Online File-Sharing Platform

Brice GUILLAUME, Adam KETTANI, Mathieu LAPEYRE, Jérémy PARRIAUD
Georgia Institute of Technology, Atlanta, GA - ECE 6102, Spring 2016
brice.guillaume[@]gatech.edu, adam.kettani[@]gatech.edu, mathieu.lapeyre[@]gatech.edu, jparriaud3[@]gatech.edu

I. ABSTRACT

Our team is interested in designing an online file-sharing platform that supports large-scale reliable storage. This system allows users to download and upload files with respect to their user group membership. Once connected to the platform, users can join one or several group(s) on which members exchange files by uploads and downloads. Only group members have access to the files of the group.

The team addressed major consistency and availability concerns as well as load balancing and secure authentication. To interact with the system, we designed a user-friendly RESTful interface. As a result, our online file-sharing platform can handle the failure of at most half of the servers located within a grid network.

II. SYSTEM ARCHITECTURE

The project can be divided into seven basic functionalities. ***Figure 1*** shows the different functionalities we are implementing.

A. File access architecture

Our file system architecture is a two-level structure. At the top level are folders which represent the groups users can join. In each group, there is a flat namespace where files are uniquely named.

B. Fault-tolerant platform

To make the servers agree on a consistent state, our architecture runs the Raft consensus algorithm[1]. Raft is a recent algorithm which is equivalent to Paxos in fault-tolerance but easier to understand and to implement. In our case, a cluster of n servers can continue to operate even if (n/2)-1 servers fail.

C. Data replication

Servers can crash, we must prevent the platform from losing files. To do so, we will implement a data replication scheme. This relies on two elements, the replication through Grid-Quorum to make the data available even in case of servers failures (crash or byzantine) and the use of Raft to make the replication consistent across the lines (recovery of failed servers). The reading operations are made on the columns and the write operations on the rows (using Raft).

D. Group-based sharing

Our platform gives users access to personalized files, depending on which "group" they belong to. A user can belong to several groups. The users and groups management is handled by a subset of servers responsible for maintaining the data bases related to these aspects.

E. Authentication-based access

To ensure the restricted access to the platform, users must identify thanks to a login/password scheme. Our system relies on OAuth2 protocol.

F. Distributed cryptography

This feature has been removed from the final version and is a part of the future work section.

G. File versioning

This feature has been removed from the final version and is a part of the future work section.

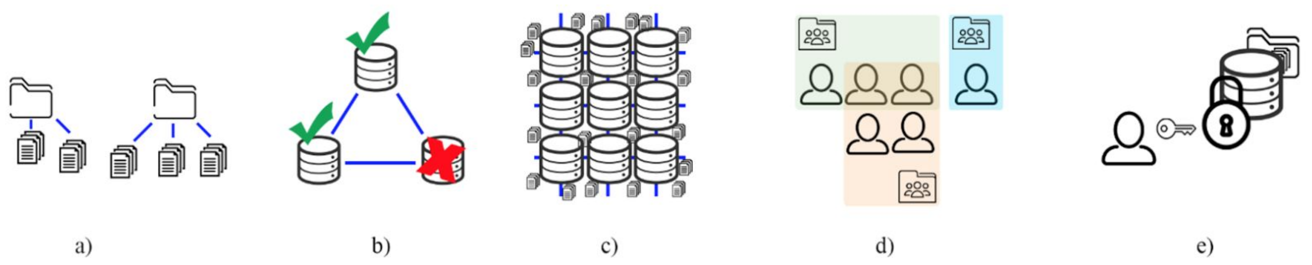


Figure 1: Basic functionalities we implement. a) 2-level file system architecture, b) Fault tolerance, c) Data replication, d) Group-based sharing, e) Authentication-based access

III. TOOLS USED

We decided to run our project on the Google Cloud platform.

A. Hardware

All the nodes consist in Virtual Machines under Linux OS running on the infrastructure of Google Cloud. Each of them has a dedicated storage and a network access through the Internet.

B. Software

Because our implementation will be powered by JAVA (EE), we have deployed Tomcat servers to support the main functions. The nodes are by HTTP through a REST API interface (JAX-RS) and store informations in a local database provided by MySQL.

C. Client

Apart from the previous element, in order to access to the service provided by our project, a web interface has been created to manage the upload and download of file securely (HTML, JQuery, SSL).

IV. IMPLEMENTATION

A. Distributed infrastructure for availability

The data replication is achieved on a Grid-Quorum architecture across all the servers of one line. This design leads to a system which is faults resilient because servers can crash or become byzantine without compromising the availability of the whole system. The settings in terms of writing and reading are set up under the hypotheses that up to two servers by lines can be byzantines (if they are they cannot give the same false response) and up to two servers by lines can crash (cf. figures below).

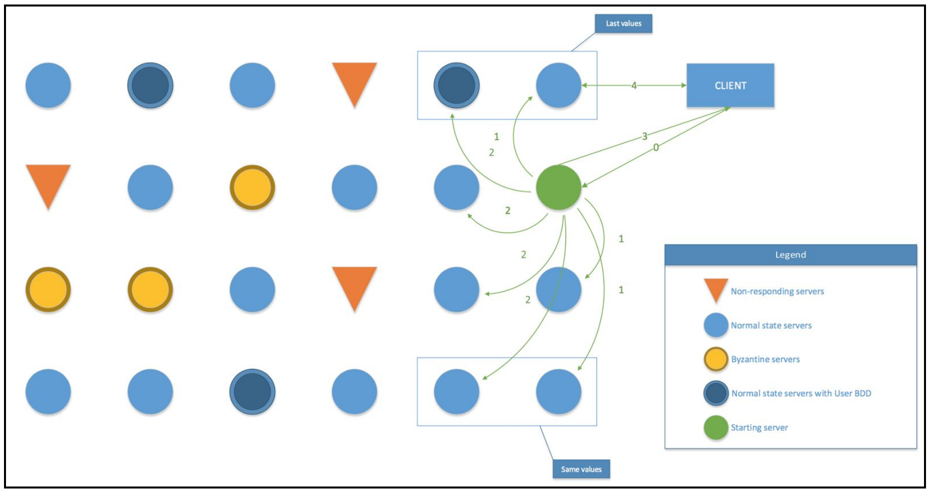


Figure 2: A simple reading operation. A read is done on several columns to get answers from more correct servers than the maximum estimated number of byzantine servers.

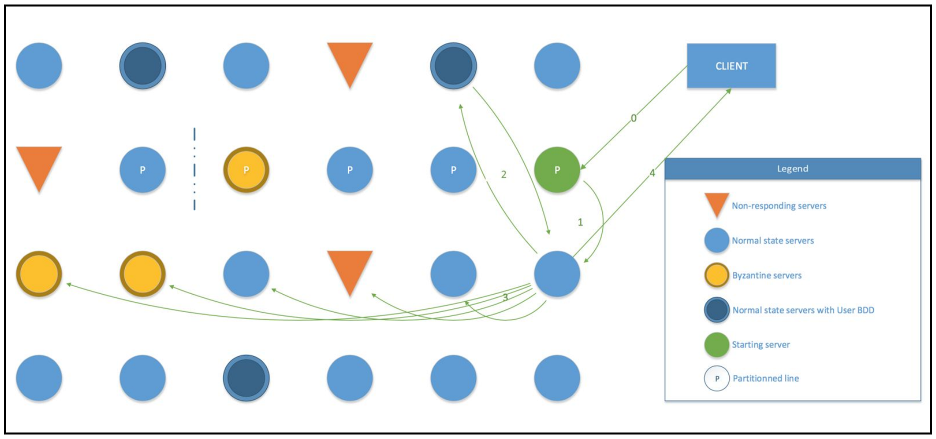


Figure 3: A simple write operation. The file is replicated using a consensus protocol (see Raft protocol). The main management servers (hosting information about the whole database) are informed of this new write.

The distributed infrastructure can be split into two parts, the user management and the files download and upload.

The first part relies on the management servers (BDD servers) which host information about the databases (users, groups, filelist). The mechanisms used to replicate the data among the management servers are the following:

- Read on all servers and take the informations that can be found on a majority of servers
- Write on all servers and consider the information commit if a majority of servers acknowledges

N.B.: There are few of management servers, all the other servers are normal ones and host the files users are uploading to and downloading from.

The second part relies on another category of servers (which also contains the management servers) which is responsible for the files writing and reading. The architecture implemented relies on a quorum system on which the data are replicated across the lines (with RAFT) and read on the columns. The reading operation is performed in two steps. First, the server which is queried by the client retrieves the metadata (file presence and timestamp) of the wanted file from all the lines by asking the servers of its column and recursively on each line until a consistent value is found for each row. Secondly, one of the servers of the row with the highest consistent timestamp is selected and its IP is sending back to the client which can make the request to download the file directly from this server. This mechanism helps to prevent effects of byzantines and crashed servers.

B. Consensus protocol and replication for consistency

To achieve a consistent state across the machines, we use the Raft consensus algorithm. Raft separates the key elements of consensus, such as leader election, log replication, and safety, and it enforces a stronger degree of coherency to reduce the number of states that must be considered [1]. The key element is the strong leader: log entries flow exclusively from the leader to the followers, which simplify the log management.

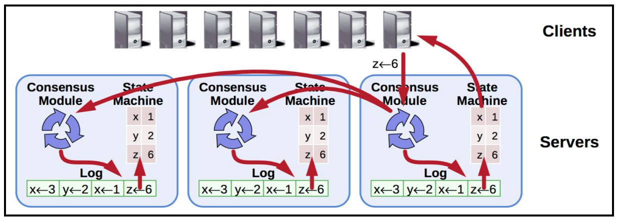
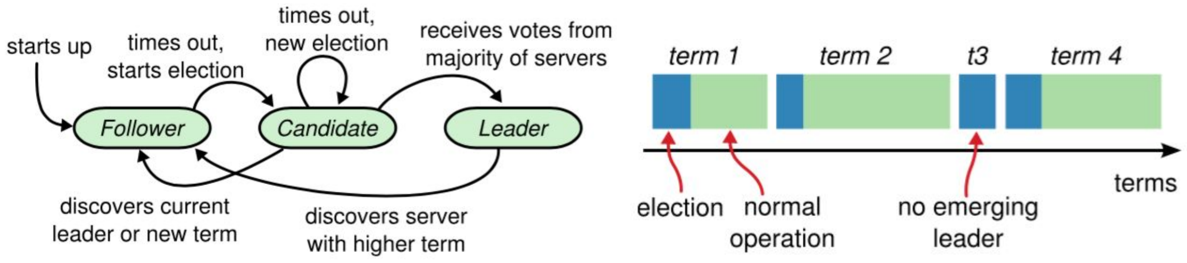


Figure 4: Typical architecture for a consensus system [1].

Raft servers communicate using remote procedure calls (RPCs), and the basic consensus algorithm requires only two types of RPCs. RequestVote RPCs are initiated by candidates during elections and AppendEntries RPCs are initiated by leaders to replicate log entries and to provide a form of heartbeat (Section IV.B.1).

1. Leader Election

Raft uses a heartbeat mechanism to trigger leader election. After a random election timeout, a follower issues vote request to the other servers and wait for the majority. To maintain the leadership, the newly elected leader send periodic heartbeat signals before the followers' election timeout runs out.



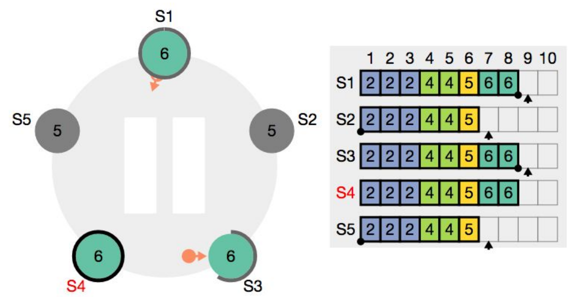
starts up
times out, starts election
times out, new election
receives votes from majority of servers
Follower
Candidate
Leader
discovers current leader or new term
discovers server with higher term
term 1
term 2
t3
term 4
terms
election
normal operation
no emerging leader

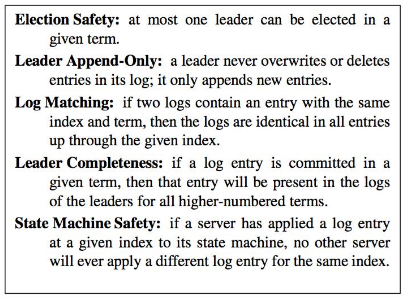
Figure 5: Server states. Followers only respond to requests from other servers. If a follower receives no communication, it becomes a candidate and initiates an election. A candidate that receives votes from a majority of the cluster becomes the new leader. Leaders indefinitely operate unless they fail. Each leader manages the cluster until the end of his term (time unit).

While waiting for votes, a candidate may receive an AppendEntries RPC from another server claiming to be leader. If the leader's term (included in its RPC) is at least as large as the candidate's current term, then the candidate recognizes the leader as legitimate and returns to follower state.

2. Log Replication

To avoid performing a strong consistency across all the servers, we implement the Raft consensus algorithm across every servers of the same line of the Grid-Quorum. Log entries that are shared between the servers are a hashtable of every files stored on the line.





Election Safety: at most one leader can be elected in a given term.

Leader Append-Only: a leader never overwrites or deletes entries in its log; it only appends new entries.

Log Matching: if two logs contain an entry with the same index and term, then the logs are identical in all entries up through the given index.

Leader Completeness: if a log entry is committed in a given term, then that entry will be present in the logs of the leaders for all higher-numbered terms.

State Machine Safety: if a server has applied a log entry at a given index to its state machine, no other server will ever apply a different log entry for the same index.

Figure 6: S4 is the leader, S5 and S2 crashed and will recover from log inconsistency later [3]. The system can still operate as more than half of the total number of server is running. The leader committed the first 8 entries as they have been replicated on a majority of nodes, in this case nodes 1 and 3. On the right, Raft guarantees that each of these properties is true at all times.

3. Safety

The term numbers in log entries are used to detect inconsistencies between logs and to ensure some of the properties in Figure 6. Each log entry also has an integer index to detect its position in the log.

The leader keeps track of the highest index it knows to be committed, and it includes that index in future AppendEntries RPCs (including heartbeats) so that the other servers eventually find out. Once a follower learns that a log entry is committed, it applies the entry to its local state machine [1].

In Raft, the leader handles inconsistencies by forcing the followers' logs to duplicate its own. This means that conflicting entries in follower logs will be overwritten with entries from the leader's log. At first sight, this can be considered as dangerous, but when coupled with the following restriction, it appears to be very safe: Raft uses the voting process to prevent a candidate from winning an election unless its log contains all committed entries.

To be elected, a candidate's log must be at least as up-to-date as any other log in that majority. Raft determines which of two logs is more up-to-date by comparing the index and term of the last entries in the logs. If the logs have last entries with different terms, then the log with the later term is more up-to-date. If the logs end with the same term, then whichever log is longer is more up-to-date.

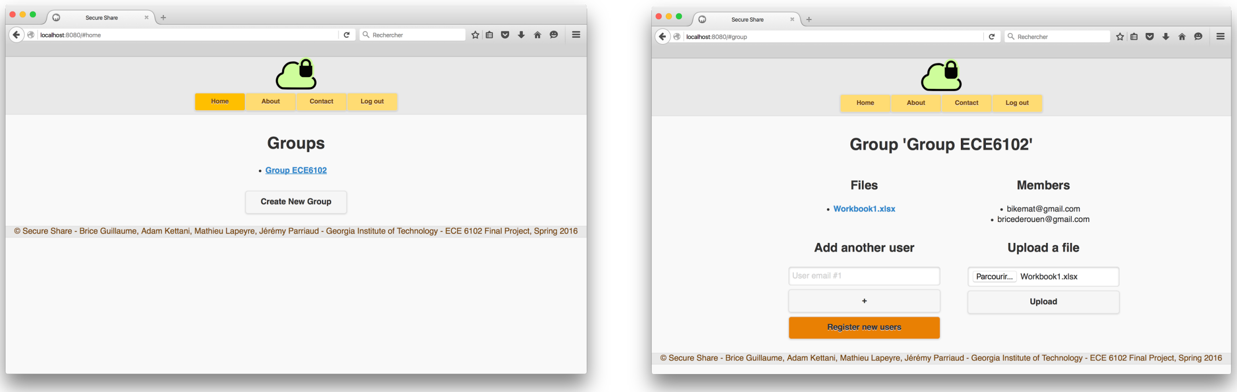
C. User interface

Users connect to the file sharing platform through a web client interface. This web interface is implemented in HTML/CSS/JavaScript and relies in part on the JQuery Mobile library. This library typically does a pretty good job to adapt to different media devices (laptops, smartphones…).

The front-end is basically composed of one single HTML document integrating multiple distinct pages. Users switch between these pages by browsing through the interface. Each request made by the client (by clicking on a button for instance) is interpreted as an HTTP POST request carrying a JSON file, which is directly addressed to a particular server URL following our REST architecture. The operational component is a JavaScript file dynamically matching user interaction with background commands.

Note that before being uploaded files are encoded in base64 on the client side. So the servers store them as a single string in their database. At download time, the files are base64-decoded on the client side. Base64 is useful here as it allows us to integrate the functions to the JSON interface. However, it expands the file length to 33%, so it is clearly not optimized.

Here are a couple of screenshots of what we did (display groups and view/manage group pages).



© Secure Share - Brice Guillaume, Adam Kettani, Mathieu Lapeyre, Jérémy Parriaud - Georgia Institute of Technology - ECE 6102 Final Project, Spring 2016

Figure 5: The user interface

D. User authentication

OAuth2 protocol is being implemented as a high-level layer on our system using Google's API. When a user wants to connect to our platform, he is redirected to a Google authentication webpage. After signing in, Google sends back a token that will allow our application to retrieve the user infos.

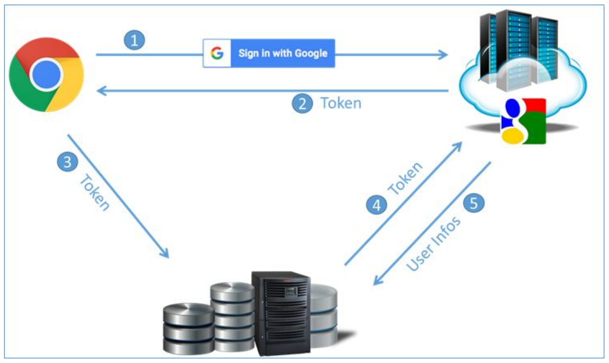


Figure 6: OAuth protocol

In the server side of our application, the user information retrieved from Google allows us to authenticate the user using its google ID. The latter is associated with a list of groups and content the user is allowed to access.

By using Google authentication API, we externalize the authentication process and guarantee the correct mapping between the authenticated user and the content he/she has access to. A possible future work would consist on implementing our own authentication protocol.

The code used for authentication is inspired from examples found on the Google developers website, but was modified to meet our architecture needs.

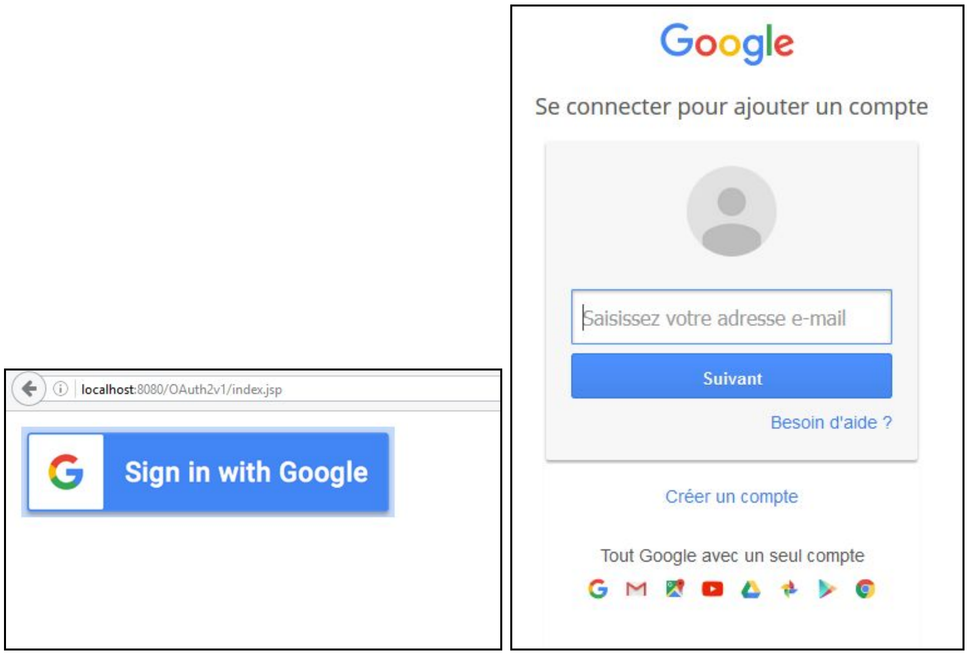
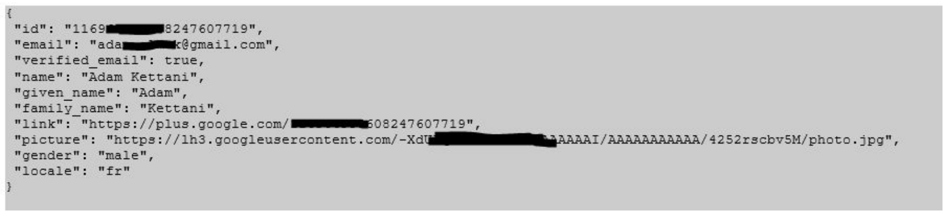

localhost:8080/OAuth2v1/index.jsp
Sign in with Google
Google
Se connecter pour ajouter un compte
Saisissez votre adresse e-mail
Suivant
Besoin d'aide ?
Créer un compte
Tout Google avec un seul compte

Figure 7: Sign in button in our application and Google's authentication webpage.



```
{
 "id": "1169[redacted]8247607719",
 "email": "ada[redacted]k@gmail.com",
 "verified_email": true,
 "name": "Adam Kettani",
 "given_name": "Adam",
 "family_name": "Kettani",
 "link": "https://plus.google.com/[redacted]608247607719",
 "picture": "https://lh3.googleusercontent.com/-XdU[redacted]AAAAAI/AAAAAAAAAAA/4252rscbv5M/photo.jpg",
 "gender": "male",
 "locale": "fr"
}
```

Figure 8: Retrieved user's information after login

V. EVALUATION AND PERFORMANCE

To assess the performance of our platform we can measure the upload and download times of files with different sizes. The two graphs in Figure 9 show the result we got, with file sizes doubling every time.

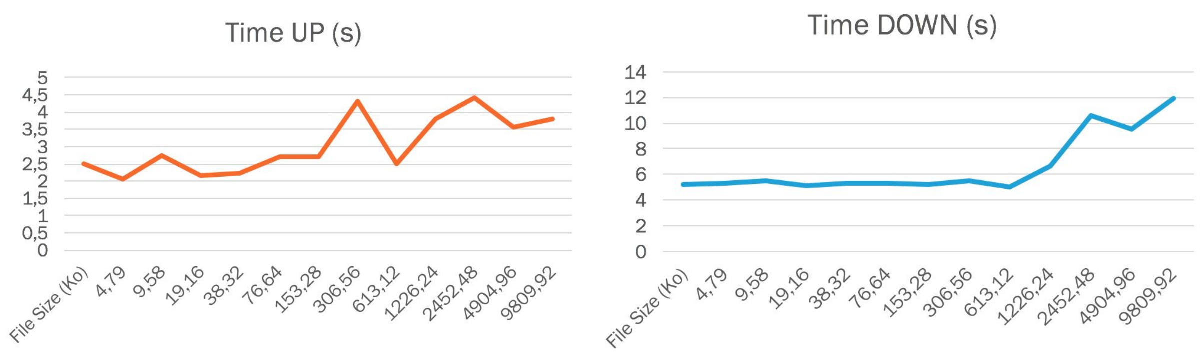


Figure 9: Upload and Download time. Upload time can be dramatically reduced by lowering Raft timeouts

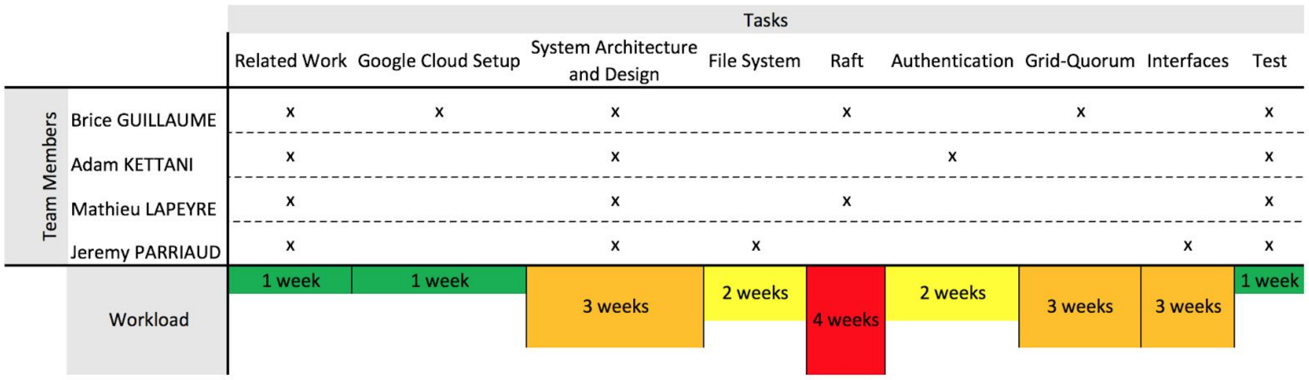
We notice that the upload time increases irregularly with the size of the file. This can be explained by the fact that we set the Raft timeout to send replica every 2.5 seconds. So if the upload happens at the beginning of the timer, then it has to wait for at least 2.5 seconds, but it can arbitrarily be faster: this is the configurable part of the upload time. The overall increase can be explained by the increasing size of the files, which implies longer upload times.

We notice that the download time increases regularly with the size of the file. The first part of the download consists in retrieving the IP address of the server where the file is stored - this operation is almost constant in time. The overall increase can be explained by the increasing size of the files, which implies longer download times.

Finally, as we read a column but get several values from each individual line - to get a consistent value for each line - we consider that 1 crashed server and 1 byzantine server can occur on the same line (assuming 4 servers per line).

VI. WORKLOAD

One can find in ***Figure 10*** the workload distribution throughout all the team members. Each person focused on his main task, but has also been implied in other tasks when necessary.



		Tasks								
		Related Work	Google Cloud Setup	System Architecture and Design	File System	Raft	Authentication	Grid-Quorum	Interfaces	Test
Team Members	Brice GUILLAUME	x	x	x		x		x		x
	Adam KETTANI	x		x			x			x
	Mathieu LAPEYRE	x		x		x				x
	Jeremy PARRIAUD	x		x	x				x	x
	Workload	1 week	1 week	3 weeks	2 weeks	4 weeks	2 weeks	3 weeks	3 weeks	1 week

Figure 10: Workload distribution. Each person has focused on his main task but has also been implied in other tasks.

VII. CONCLUSION

The goal of our project was to address the issue of dependability in the field of files sharing in a wide sense (authentication, access control, consistency of data and availability of servers).

By combining the characteristics of a distributed system and the tools to achieve dependability, this project allows users to authenticate to the file sharing platform, create sharing groups, and share files by saving them into the platform’s database.

In spite of its simple design on the client side, this project implied a lot of backend features, with more than 40 Java classes that need to interact between each other.

VIII. FUTURE WORK

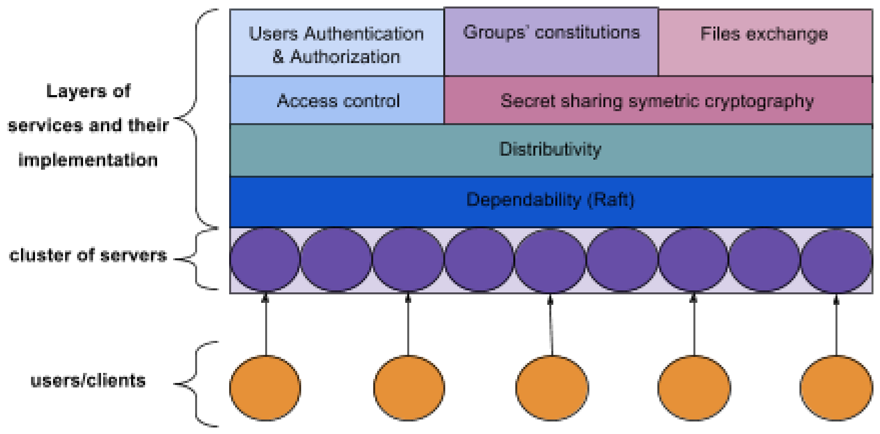
There is a lot of room for improvements. Our project was designed to prove that we can create a simple yet efficient reliable file-sharing system. First, by reducing the timeouts in the Raft protocol, we can speed up the overall running-time of the system (e.g. faster uploads).

Secondly, files could be encrypted by a secret-sharing algorithm to further enhance the security of the platform. This would allow users of a same group to secure their files even if servers leaked data. Moreover, the platform could implement HTTPS for network exchanges.

An additional file integrity check could be implemented by sending to the client the hash of the file corresponding to the consistent state given by the first step of the read operation. So the download of a tampered file would be noticed by the client.

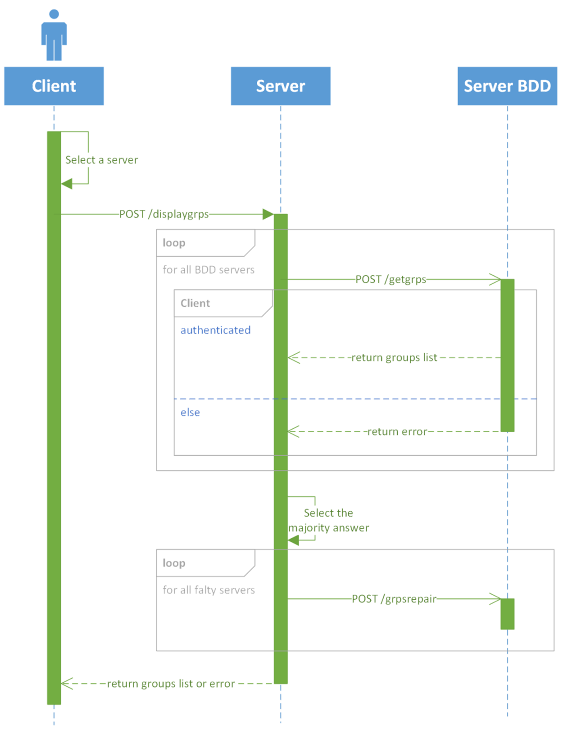
IX. APPENDIX

A. High-level architecture


Layers of services and their implementation
Users Authentication & Authorization
Groups' constitutions
Files exchange
Access control
Secret sharing symetric cryptography
Distributivity
Dependability (Raft)
cluster of servers
users/clients

Appendix A. High-level architecture of the file-sharing platform

B. UML Sequence diagram (Display Groups)



Appendix B. UML Sequence diagram . Flows for the client to retrieve the list of the groups he belongs to with a repair process on the faulty servers

C. RESTful interfaces (sample)

URL	Service	Input / Output	
/displayfiles (POST)	Get the list of files within a group	I	{“token”: “token_str”, “group_id”: “id”}

		O	Success: {“files_list”: [{“name”: “name1”, “id”: “id1”}, {“name”: “name2”, “id”: “id2”}]}
			Fail: {“error”: “error_msg”, “error_type”: “error_type”}

/displayusers (POST)	Get the list of users within a group	I	{“token”: “token_str”, “group_id”: “id”}

		O	Success: {“users_list”: [“mail1”, “mail2”]}
			Fail: {“error”: “error_msg”, “error_type”: “error_type”}

/getfile (POST)	Download a file from a server	I	{“token”: “token_str”, “group_id”: “id”, “file_id”: “file_id”}
		O	(Success: HTTP response (content type: application)) or Success: {“file”: “base64_file”} (+33%) Fail: {“error”: “error_msg”, “error_type”: “error_type”}
/uploadfile (POST)	Upload a file to the servers	I	(HTTP form request (enctype: multipart)) or {“token”: “token_str”, “group_id”: “id”, “file”: “base64_file”}
		O	Fail: {“error”: “error_msg”, “error_type”: “error_type”}

IX. References

1. Diego Ongaro and John Ousterhout, “In Search of an Understandable Consensus Algorithm”, Stanford University, 2014
2. Adi Shamir, “How to share a secret”, Communications of the ACM.
3. RaftScope: https://github.com/ongardie/raftscope
4. Authentication: https://developers.google.com/identity/protocols/OAuth2WebServer