

*Stockage dans le cloud décentralisé*

Documentation technique

Cette documentation s’adresse à toute personne sur le point de participer au développement technique du projet StoreIt. Il détaille la conception de chaque composant logiciel du projet ainsi que le protocole réseau utilisé.

***Description du document***

|  |  |
| --- | --- |
| **Titre** | DA2\_StoreIt |
| **E-Mail** | Storeit\_2017@labeip.epitech.eu |
| **Auteurs** | Adrien MOREL  Kenny NGUYEN  Alexandre FULGONI  Louis MONDESIR  Romain GJURA |
| **Date** | 19 Juillet 2016 |
| **Version du modèle** | 1.0 |

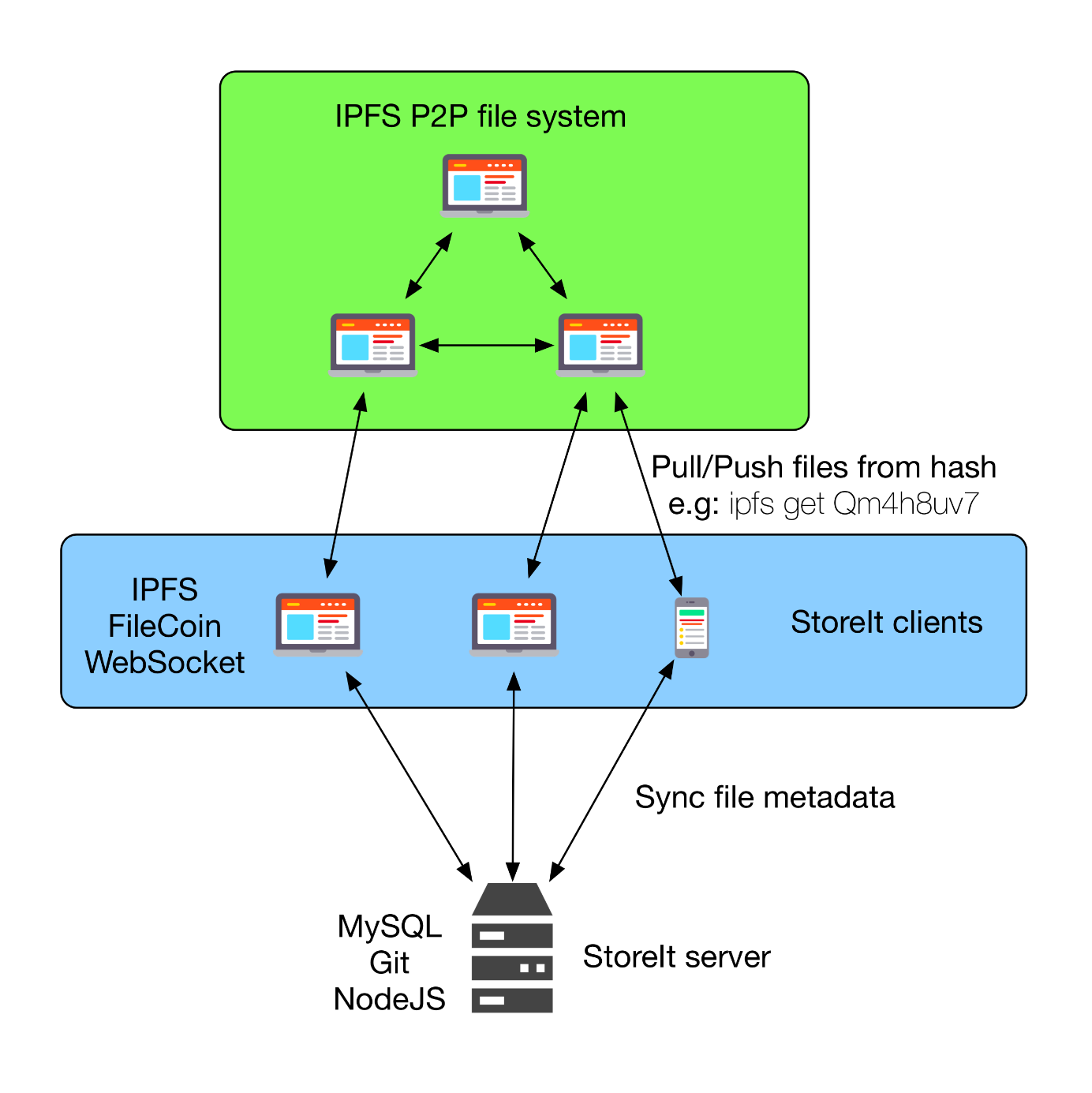
***Tableau des révisions***

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Auteur** | **Section(s)** | **Commentaire** |
| 19/07/2016 | Kenny NGUYEN | Toutes | Mise en page du document |
| 15/07/2016 | Adrien MOREL  Alexandre FULGONI  Louis MONDESIR  Romain GJURA | Toutes | Rédaction des parties respectives |
| 23/07/2016 | Adrien Morel | Serveur | Améliorations |

SOMMAIRE

Vue globale du projet

L’architecture du projet est résumée dans le schéma ci dessous.



Le serveur est développé en javascript et assure la synchronisation des utilisateurs. IPFS est un outil qui facilite l’envoi en P2P des données présentes dans les fichiers des utilisateurs. Le serveur ne manipule jamais les données des fichiers directement mais ne connaît que les métadonnées, la structure de l’arborescence d’un client et les hash IPFS. Il a aussi pour rôle d’imposer aux client le téléchargement blocs de données arbitraire afin de produire la persistance/disponibilité des fichiers.

Le protocole de communication réseau

# 1. Introduction

This is the documentation for the StoreIt protocol. It is used to com- municate with our server. We will be implementing this protocol on top of WEBSOCKETS to enjoy its messaging model. Everything should be JSON objects.

# 2. The JSON data structures

## 2.1 Command

{  
 "uid": unique\_command\_id,  
 "command": command\_name,  
 "parameters": {  
 "parameter1Name": parameter1,  
 "parameter2Name": parameter2,  
 }  
}

## 2.2 Response

{  
 "code": code,  
 "text": response\_message,  
 "commandUid": command\_id,  
 "command": "RESP",  
 (optional) "parameters": {  
 ...  
 }  
}

TODO: document possible errors.

## 2.3 Commands

### JOIN

From a client to the server. This is the first request to make whenever a client wants to get online.

{  
 "uid": 263,  
 "command": "JOIN",  
 "parameters": {  
 "authType": "fb", // fb for facebook and gg for google  
 "accessToken": "34j8b4jhb343hbKJH54",  
 "hosting": [  
 'QmNMNRCgNBvkdXyXuVa2cHwTJJ9wtJQht1Njx1pqNBC9cV',  
 'QmYwAPJzv5CZsnA625s3Xf2nemtYgPpHdWEz79ojWnPbdG'  
 ]  
 }  
}

The response will contain a FILE object named "home" and user profile info. Example :

{  
 "code": 0,  
 "text": "welcome",  
 "commandUid": 42,  
 "command": "RESP",  
 "parameters": {  
 "home": FILEObject,  
 "userProfile": {"picture": "pic.jpg"}  
 }  
}

### FDEL

From a client or the server. Delete a file/directory.

{  
 "uid": 765,  
 "command": "FDEL",  
 "parameters": {  
 "files": ["/a.txt", "/archive/b.txt", "/dir"]  
 }  
}

### FADD

From a client or a server. Add a file to the user three.

{  
 "uid": 766,  
 "command": "FADD",  
 "parameters": {  
 "files": [FILEObject, ...]  
 }  
}

### FUPT

From a client or a server. Update a file.

{  
 "uid": 767,  
 "command": "FUPT",  
 "parameters": {  
 "files": [FILEObject, ...]  
 }  
}

You should do only one FUPT per file/directory and omit the files parameter of your directory. For example, if your home is :

| foo  
L\_\_\_ bar.txt  
L\_\_\_ pictures

And you want to update foo's timestamp, just send :

{  
 "uid": 767,  
 "command": "FUPT",  
 "parameters": {  
 "files": [{  
 "path": "/foo",  
 "metadata": "updated metadata",  
 "IPFSHash": null,  
 "isDir": true,  
 "files": null  
 }]  
 }  
}

### FMOV

From a client or a server. move or rename a file.

{  
 "uid": 768,  
 "command": "FMOV",  
 "parameters": {  
 "src": "/foo/bar.txt"  
 "dest": "/foo/toto.txt"  
 }  
}

If you are moving a file, please don't omit the file name in the destination. For example :

DON'T DO:

{  
 "src": "/foo/bar"  
 "dest": "/target/"  
}

expecting to move /foo/bar into /target/bar

DO:

{  
 "src": "/foo/bar"  
 "dest": "/target/bar"  
}

###### FSTR

From the server to a client Store an IPFS object

{  
 "uid": 7668,  
 "command": "FSTR",  
 "parameters": {  
 "hash": "QmYwAPJzv5CZsnA625s3Xf2nemtYgPpHdWEz79ojWnPbdG",  
 "keep": true|false  
 }  
}

If the "keep" parameter is false, the object should be deleted. Otherwise it should be downloaded from IPFS and stored in the local repository.

## 2.4 FILE object

This object describe a file or a directory.

javascript {

"path": "/foo/bar",

"metadata": metadata,

"IPFSHash": "IPFS hash of all the data in the file",

"isDir": true,

"files": {

"foo.txt": FILEObject,

"someDirectory": FILEObject,

}

}

Le serveur StoreIt

# Obtenir le code

Le code source du serveur est disponible sur github (<https://github.com/Sevauk/storeit>).

git clone https://github.com/Sevauk/storeit

Consultez le document sur l'installation pour installer et lancer le serveur.

# Norme

Le code doit respecter les normes ECMAScript en utilisant ESLint.

# Les tests

Actuellement il existe des tests unitaires qui utilisent les frameworks chai et mocha. Ils se trouvent dans le fichier test.js et se lancent via la commande `npm test`

Les outils qui vont être mis en place prochainement sont les suivants :

travis CI + slack intégration avec des webhooks

istanbul (test coverage)

coffeescript (language) pour les tests unitaires.

# Description des fichiers de code du serveur

* ├── **auth**.js: Validation du token d’authentification reçu par la requête JOIN
* ├── **lib** -> ../../lib/js/src: fichiers partagés avec le client desktop
* │   ├── **log**.js: outils pour le log
* │   ├── **protocol**-**objects**.js: objects JSON du protocole réseau
* │   └── **tree**.js: fonctions de manipulation de l’objet FILEObject du protocole
* ├── **main**.js: point d’entrée de l’application
* ├── **parse**.js: parsing des commandes réseau reçues
* ├── **store**.js: décision de l’envoi de commandes FSTR: la persistence des données utilisateur
* ├── **test**.js: tests unitaires du serveur
* ├── **tool**.js: outils génériques pour le serveur
* ├── **user**.js: objet qui représente un utilisateur/client et fonctions qui manipulent son arbre
* └── **ws**.js: gestion des évènements WebSocket et fonctions pour répondre à un utilisateur

Application Android de StoreIt

# Liste des logiciels requis

* Android Studio 2.1
* Git

# Procédure de mise en place du projet

* Cloner le dépôt git
* Depuis Android Studio ouvrir le projet situé dans le répertoire **/src/android-app/StoreIt**
* L'ensemble des bibliothèques nécessaires seront automatiquement téléchargées via une tâche gradle

# Architecture du projet

Le projet est organisé autour de 3 grandes parties :

1. Un module de gestion réseau (websocket)
2. Un module de gestion des fichiers
3. L'ensemble du code lié à l'interface utilisateur

Chacun de ces modules se situe dans un package Java différent

# Procédure de test

Des tests unitaires ont été mis en place. Ils reposent sur l'utilisation de JUnit dont l'utilisation est détaillée ici <http://developer.android.com/tools/testing/testing_android.html>

Pour l'instant seulement la fonctionnalité de création de hash utilise des tests unitaires.

Application iOS de StoreIt

# Conception du projet

Le projet est constitué de trois parties principales :

Network

Cette partie contient toutes les classes relatives à la gestion du réseau, en l'occurrence les classes Server et Client, ainsi que plusieurs classes utilitaires, comme la classe RequestBuilder qui permet la construction des requêtes envoyées par les deux classes citées précédemment. Client permet de recevoir et d'envoyer des requêtes au serveur, alors que Server permet de recevoir et d'envoyer des requêtes à d'autres clients.

FileManagement

Cette partie contient toutes les classes relatives à la gestion de fichier, aussi bien la gestion du dossier synchronisé StoreIt que le dossier de stockage des chunks.

Views

Cette partie contient toutes les vues de l'application.

# Test unitaires

Les tests unitaires se trouvent dans leurs dossiers respectifs Xcode : StoreItTests et StoreItUITests. Ces dossiers contiennent des sous-dossiers correspondants aux parties évoquées ci-dessus.

# Norme

Il n'y a pas de norme de codage particulière, si ce n'est qu'il faut suivre au possible les conventions établies par Apple

[Consulter ces conventions ici](https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/TheBasics.html#//apple_ref/doc/uid/TP40014097-CH5-ID309).

Web-App

# Technologies

* Langage: JavaScript (ES2015 + Tc39 stage-3)
* Framework: AngularJS 1.5+
* Templates: Jade
* Stylesheets: PostCss
* Gestionnaire de dépendances: JSPM
* Automatiseur de tâches: Gulp
* Testing: Karma + Mocha + Chai + Istanbul

# Structure

Les sources de l'application sont organisées en Modules Angular. L'application contient 2 types de modules:

1. **Components**: Route + Controlleur + Vue (Template et Style). Un component est intimement lié à ça vue, mais peut être composé à partir de component plus petits.
2. **Core**: Contient de la logique réutilisable (principalement des services et models)

Example de component:

home

| home.js

| home.jade

| home.css

|

|\_\_sidebar

| sidebar.js

| sidebar.jade

| sidebar.css

# Développement

Suivre les instructions d'installation, puis utilisez:

npm run dev