Assignment 04: Distributed File systems

University of Puerto Rico at Rio Piedras

Department of Computer Science

CCOM4017: Operating Systems

Introduction

In this project the student will implement the main components of a file system by implementing a simple, yet functional, distributed file system (DFS). The project will expand students' knowledge of the main components of a file system (inodes, and data blocks), will further develop the student skills in inter-process communication, and will increase their system security awareness.

The components to implement are:

- **Metadata server**, which will function as an inodes repository
- **Data servers**, that will serve as the disk space for file data blocks
- **List client**, that will list the files available in the DFS
- **Copy client**, that will copy files from and to the DFS

Objectives

- Study the main components of a distributed file system
- Get familiarized with File Management
- Implementation of a distributed system

Prerequisites

- Python:
    - www.python.org
- Python SocketServer library: for **TCP** socket communication.

    - https://docs.python.org/3/library/socketserver.html
- uuid: to generate unique IDs for the data blocks
    - https://docs.python.org/3/library/uuid.html
- **Optionally** you may read about the json and sqlite3 libraries used in the skeleton of the program.
    - https://docs.python.org/3/library/json.html
    - https://docs.python.org/3/library/sqlite3.html

The metadata server's database manipulation functions.

No expertise in database management is required to accomplish this project. However sqlite3 is used to store the file inodes in the metadata server. You don't need to understand the functions but you need to read the documentation of the functions that interact with the database. The metadata server database functions are defined in file `mds_db.py`.

Inode

For this implementation an **inode** consists of:

- File name
- File size
- List of blocks

Block List

The **block list** consists of a list of:

- data node address - to know the data node the block is stored
- data node port - to know the service port of the data node
- data node block_id - the id assigned to the block

Functions:

- `AddDataNode(address, port)`: Adds new data node to the metadata server Receives IP address and port. I.E. the information to connect to the data node.
- `GetDataNodes()`: Returns a list of data node tuples **(address, port)** registered. Useful to know to which data nodes the data blocks can be sent.
- `InsertFile(filename, fsize)`: Insert a filename with its file size into the database.
- `GetFiles()`: Returns a list of the attributes of the files stored in the DFS. (addr, file size)

- `AddBlockToInode(filename, blocks)`: Add the list of data blocks information of a file. The data block information consists of (address, port, block_id)
- `GetFileInode(filename)`: Returns the file size, and the list of data block information of a file. (fsize, block_list)

The packet manipulation functions:

The packet library is designed to serialize the communication data using the json library. No expertise with json is required to accomplish this assignment. These functions were developed to ease the packet generation process of the project. The packet library is defined in file `Packet.py`.

In this project all packet objects have a packet type among the following `command type` options:

- `reg`: to register a data node
- `list`: to ask for a list of files
- `put`: to put a files in the DFS
- `get`: to get files from the DFS
- `dblks`: to add the data block ids to the files.

Functions:

General Functions

- `getEncodedPacket()`: returns a serialized packet ready to send through the network. First you need to build the packets. See Build**<X>**Packet functions.
- `DecodePacket(packet)`: Receives a serialized message and turns it into a packet object.
- `getCommand()`: Returns the `command type` of the packet

Packet Registration Functions

- `BuildRegPacket(addr, port)`: Builds a registration packet.
- `getAddr()`: Returns the IP address of a server. Useful for registration packets
- `getPort()`: Returns the Port number of a server. Useful for registration packets

Packet List Functions

- `BuildListPacket()`: Builds a list packet for file listing

- `BuildListResponse(filelist)`: Builds a list response packet with the list of files.
- `getFileArray()`: Returns a list of files

Get Packet Functions

- `BuildGetPacket(fname)`: Builds a get packet to get a file name.
- `BuildGetResponse(metalist, fsize)`: Builds a list of data node servers with the blocks of a file, and the file size.
- `getFileName()`: Returns the file name in a packet.
- `getDataNodes()`: Returns a list of data servers.

Put Packet Functions (Put Blocks)

- `BuildPutPacket(fname, size)`: Builds a put packet to put fname and file size in the metadata server.
- `getFileInfo()`: Returns the file info in a packet.
- `BuildPutResponse(metalist)`: Builds a list of data node servers where the data blocks of a file can be stored. I.E a list of available data servers.
- `BuildDataBlockPacket(fname, block_list)`: Builds a data block packet. Contains the file name and the list of blocks for the file. See block list to review the content of a block list.
- `getDataBlocks()`: Returns a list of data blocks

Get Data block Functions (Get Blocks)

- `BuildGetDataBlockPacket(blockid)`: Builds a get data block packet. Usefull when requesting a data block from a data node.
- `getBlockID()`: Returns the block_id from a packet.

Instructions

Write and complete code for an unreliable and insecure distributed file server following the specifications below.

Design specifications.

For this project you will design and complete a distributed file system. You will write a DFS with tools to list the files, and to copy files from and to the DFS.

Your DFS will consist of:

- `A metadata server`: which will contain the metadata (inode) information of the files in your file system. It will also keep a registry of the data servers that are connected to the DFS.
- `Data nodes`: The data nodes will contain chunks (some blocks) of the file that you are storing in the DFS.
- `List command`: A command to list the files stored in the DFS.
- `Copy command`: A command that will copy files from and to the DFS.

The metadata server

The metadata server contains the metadata (inode) information of the files in your file system. It will also keep a registry of the data servers that are connected to the DFS.

Your metadata server must provide the following services:

1. Listen to the data nodes that are part of the DFS. Every time a new data node registers to the DFS the metadata server must keep the contact information of that data node. This is (IP Address, Listening Port).
   - To ease the implementation of the DFS, the directory file system must contain three things:
     - the path of the file in the file system (filename)
     - the nodes that contain the data blocks of the files
     - the file size
2. Every time a client (commands list or copy) contacts the metadata server for:
   - `get`: requesting to read a file: the metadata server must check if the file is in the DFS database, and if it is, it must return the nodes with the blocks_ids that contain the file.
   - `put`: requesting to write a file: the metadata server must:
     - insert in the database the path of the new file (with its name), and its size.
     - return a list of available data nodes where to write the chunks of the file
     - `dblks`: then store the data blocks that have the information of the data nodes and the block ids of the file.
   - `list`: requesting to list files:
     - the metadata server must return a list with the files in the DFS and their size.

The metadata server must be run:

```
python meta-data.py <port, default=8000>
```

If no port is specified the port 8000 will be used by default.

The data node server

The data node is the process that receives and saves the data blocks of the files. It must first register with the metadata server as soon as it starts its execution. The data node receives the data from the clients when the client wants to write a file, and returns the data when the client wants to read a file.

Your data node must provide the following services:

1. `put`: Listen to writes:
    - The data node will receive blocks of data, store them using an unique id, and return the unique id.
    - Each node must have its own block storage path. You may run more than one data node per system.
2. `get`: Listen to reads
    - The data node will receive requests for data blocks, and it must read the data block, and return its content.

The data nodes must be run:

```
python data-node.py <server address> <port> <data path> <metadata port,default=8000>
```

Server address is the metadata server address, port is the data-node port number, data path is a path to a directory to store the data blocks, and metadata port is the optional metadata port if it was run in a different port other than the default port.

Note: Since you most probably do not have many different computers at your disposal, you may run more than one data-node in the same computer but the listening port and their data block directory must be different.

The list client

The list client just sends a list request to the metadata server and then waits for a list of file names with their size.

The output must look like:

```
/home/cheo/asig.cpp 30 bytes
/home/hola.txt 200 bytes
/home/saludos.dat 2000 bytes
```

The list client must be run:

```
python ls.py <server>:<port, default=8000>
```

Where server is the metadata server IP and port is the metadata server port. If the default port is not indicated the default port is 8000 and no ':' character is necessary.

The copy client

The copy client is more complicated than the list client. It is in charge of copying the files from and to the DFS.

The copy client must:

1. Write files in the DFS
    - The client must send to the metadata server the file name and size of the file to write.
    - Wait for the metadata server response with the list of available data nodes.
    - Send the data blocks to each data node.
        - You may decide to divide the file over the number of data servers.
        - You may divide the file into X size blocks and send it to the data servers in round robin.
2. Read files from the DFS
    - Contact the metadata server with the file name to read.
    - Wait for the block list with the bloc id and data server information
    - Retrieve the file blocks from the data servers.
        - This part will depend on the division algorithm used in step (1).

The copy client must be run:

Copy from DFS:

```
python copy.py <server>:<port>:<dfs file path> <destination file>
```

To DFS:

```
python copy.py <source file> <server>:<port>:<dfs file path>
```

Where server is the metadata server IP address, and port is the metadata server port.

Creating an empty database

The script createdb.py generates an empty database *dfs.db* for the project.

```
    python createdb.py
```

Deliverables

- The source code of the programs (well documented)
- A README file with:
    - description of the programs, including a brief description of how they work.
    - who helped you or discussed issues with you to finish the program.
- Video description of the project with implementation details. Any doubt please consult the professor.

Rubric

- (10 pts) the programs run
- (80 pts) quality of the working solutions
    - (20 pts) Metadata server implemented correctly
    - (25 pts) Data server implemented correctly
    - (10 pts) List client implemented correctly
    - (25 pts) Copy client implemented correctly
- (10 pts) quality of the README
    - (10 pts) description of the programs with their description.

- No project will be graded without submission of the video explaining how the project was implemented.