University of Pisa

Laurea Magistrale (MSc) in Artificial Intelligence and Data Engineering

Project

Cloud Computing

Ceph-based File Manager

Gaetano Niccolò Terranova 506349

Giacomo Piacentini 627555

Rossella De Dominicis 623902

GITHUB: https://github.com/gaetanoterra/Ceph-based_file_manager

Academic year 2020-2021

Sommario

Project Specification

Develop a distributed file storage system based on Ceph. The file storage must expose a REST interface through which external users can perform the following operations:

- **Retrieve the list of files currently stored in the system**
- **Delete a file**
- **Upload / Download a file**
- **Shows current statistics on the status of the cluster**

The application must be composed of two layers:

- Frontend layer, exposing a REST interface and receiving requests from the users;

- Backend layer;

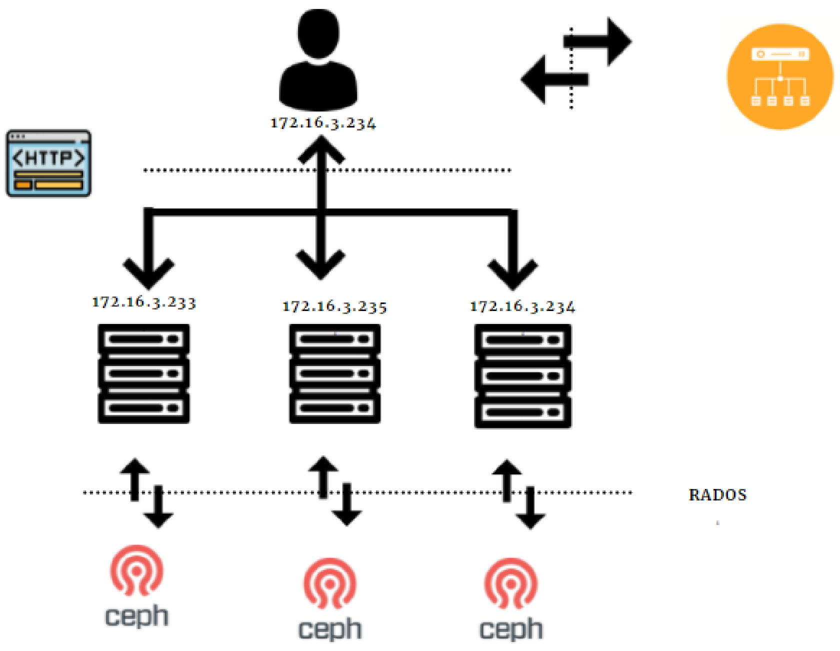
A different instance of the backend layer must be deployed on each of the three ceph-mon modules.

Architecture

The Architecture is a client-server architecture with a Load Balancer ; each module exposes a REST interface. A RESTful API is an architectural style for an application program interface (API) that uses HTTP requests to access and use data. That data can be used to GET, PUT, POST and DELETE data types, which refers to the reading, updating, creating and deleting of operations concerning resources.

The Load Balancer is connected to the client (on) in the same tier, so you don't have a bottleneck and a single bottom of failure; receives requests from it and answers with the solution. The LB is in charge of decree in which instance of the Backend to send the client http request, distributes requests between servers equally, this is for ensure scalability. Once the clients has received communication from the LB, it communicate with the servers through http message.

The servers interact with the Ceph monitor and retrieve the answers.



Frontend Methods

On the Frontend side we used the “requests” library to send HTTP requests to the backend side and to receive responses. We also imported a load_balancer file we created to use a load balancing function. On the follow we describe client methods, the main and the load_balancer methods:

- methods:
  - menu(): print the menu showing available functions to the user
- main: inside the main we have a while loop until the user use the exit command. Inside the loop the program wait a command from user. After this the load_balancer pick a server to send the request randomly and finally the request is sent entering into the corrispondent if.
  - Request = get_object_list: here we want to show the list of objects inside the pool. We need only the server id for the request
  - Request = get_object: here we want to download an object from the pool, so we need the server id and the name of the file we want to download. After the response is received a new file is created with the name we chose and the content is written.
  - Request = add_object: here we want to upload a file, so we need to give the server id, the name of the file to upload and its content.
  - Request = delete_object: here we want to delete an object from the pool, so we need to give the server id and the name of the file we want to delete.
  - Request = get_status: here we want to list the status of the cluster, so we need only the server id.
- load_balancer methods:
  - select_server(): we have an array containing the server id of our cluster, so we select randomly one element of the array using an hash of a timestamp.

Backend Methods

On the Backend side we used the Flask framework to intercept the requests from the client: with the decorator “route()” of @app.route we tell to Flask which URL trigger the following function, so we have to specify the URL and the method we used, like: @app.route('/list', methods=['GET']) .
So we have an @app.route for each type of request and the correspondent method to handle the request. On the following all methods are described:

- create_pool_if_non_existent(cluster, pool): this method check if the pool named pool exists into the cluster, if not the pool is created
- handle_request(func, *args): this method open/close the connection with the cluster and handle the request, calling the function func with arguments *args, cluster, pool. This function is called inside the routing methods after @app.route(), passing the function that will handle the request as argument.
- get_object_list(cluster, pool): method that open the pool and return the list of elements isnide of it.
- get_object(file_name, cluster, pool): method that open the pool and return the content of the file called file_name.
- add_object(file, body, cluster, pool): this method open the pool and first of all check if the file called file is inside the pool, if it is then the object can't be added and the user is informed about that, otherwise the file is added. The file and body arguments are passed by the routing method, where the file received is opened and red.
- delete_object(file_name, cluster, pool): this method open the pool and delete the file called file_name from it if it exists, otherwise an exception is raised
- get_cluster_state(): method that open the pool and return the status of the cluster.