Internet Programming

Programming Assignment 3: A distributed dropbox-like file system service.

Deadline: 17-Oct-2018 23:59:59

1 Introduction

In this assignment you have to create a distributed file service in which clients can upload and retrieve files. The system consists of a number of servers that store files and a registry server that is used for coordination. A single copy of each file is stored on the appropriate server and clients can request uploads/downloads from any server, regardless of whether the server has the file in its directory or not. Files are retrieved by filename and integrity is maintained by hashing the files' name and contents. Each component will be implemented as a separate program. The entire system will run locally on your machine. In the next sections we will describe each component in more detail as well as expected behaviors and constraints.

2 The Registry [1.5 points]

The registry is a special server providing port mapping services to the file server(s). That is, when a file server wants to find out the address and port of port another file server is listening at, it should query the registry. There is always **exactly one** registry server and is expected to be fired up before any file server.

The registry program should be started with `./registry <PORT> <N>` where PORT is the port it is listening to and N the number of file servers in the system. Before N file servers are connected, no file server should listen for client requests. Once N servers are registered, the registry notifies them all to start accepting client requests. There are only two kinds of messages the registry recognizes:

`REGISTER`. This message is sent by file servers to register themselves. The format of the message is **10:PORT** where PORT is the port number the file server will be listening to. Note that the colon is only used for demonstration purposes, i.e., the port value should immediately follow the value 10. You may assume the same for the remaining of this document. The registry should respond with an `R_OK` message that has the format **20:N:ID** where N is the total number of file servers and ID is a unique identifier assigned by the registry to that file server (you will need these later). For simplicity you can keep IDs in the range $[0, N-1]$. A server's ID cannot change.

`QUERY`. This kind of message is sent by file servers to ask for the address and port of some other file server and has the format **11:ID** where ID is the id of the server of interest. If a server with such id exists the registry replies with a `Q_OK` message **21:IP:PORT**. If not, then it replies with a `Q_ERROR` message **22**.

Any other message can be safely discarded. Once N files servers have registered the registry server broadcasts a `START` message with the value **40** to all file servers.

3 File Servers [5 points]

The file server program should be started with `./server <PORT> <REG_IP> <REG_PORT> <DIR>` where `PORT` is the port it is listening at, `REG_IP` and `REG_PORT` the ip address and port of the registry server respectively, and `DIR` the directory it stores its files to. It should be possible to spawn multiple (N) server processes.

The first thing a file server does is register with the registry server, as mentioned in the previous section, and then wait for the corresponding `START` message to start accepting client requests. You

may assume that this message will always arrive, although you may want to set a timeout for testing purposes throughout development. A file server must be able to handle multiple requests concurrently.

A file may only exist once in the entire file system (throughout servers). You have to come up with a protocol for mapping a file to a unique file server. You may use a hash of the file's name for it as well as the knowledge of N (sent by the registry as a response to the `REGISTER` message). A client should be able to connect to any server and request a file upload. This request should always succeed regardless of the file server that the file maps to. If the requested file maps to a different file server, the current server is responsible for making sure the file ends up in the right server. A possible solution to this is having the current server act as a client for the purposes of uploading the file to the appropriate server, after sending a `QUERY` message to the registry to find out its address and port. A file server recognizes the `R_OK`, `Q_OK`, `R_ERR` and `START` messages from the previous section as well as :

`PUT`. This message is sent by clients to upload a file and has the format **12:FILENAME:FILE**. **FILENAME** should be treated as a null terminated string and anything following it should be considered as the file contents. If the file maps to another server the current server handles it transparently as mentioned in the previous paragraph. If the file maps to the current server, then a file with the appropriate filename and contents should be created under the directory specified by the DIR command line argument. If a file with this name already exists, its contents should be updated. If something went wrong the server should respond with a `P_ERR` **24** message. If the file has been stored or updated (locally or at some other server), the server should respond with a `P_OK` message that has the format **23:hash(FILENAME):hash(FILE)**.

`GET`. This message is sent by clients to retrieve a file and has the format **13:FILENAME**. If the file maps to that server, the server should respond with a `G_OK` message that has the format **25:hash(FILENAME):hash(FILE):FILE** if the file is actually present, otherwise with a `G_UNKNOWN` **28**. If the file belongs to another server, it should respond with a `REDIDERECT` message **26:IP:PORT**. Note that the server may first calculate the ID the file maps to and then consult the registry using a `QUERY` message. In the case of an error a `G_ERR` **27** is returned.

`DELETE`. This message is sent by clients wishing to delete a file and its format is **14:FILENAME**. If the file is not present locally, the server should transparently contract the appropriate server similarly to how it's down for `PUT`. If the delete succeeds the server should respond with a `D_OK` message **29**. Otherwise, it should send back a `D_ERR` **30** message.

Any other message can be safely discarded without a response. For hashing the name as well as the contents of a file you may use any hash function, e.g, `SHA256`, `MD160`, **djb2** see listing 2.

4 Client [2.5 points]

The client should be run with `./client <OP> <FILENAME>` where OP $\in$ { `put`, `get`, `delete` } is the operation to be performed and FILENAME is the path to the relevant file. The client recognizes all the response messages from the previous section and can discard anything else. When a `G_REDIRECT` message is received, the client stab handles it transparently, meaning that it should repeat the request, now to the appropriate server, without informing the user. When any kind of `ERR` message is received the client can print a message and exit. Correctly received files should be stored in the current directory and the client should exit after printing an appropriate message. A file is considered to be correctly received if and only if:

- hash(requested_filename) == received_filename_hash, and
- hash(received_file_contents) == received_file_contents_hash

In any other case the client prints an appropriate message and exits. Please use short (up to 80 characters), 1-line messages for the client in order to simplify our testing.

5 Report [1 point]

You have to submit a report giving explanations to the following:

- Your design choices for every component of the system.
- Your protocol for mapping files to servers. We expected a detailed description of this in order to understand (and evaluate) your implementation.
- Why (or why not) your implementation is efficient when writing files to another server.
- Problems you faced and how you solved them.
- If some components is unfinished/not working/exhibits weird behaviour, describe what the problem is and how a possible solution might look like.

If you are also implementing a bonus, you have to completely describe the protocol followed as well as what modifications you had to do to the core assignment and why.

6 Bonus 1. Dynamically adding file servers. [1 point]

For this bonus you have allow file server to join the system while the system is running. A new server joining the system has to acquire an ID from the registry, and increases N by one. The latter may interfere with the mappings of existing files to file servers, and files may need to be relocated. You have to come up with a solution that ensures correctness in the sense that any file that was retrievable before the new join, should also be retrievable afterwards (perhaps from a different now). You can support only 1 server joining at a time, meaning that while this procedure takes place it is acceptable to block other servers from joining. You may also block client requests too. You are completely free to decide what protocol should be followed and you are allowed to add additional messages if needed and/or modify the command line interfaces.

7 Bonus 2. Dynamic Load Balancing. [1 point]

For this bonus you have to maintain load balancing. That is, the following should hold at any given time:

- Let N be the number of servers and M be the total number of files in the system (throughout all servers). As long as $M \geq N$ no server should hold 0 files.
- Let M_i be the number of files stored at file server i and M_{max} the maximum number of files at any particular server. Then it should hold that $\forall_i, 0.5 \times \frac{M_{max}}{M_i} \leq 1$. That is, no file server should store more than twice the amount of files than any other server.

Implementing this bonus will have a direct impact on the mapping of files to file servers and may require a more advanced and dynamic approach, especially in combination with **Bonus 1**. Again you are free to decide on the protocol to follow, add extra messages if needed and/or modify command line interfaces.

8 Requirements and Constraints

- The assignment should be implemented in the C language.
- You are not allowed to use any external libraries.
- You should submit (at least) 3 files: registry.c, server.c and client.c
- You should include a Makefile that compiles everything correctly. Submissions without a Makefile will not be considered.
- Both file servers and the registry must be able to handle multiple requests concurrently.
- You may use any type of server (i.e., on demand forking, multi-threaded, etc.)

- Make sure you handle race condition correctly, You may use any synchronization primitive through any API you want.
- All communication should be done with sockets and TCP.
- The service should run (and will be tested) locally but it should be able to work on multiple machines too. No distinction should be made.
- You may assume no failures may exist, i.e., servers crashing, network partitions and so on.
- You may assume that we won't push your implementation to the limits upon testing, i.e, using very large values of N, DoS-ing file servers, applying extreme corner case scenarios and so on.
- Submission should done through Canvas

9 Grading

Your final grade for the assignment is the sum of the partial grades for each components and the report. Implementing a bonus can indeed get you a grade higher than 10. The excess points will contribute towards your final grade for the course. Your final grade for the course cannot be higher than 10.

10 Appendix

Listing 1: Summary of the messages, their codes and expected formats.

Message	**Code (8-bits)**	**Format**
REGISTER	10	10:PORT
R_OK	20	20:N:ID
QUERY	11	11:ID
Q_OK	21	21:IP:PORT
Q_ERR	22	22
START	40	40
PUT	12	12:FILENAME:FILE
P_OK	23	23:hash(FILENAME):hash(FILE)
P_ERR	24	24
GET	13	13:FILENAME
G_OK	25	25:hash(FILENAME):hash(FILE):FILE
G_UNKNOWN	28	28
G_REDIRECT	26	26:IP:PORT
G_ERR	27	27
DELETE	14	14:FILENAME
D_OK	29	29
D_ERR	30	30

Listing 2: djb2 hash

```
unsigned long
djb2_hash(unsigned char *str)
{
  unsigned long hash = 5381;
  int c;

  while (c = *str++)
    hash = ((hash << 5) + hash) + c; /* hash * 33 + c */

  return hash;
}
```

Good luck!