Advanced Computer Communication (CNCO3002)

CURTIN UNIVERSITY
School of Electrical Engineering, Computing and Mathematical Sciences
Discipline of Computing

Assignment 1: Anonymous Storage Server.

Due Date: **4.00 p.m. Monday, 14 September 2020**

The objective of this network programming assignment is to develop two complementary programs, a ***client*** and a ***server***, that implement an anonymous storage server.

The ***server*** stores user files anonymously. A ***user***, via ***client***, sends a file to the ***server***. The ***server*** stores the file in a folder, and returns a hash **key** to the ***client***. The user can retrieve the stored file using the **key**. The **key** is sufficiently long and complex to guarantee that nobody, except the user, can guess the **key** of each file.

A **key** is generated as follows. When the ***server*** receives a file, it creates a new name, i.e., **file_*n*** for the file, generates a hash **key** for the **file_*n***, stores the file in a folder **storage**, and returns the **key** to the ***client***. Note, the ***server*** can use a global counter to create a unique file name, i.e., the value of ***n*** is positive integer {1, 2, 3, … }. The **key** is an MD5 hash, which is a 128-bits value usually printed as 32 hexadecimal numbers created by Unix function **md5sum**. You may use **popen()** function to call **md5sum** in your program, and **pclose()** function to close the created stream.

The ***client*** and ***server*** recognize the following five ***user*** commands. Each command is **not case sensitive**.

1) **STORE local_file**: A ***user*** asks the ***server*** to store a file **local_file** that is located at ***client***.

   The ***client*** sends command **STORE** and the content of file **local_file** to the ***server***, waits for reply from the ***server***, i.e., the file's **key**, shows the reply on the screen, and waits for another ***user*** command. Note that the name of the file, i.e., **local_file**, is not sent to the ***server***.

   Receiving this request, the ***server*** creates a unique file name, e.g., **file_100**, generates the hash **key** for **file_100**, writes the content of **local_file** in **file_100,** which is stored in a directory **storage**, records this transaction in the file's **history**, sends a message "STORE: File has been stored with hash **key**", and waits for another ***client*** request.

2) **GET key file_name**: A ***user*** wants to retrieve from the ***server*** a file whose hash is **key**, and writes its content in **file_name** located at ***client***.

The ***client*** sends this request to the ***server***, waits for the file sent by the ***server***, stores its content in a file **file_name** in ***client***'s directory, and waits for another ***user*** command. Note that **file_name** is not sent to the ***server***.

Receiving this request, the ***server*** sends the file whose hash is **key** to the ***client***, records this transaction in the file's **history**, and waits for another ***client*** request.

If the received **key** does not correspond to hash of any file on the ***server***, the ***server*** sends a message "GET: Error! Hash **key** is not valid" and waits for another client request.

A ***client*** is given k chances to send a valid **key**. The k^{th} time a ***client*** provides an invalid **key**, the ***client*** is disconnected and no new connection will be accepted from the same IP address within t_1 seconds. Let $t_1 \geq 1$ and $k \geq 1$ be command line arguments when you run ***server***.

3) **DELETE key**: A ***user*** wants to delete a file whose hash is **key** from the ***server***.

The ***client*** sends this request to the ***server***, waits for a reply, shows the reply on the screen, and waits for another ***user*** command.

Receiving this request, the ***server*** deletes a file from folder **storage** whose hash is **key**, records this transaction to the file's **history**, and replies with "DELETE: File with hash **key** has been deleted", and waits for another ***client*** request.

If the received **key** does not correspond to the hash of any file on the ***server***, the ***server*** sends a message "DELETE: Error! Hash **key** is not valid" and waits for another client request.

A ***client*** is given k chances to send a valid **key**. The k^{th} time a ***client*** provides an invalid **key**, the ***client*** is disconnected and no new connection will be accepted from the same IP address within t_1 seconds. Let $t_1 \geq 1$ and $k \geq 1$ be command line arguments when you run ***server***.

4) **HISTORY key**: A ***user*** wants to read the recorded **history** of the file whose hash is **key**.

The ***client*** sends the request to the ***server***, waits for a reply, shows the reply on the screen, and waits for another user command.

Receiving this request, the ***server*** sends to the ***client*** the history records for **key**, and waits for another ***client*** request.

If there is no entry for **key**, the ***server*** sends a message "HISTORY: Error! There is no history for hash **key**" and wait for another client request.

A ***client*** is given k chances to send a valid **key**. The k^{th} time a ***client*** provides an invalid **key**, the ***client*** is disconnected and no new connection will be accepted from the same IP

address within t_1 seconds. Let $t_1 \geq 1$ and $k \geq 1$ be command line arguments when you run ***server***.

5) **QUIT**: A ***user*** wants to disconnect from the ***server***.

The ***client*** sends the request to the ***server***, waits for an acknowledgement, and closes the connection.

Receiving this request, the ***server*** sends a message "Thank you for using our anonymous storage", and closes the connection.

A. Requirements

1. A ***client*** should be started with:

./*client* IP_address port

or

./*client* host_name port

Note, **IP_address** is the ***server***'s IP address (in dotted decimal notation), **host_name** is the name of the ***server***'s host computer, and **port** is the port number of the ***server***. For the port number, use the **first** of the ten ephemeral port numbers assigned to you.

Once a ***client*** is connected to the ***server***, the ***client*** waits for ***user*** command, and a prompt **ID>** is shown on the screen, where **ID** is your student ID.

2. A ***server*** creates a new process for each connecting ***client***.

3. A ***server*** disconnects a ***client*** if it does not receive any request from the ***client*** within t_2 seconds. Let $t_2 \geq 1$ be a command line argument when you run the ***server***. Thus, the ***server*** is run as:

./server k t_1 t_2

4. A ***server*** maintains a **history** of each file it stores. This **history** is kept in memory and is updated for every operation on the file. The **history** of a file contains the following information: the file's **key**, time of each operation, type of each operation, i.e., **store**, **get**, **delete**, **history**, and the IP address of the ***client*** that requests for each operation.

Note: the **history** of each file is shared among all child servers, and thus a shared memory should be created, and a proper synchronization/mutual exclusion handling is required.

5. A ***client*** must check the syntax and validity of each ***user*** command. Similarly, each child ***server*** must check the validity and syntax of its ***client***'s request.

B. Implementation

1. Your program must be written in C, and must run on our department's laboratory computers.

2. You are allowed to use any existing functions/codes. However, you are responsible for the correctness of the functions, and you have to mention/cite the sources.

3. Make sure to check for error return from each system call, and to close every socket created.

4. Your main program for the ***server*** should be in **server.c**, and that for your ***client*** program should be in **client.c**. You must provide files **makeserver** and **makeclient** for the ***server*** and ***client***, respectively.

5. While developing your program, make sure that you kill all your (possibly run-away) processes using ***'kill –9'***, and release any unused resources, e.g., shared memory.

6. You MAY make your own justifiable assumptions/requirements other than those already given. However, YOU HAVE TO DOCUMENT ANY ADDITIONAL ASSUMPTIONS/LIMITATIONS FOR YOUR IMPLEMENTATIONS.

C. Instruction for submission and demo

1. Assignment submission is **compulsory**. Students will be penalized by a deduction of ten percent per calendar day for a late submission. **An assessment more than seven calendar days overdue will not be marked and will receive a mark of 0**.

   Due dates and other arrangements may only be altered with the consent of the majority of the students enrolled in the unit and with the consent of the lecturer.

2. You must (i) put your program files, i.e., client.c, server.c, makeclient, makeserver, and other required files, in your home directory named ACC/assignment1, (ii) submit the soft copy of assignment **report** to the unit Blackboard **(in one zip file),** i.e., **ID_Assignment1.zip**, where ID is your student ID.

3. The soft copy of your **report**, as stated in 2(ii), MUST include:

   - A signed cover page that explicitly states the submitted assignment is **your own work**. The cover includes the words "Advanced Computer Communication Assignment 1", and your name in the form: family, other names. Your name should be as recorded in the student database.

   - Software solution of your assignment that MUST include (i) **all source code** for the programs with proper in-line and header documentation. Use proper indentation so that your code can be easily read. Make sure that you use meaningful variable names, and delete all unnecessary comments that you created while debugging your program;

and (ii) **readme file** that, among others, explains how to compile your program and how to run the program.

- Detailed discussion on how any mutual exclusion is achieved on shared resources, e.g., memory, and what processes access the shared resources.

- Description of any cases for which your program is not working correctly or how you test your program that make you believe it works perfectly.

- Sample inputs and outputs from running your programs.

Your report will be assessed (worth 20% of the overall mark for Assignment 1).

4. Demo requirements:

- You will be required to demonstrate and explained your programs.
- You MUST keep the source code for your programs in your home directory, and the source code MUST be that submitted.
- The programs MUST run on our Department's computer system.

Failure to meet these requirements may result in the assignment not being marked.