implementation are not handled in fancy way, we just directly forward them to user.

The difficulties also appeared in docker deployment part, because in order to build and compile haskell project we need *stack* (a cross-platform program for developing haskell projects), which image is about 11GB. Therefore, we needed to have 2 stage *Dockerfile* in order to make docker images with components of *DFS* light.

What was good? * Purely functional * Strong type system * Team organization

What could be improved? * Time management * Implement *change dir* on namenode

References

- Link to Github repository
- Link to Project description
- Link to Presentation
- Link to docker image for name server
- Link to docker image for storage server
- Link to docker image for client

Useful Links

- Haskell Stack tool
- Haskell Platform
- directory-tree
- MultipartData
- disk-free-space
- Parsing command line
- servant
- servant-server
- servant-client
- aeson
- bytestring
- cryptonite
- splitmix
- haskeline