Project Name: Distributed Cloud Storage System

Team Members: Jialun Chen, Meiqing Pan, Yun Feng

[GitHub Repo]

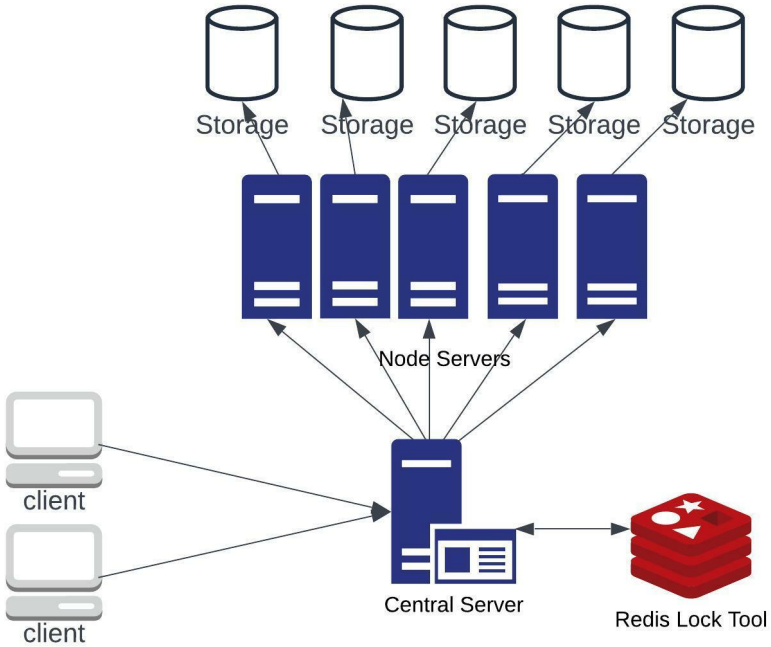
1. Summary description of the project use case

   Our team will design a distributed storage system like Google Drive or Dropbox with a terminal client interface. Feature expectations are listed below:

   Functional Requirements:
   - Users can connect to the system and interact with it through terminals;
   - Users can browse and navigate to folders;
   - Users can create and delete folders;
   - Users can upload, and delete files;
   - Users can download files;
   - Files should be synchronized in all node servers of the system;

   Nonfunctional Requirements:
   - Performance: Service should handle hundreds of users at the same time.
   - Availability: Service needs to be highly available;
   - Reliability: Data should be replicated to tolerate server failure.
   - Scalability: The system should scale with the addition of new servers.

2. Architecture overview diagram (AOD) and design description



The system is composed of clients, a central server, file servers, redis lock tool, and file storages.

A client can upload/edit/delete/view files and make/delete directories.

For the server side, we need a central server for client interaction, one redis storage to store locks, and some file server and storage for upload, editing, deleting, and searching files.

Memory caching is also an option for the system to boost performance.

System APIs:

Directory: /dirs?path=<dirpath>

- GET
  Get all the files and directories under a folder
- POST
  Create a folder by its path
- DELETE
  Delete a folder by its path

Files: /files?path=<filepath>

- GET
  Get a file by its path
- POST
  Upload a file by its path, and the file is added to a multipart form.
- DELETE
  Delete a file by its path

3. Instructions

Start the server and the client

- Navigate to the root folder.
- `docker-compose up` to build images and run container for central server and 5 node server.
- `java -cp cloud-storage-client.jar client.Client` to start the client.

Client commands

- `cd <relative or absolute path>` to navigate to a folder
- `pwd` to print current path
- `ls` to print all the files and entries in current path
- `mkdir <folder path>` to create a new folder in current directory
- `rmdir <folder path>` to remove a folder
- `curl <file path>` to download a file
- `scp <local file path>` to upload a file from your computer
- `rm <file path>` to delete a file

4. Implementation Approach (high-level design)
    1. Client

       Because files need to be transferred from the client to the central server, as well as requests to browse, delete, download and upload files, HTTP will be used in this part.

       Apache HTTP Client library is used in the client as it has some important features:
        a. Supports synchronize and asynchronize requests.
        b. Cookies. In our project, cookies can be used to maintain the user login session
        c. Authentication. Apache HTTP client supports multiple methods like Digest, NTLM, and SPNEGO, which will be used to authenticate users in our project.
        d. Compression. Apache HTTP client supports GZip and Deflate compression method. File compression can increase the throughput of our file system.
        e. Caching. Caching is supported in the Apache ATTP client. Caching will decrease the number of requests needed, thus decreasing the workload of the whole system

    2. Central server(CS)

       The features of the central server are shown below:
        - Listen for requests from clients. For GET requests, the central server evenly distributes them to other servers, and the server that received this request would send the data to the client. For requests that modify existing data, it uses Redis to lock the file, and use PAXOS to propose this file to all node servers for consistency.
        - Forward request to the node server. The central server can forward the request to node servers using HTTP request. The actual work is done in different node servers so that the system is highly scalable and the throughput is increased.
        - Manage servers. The central server can add and remove other servers. If a server is determined to be down, it will be removed from the system and stop distributing requests to it.
        - Log system information. The central server would log the state of the whole system and each server, it can also record the requests received.
        - Recover nodes from crash. The central server is able to recover nodes from crash.

    3. Node Server(NS)

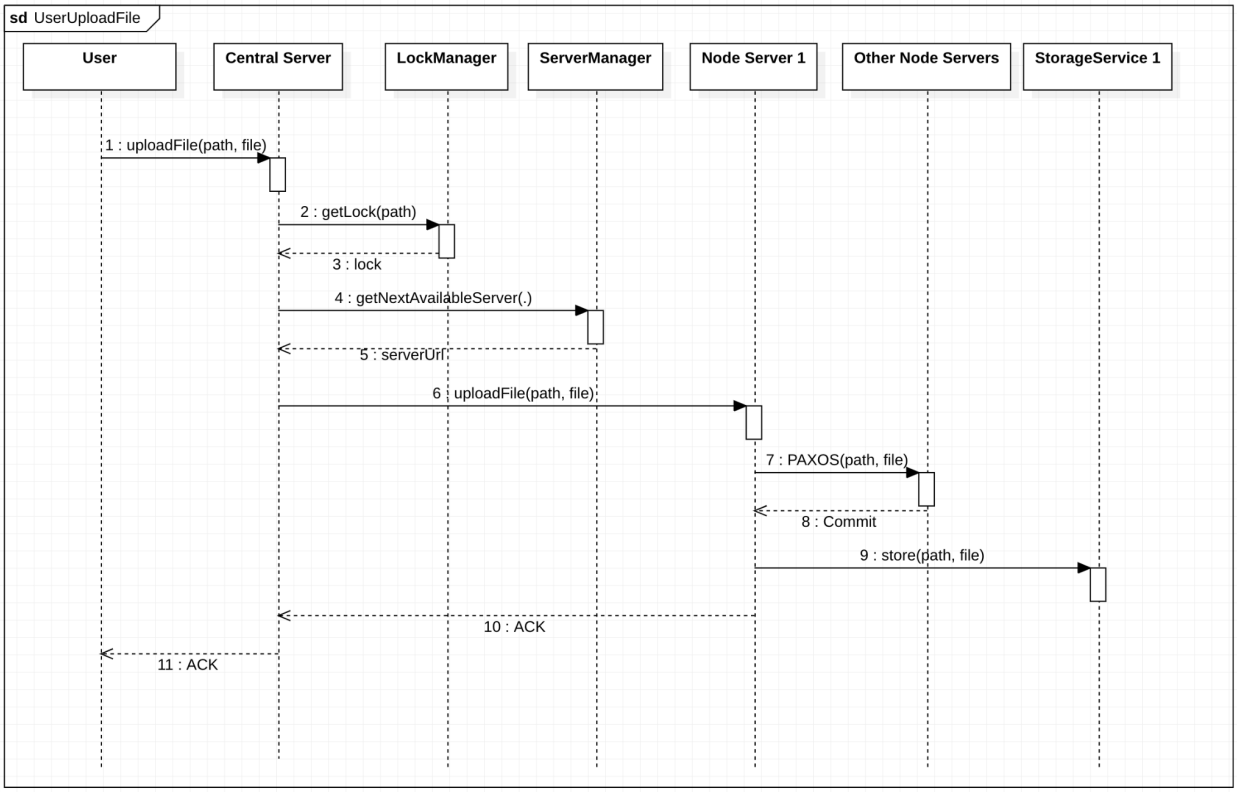
       The features of the node server are shown below:
        - Send data back to CS. The node server can send files and directory information back to CS.
        - Store files and directories to local storage.

- Listen for requests sent by CS.

5. Scenario Explanation

User upload file

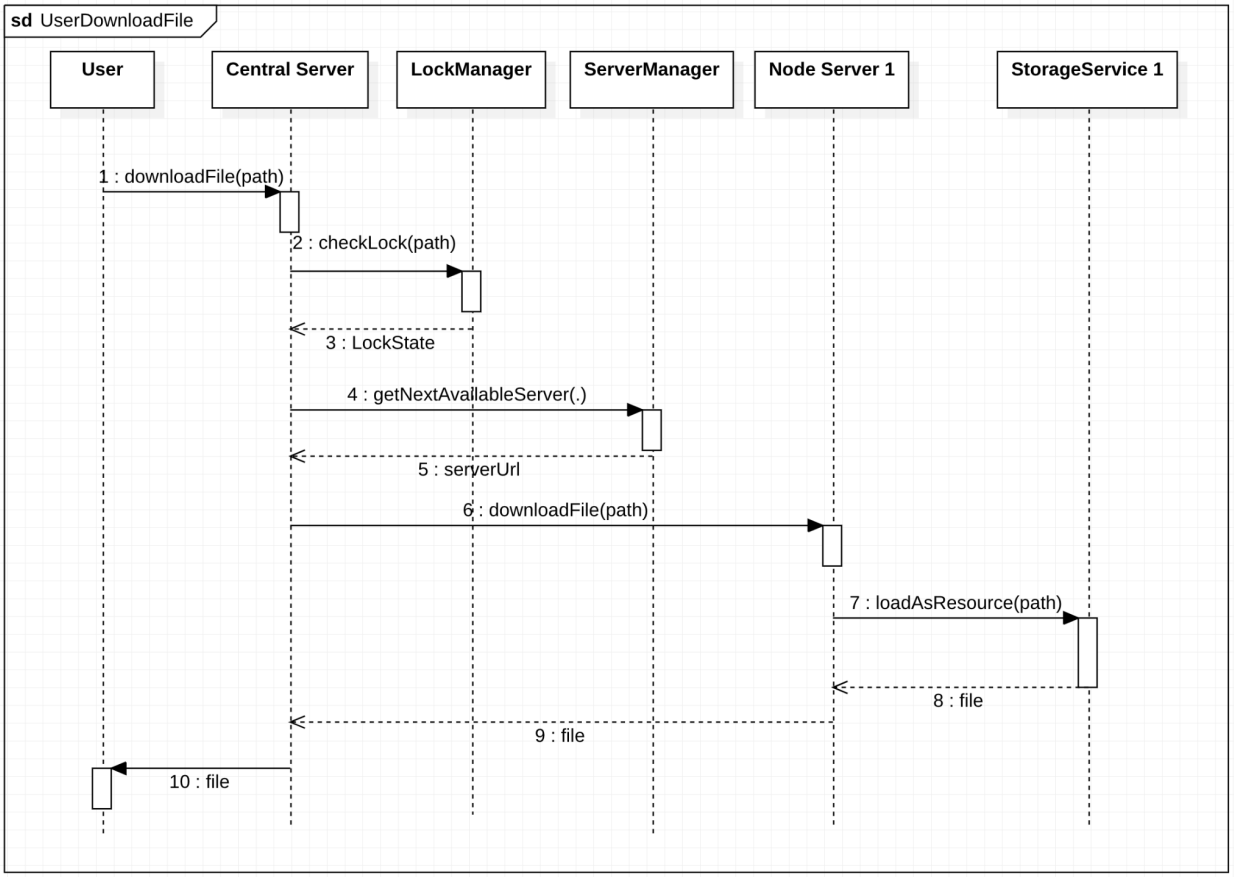
1. The user sends POST request to CS with the file and its path.
2. CS tries to get the lock by its path from LockManager; If the file is being locked, CS replies to the client with an “Upload Failed” message. Otherwise, it locks the file path.
3. CS gets the URL of the next available NS from ServerManager.
4. CS forwards the file and its path to the next available NS.
5. The NS which received the request starts the PAXOS process and proposes this value (path, file) to all other NSs.
6. If consensus is reached on this value, each NS will commit this value and store the file using their StorageService.
7. Then the NS sends an ACK message back to CS.
8. CS sends the ACK message back to the client, file is uploaded.



User download file

1. The user sends GET request to CS with the file path
2. CS checks the lock state of the file by its path from LockManager; If the file is being locked, CS replies to the client with a “Download Failed” message. Otherwise, the file can be downloaded.

9. CS gets the URL of the next available NS from ServerManager.
10. CS forwards the file path to the next available NS.
11. The NS which received the request loads the file using its StorageService.
12. Then the NS sends the file back to CS.
13. CS sends the file back to the client, the file is downloaded.



6. What libraries will use and how will you implement the project

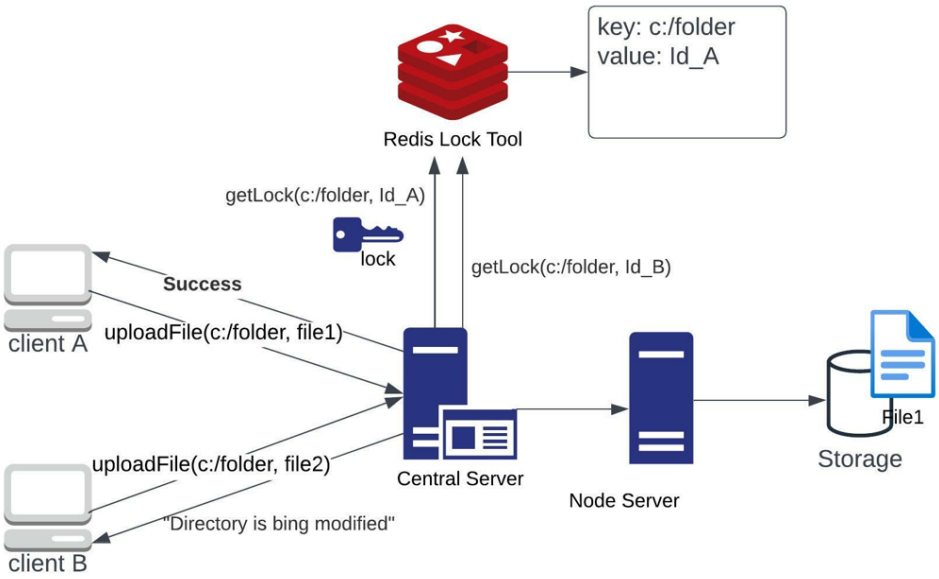
- Database: Redis
- Server framework: Java Spring
- Cache and Distributed Lock Manager: Redis
- Proxy server: Apache
- Send HTTP request: Apache HTTP Client
- Lock: Redis-based distributed lock

7. Key algorithms involved (must include at least 4 significant algorithms we covered.)

- Distributed Mutual Exclusion;
    - A distributed mutual exclusion solution is key to our distributed system, especially to guarantee safety and liveness. For example, concurrent updates on the same file should be protected by a mutex lock.

- Safety and Liveness Guarantees:
  - Mutual exclusion: Use a key-value store to ensure only one client can hold a lock at any given moment;
  - Deadlock free: The lock server automatically releases the key based on TTL;
  - Locking and unlocking must be the same client: Process generates a unique identity and inserts it into the store as value;
  - High performance: Locking and unlocking need to be efficient;
  - Fault tolerance: Clients are able to acquire and release locks as long as the majority of lock servers are up;

- Implementation:
  - Redis acts as a key-value store to obtain the global lock.
  - Locking: When the central server receives requests, it first obtains the lock from Redis. A node server will execute the business logic code after obtaining the lock. If the request does not compete for the lock, the central server will give up the execution.
  - Releasing: When the execution is finished, the central server will send a release request to Redis. As we need to ensure that A key should be released only by the client which has acquired it(if not expired), the lock server will first check id/value, and then delete the key.



Locking Process

  - Usage in Spring Boot project:
    - We use Spring Boot to integrate Redis to implement Redis distributed locks;

        - Spring Boot Data Redis Starter dependency enables us to use Redis key-value data store with Spring Data Redis and the Lettuce client;
        - Redis Enterprise Cloud enables us to use Redis quickly and easily without worrying about local server problems.
    - Solving atomic problems: Lua script is used to ensure the atomicity of operation.

- PAXOS;
    - We use Paxos to implement distributed transactions. Read operations such as upload and delete are implemented in Paxos. The central server randomly chooses a proposer and starts the Paxos process.

- Group Communication;
    - In our Paxos implementation, we broadcast our messages to all the acceptors and learners.

- Fault tolerance;
    - Paxos is fault tolerance by default. By using Paxos we can achieve fault tolerance along with consensus.

8. Expected Results

The distributed system will use Mutual Exclusion, PAXOS, Group Communication, and Fault tolerance to implement a cloud file storage service.