CSE 5306 **Lab # 2** **Fall – 2019**

Distributed File Service with Consistency

Objectives:

1. An introduction to file consistency.
2. Exposure to Push/Pull/Invalidation Notices.
3. Further experience with multithreading.
4. Exposure to real-time file monitoring.

Due: October 18th, 2019 before 11:58pm via Canvas

Project Specification:

These will be individual projects. You may write the program in any language that is supported under any Integrated Development Environment (IDE). Keep in mind that available controls, objects, libraries, et cetera, may make some of these tasks easier in one language than in another. Finally, because of the lack of restrictions on IDEs, you will have to have that IDE available to demo to the TA (e.g., you will demo the program on your own laptop).

This lab is intended to be built on top of Lab 1. If you successfully completed Lab 1, most of the foundation for this program should already be in place. An explanation of the mechanics of updates may be found in §7.4.3 of Distributed Systems: Principles and Paradigms, 2/ed.

You will implement a distributed file service consisting of a server process and three clients. Each client process will connect to the server over a socket. The server should be able to handle all three clients concurrently.

Clients will designate a directory on their system to serve as their shared directory. A single, identical .txt file will be pre-loaded into this directory (e.g., the file will be in place at startup). A simple text editing program, such as Notepad, will be used to modify the contents of that .txt file on any individual client. When one copy of the .txt file disagrees with the other copies, the system is said to be inconsistent.

The client will automatically detect that the file has been updated and `Push` those updates to the file server. When the server receives a file update from a client, it will issue `Invalidation Notices` to the remaining clients.

Upon receipt of an `Invalidation Notice`, a client will `Pull` the updated file from the server and update their local copy. When all copies of the shared .txt file agree with one another, the system is said to be consistent.

Your program will repeat this update process, beginning with any individual client, as many times as necessary until the program is killed. File updates may be handled according to developer discretion. For example, the updated file may be exchanged in its entirety, or only the updated portions of the text may be exchanged.

Clients will prompt the user for a username. When a client connects to the server, its username should be displayed by the server in real time. Two or more clients may not have the same username. Should

the server detect a conflict in username, the client's connection should be rejected, and the client's user should be prompted to input a different name.

All components should be managed with a *simple* GUI. The GUI should provide a way to kill the process without using the 'exit' button on the window.

The required actions are summarized below.

Client

Upon startup, the client will perform the following actions:

Startup:

1. Prompt the user to input a username.
2. Connect to the server over a socket and register the username.
    a. When the client is connected, the user should be notified of the active connection.
    b. If the provided username is already in use, the client should disconnect and prompt the user to input another username.
3. Proceed to manage the consistency of the shared file until killed by the user.

Updating Files:

1. The client will actively monitor the designated text file.
2. When an update to the file is noted, the client will notify the user of the update.
3. The client will then `Push` the updated file to the server. The user should be notified when the upload is complete.
4. Return to step 1.

Retrieving Updates:

1. The client will listen for an `Invalidation Notice` from the server.
2. Upon receipt of the notice, the client will `Pull` the update and apply the update to its copy of the shared .txt file.
3. The client will inform the user that an update has been completed.
4. Return to step 1.

Notes:

- The shared directory may be hardcoded or assigned at run time, according to the developer's preference.
- The files will **not** be updated concurrently (meaning only one client will update the file at a time).
- Any basic text editing program may be used to update the .txt file.
- All three clients and the server may run on the same machine. Directories may be managed according to the developer's preferences.

Server

The server should support three concurrently connected clients and display a list of which clients are connected in real-time. The server will execute the following sequence of steps:

1. Startup and listen for incoming connections.

2. Print that a client has connected, and:
    a. If the client has an available username, fork a thread to handle that client. Or,
    b. If the username is taken, reject the connection from that client.
3. When a client pushes an update to the server, the server should:
    a. Accept the data from the client;
    b. Print that an update has been received and from which client;
    c. Issue `Invalidation Notices` to the remaining clients; and,
    d. Allow the remaining clients to `Pull` the update.
4. Begin at step 2 until the process is killed by the user.

Notes:

- It is not necessary for the server to store a copy of the update after clients are finished retrieving it.
- The server must correctly handle an unexpected client disconnection without crashing.
- When a client disconnects from the server, the server GUI must indicate this to the user in real time.

Your program must operate independently of a browser.

Citations:

You may use open source code found on the Internet in your program. When citing this code:

YOU MUST CITE THE EXACT URL TO THE CODE IN THE METHOD / FUNCTION / SUBROUTINE HEADER WHERE THE CODE IS UTILIZED.

Failure to cite the exact URL will result in a twenty (20) point deduction on the grade of your lab.

A full list of your source URLs should be included in your writeup file. Including generic citations (for instance, simple listing "StackOverlow.com" without additional details) will result in a ten (10) point deduction in your lab grade, per instance.

Submission Guidelines:

FAILURE TO FOLLOW ANY OF THESE DIRECTIONS WILL RESULT IN DEDUCTION OF SCORES.

Submit your assignment via the submission link on Blackboard. You should zip your source files and other necessary items like project definitions, classes, special controls, DLLs, etc. and your writeup into a single zip file. No other format other than zip will be accepted. The name of this file should be your **lastname_loginID.zip**. Example: If your name is John Doe and your login ID is jxd1234, your submission file name must be "Doe_jxd1234.zip".

Be sure that you include everything necessary to unzip this file on another machine and compile and run it. This might include forms, modules, classes, config. files, etc. DO NOT INCLUDE ANY RUNNABLE

EXECUTABLE (binary) program. The first two lines of any file you submit must contain your name and student ID. Include it as comments if it is a code file.

You may resubmit the project at any time. Late submissions will be accepted at a penalty of 10 points per day. This penalty will apply regardless of whether you have other excuses. In other words, it may pay you to submit this project early. If the TA cannot run your program based on the information in your writeup then he or she will email you to schedule a demo. The TA may optionally decide to require all students to demonstrate their labs. In that case we will announce it to the class. It is your responsibility to keep checking blackboard for announcements, if any.

If your program is not working by the deadline, send it anyway and review it with the TA for partial credit. Do not take a zero or excessive late penalties just because it isn't working yet. We will make an effort to grade you on the work you have done.

Writeup:

Your write-up should include instructions on how to compile and run your program. Ideally it should be complete enough that the TA can test your program without your being there. Your writeup should include any known bugs and limitations in your programs. If you made any assumptions, you should document what you decided and why. This writeup can be in a **docx** or **pdf** format and should be submitted along with your code.

Grading:

Points – element

20 – Client Process works correctly.
20 – Server Process works correctly.
05 – Client provides username to server.
05 – Server rejects duplicate usernames.
15 – Clients correctly pushes updates to server.
10 – Clients process invalidation notices correctly.
10 – Clients pull the update correctly.
05 – Client and server handle disconnections correctly.
05 – Server displays connected clients in real time.
05 – Comments in code.

Deductions for failing to follow directions:

-10 – Late submission per day.
-05 – Including absolute/ binary/ executable module in submission.
-02 – Submitted file doesn’t have student name and student ID in the first two lines.
-05 – Submitted file has a name other than student's lastname_loginID.zip.
-05 – Submission is not in zip format.
-05 – Submitting a complete installation of the Java Virtual Machine.
-10 – Per instance of superfluous citation.
-20 – Server and/or clients run exclusively from a command line.

Distributed File Service with Consistency

To receive full credit for comments in the code you should have headers at the start of every module / function / subroutine explaining the inputs, outputs, and purpose of the module. You should have a comment on every data item explaining what it is about. (Almost) every line of code should have a comment explaining what is going on. A comment such as /* Add 1 to counter */ will not be sufficient. The comment should explain what is being counted.

Important Note:

You may discuss the problem definition and tools with other students. You may discuss the lab requirements. You may discuss or share project designs. All coding work must be your own. You may use any book, WWW reference or other people's programs (but not those of other students in the class) as a reference as long as you cite that reference in the comments. If we detect that portions of your program match portions of any other student's program, it will be presumed that you have colluded unless you both cite a specific source for the code.

You must not violate University of Texas at Arlington regulations, laws of the State of Texas or the United States, or professional ethics. Any violations, however small, will not be tolerated.

DO NOT POST YOUR CODE ON PUBLICLY ACCESSIBLE SECTIONS OF WEBSITES UNTIL AFTER THE COURSE CONCLUDES. SHOULD YOU DO SO, THIS WILL BE CONSIDERED COLLUSION, AND YOU WILL BE REFERRED TO THE OFFICE OF STUDENT CONDUCT AND RECEIVE A FAILING GRADE IN THE COURSE