CSE310

Course project: Interest Groups

Using the Internet domain sockets, implement a network application that supports interest-based discussion groups.

Your implementation consists of a client program and a server program. The client program allows an application user to login to the application, browse existing discussion groups, subscribe to those groups that are of interest, and read and write posts in a subscribed group. The server program allows a set of servers to maintain all the discussion groups, update the posts in each group, and interact with clients to support user activities. You will design all communication protocols involved as well as other supporting elements such as how discussion groups and user posts are formatted, stored, and accessed and how user history is formatted and maintained.

You are to develop this project in groups of 3 students.

Building a single-server system

All discussion groups are hosted at a single server. All users access this single server to participate in discussion groups. Each user has a unique user ID. Each discussion group has a unique group ID and a unique group name. Each user post has a unique post ID, a subject line, and a content body. Each post is also associated with the user ID of the post author as well as a time stamp denoting when the post is submitted.

The server is started first and waits at a known port for requests from clients. The port number that the server listens at can be hard-coded, or can be output to the standard output from your program and used when starting client programs. The client program takes two command line arguments: (i) the name of the machine on which the server program is running, (ii) the port number that the server is listening at.

Commands You are required to implement a command line interface similar to the Linux command line for the client program. Once the client program is started, say, by you, the following commands should be supported:

`login` – this command takes one argument, your user ID. It is used by the application to determine which discussion groups you have subscribed to, and for each subscribed group, which posts you have read. For simplicity, we do not prompt the user for a password and skip the authentication process.
You should maintain this user history information locally on the client machine using a file. This approach assumes that a user always accesses this application on the same machine. It reduces the number of messages exchanged between the client and the server, and may allow the server to more efficiently support a large number of clients.

`help` – this command takes no argument. It prints a list of supported commands and sub-commands. For each command or sub-command, a brief description of its function and the syntax of usage are displayed.

Once a user is logged in, the following commands should be supported:

`ag` - this command stands for "all groups". It takes an optional argument, N, and lists the names of all existing discussion groups, N groups at a time, numbered 1 to N. If N is not specified, a default value is used. Below is an example output of the command "`ag 5`". Whether or not a group is a subscribed group is indicated in parentheses. In this example, among the five groups displayed, the user is currently subscribed to groups comp.lang.python and comp.lang.javascript.

```
1. ( ) comp.programming
2. ( ) comp.os.threads
3. ( ) comp.lang.c
4. (s) comp.lang.python
5. (s) comp.lang.javascript
```

At this time, the following sub-commands should be supported:

`s` – subscribe to groups. It takes one or more numbers between 1 and N as arguments. E.g., given the output above, the user may enter "`s 1 3`" to subscribe to two more groups: comp.programming and comp.lang.c

`u` – unsubscribe. It has the same syntax as the `s` command, except that it is used to unsubscribe from one or more groups. E.g., the user can unsubscribe from group comp.lang.javascript by entering the command "`u 5`"

`n` – lists the next N discussion groups. If all groups are displayed, the program exits from the `ag` command mode

`q` – exits from the `ag` command, before finishing displaying all groups

`sg` - this command stands for "subscribed groups". It takes an optional argument, N, and lists the names of all subscribed groups, N groups at a time, numbered 1 to N. If N is not specified, a default value is used. Below is an example output of the command "`sg 5`". The number of new posts in each group is shown before the group. E.g., there are 18 new posts in group comp.programming since the user last listed this group, and there are no new posts in rec.arts.ascii

```
1.  18   comp.programming
2.   2   comp.lang.c
3.   3   comp.lang.python
4.  27   sci.crypt
5.       rec.arts.ascii
```

The same set of sub-commands as the `ag` command should be supported, except the `s` sub-command. These include the `u`, `n`, and `q` sub-commands.

`rg` - this command stands for "read group". It takes one mandatory argument, *gname*, and an optional argument N, and displays the (status – new or not, time stamp, subject line) of all posts in the group *gname*, N posts at a time. If N is not specified, a default value is used. *gname* must be a subscribed group. When displaying posts, those unread (new) posts should be displayed first. Below is an example output of the command "`rg comp.lang.python 5`"

```
1. N   Nov 12 19:34:02   Sort a Python dictionary by value
2. N   Nov 11 08:11:34   How to print to stderr in Python?
3. N   Nov 10 22:05:47   "Print" and "Input" in one line
4.     Nov  9 13:59:05   How not to display the user inputs?
5.     Nov  9 12:46:10   Declaring custom exceptions
```

A list of 5 posts are displayed. Three new posts are shown first, indicated by the letter 'N'. The following sub-commands are supported:

`[id]` – a number between 1 and N denoting the post within the list of N posts to display. The content of the specified post is shown. E.g., entering '`1`' displays the content of the post "Sort a Python dictionary by value".

While displaying the content of a post, two sub-sub-commands are used:

'`n`' – would display at most N more lines of the post content.

'`q`' – would quit displaying the post content. The list of posts before opening the post is shown again with the post just opened marked as read.

`r` – marks a post as read. It takes a number or range of number as input. E.g., '`r 1`' marks the first displayed post to be read. '`r 1-3`' marks posts #1 to #3 in the displayed list to be read.

`n` – lists the next N posts. If all posts are displayed, the program exits from the `rg` command mode

`p` – post to the group. This sub-command allows a user to compose and submit a new post to the group.

The client program prompts the user for a line denoting the post subject, and then the content of the post, until some special character sequence, such as "`\n.\n`" – a dot by itself on a line, which denotes the end of post, is entered by the user. The post is then submitted to the server. Afterwards, a new list of N posts should be displayed, including the newly submitted post which is shown as unread.

`q` – exits from the `rg` command

The format to use to display the content of a post is as follows:

```
Group: comp.lang.python
Subject: Sort a Python dictionary by value
Author: Gern Blanston
Date: Sat, Nov 12 19:34:03 EST 2016

I have a dictionary of values read from two fields in a database: a string field and a
numeric field. The string field is unique, so that is the key of the dictionary.

I can sort on the keys, but how can I sort based on the values?
```

`logout` – this command takes no argument. It logs out the current user, and terminate the client program after all proper updates are completed.

Discussion groups and posts You may assume that a set of discussion groups have already been created. This can be implemented by manually creating a set of discussion groups on the server machine, storing their information in a (set of) file on the server machine, before starting the server program. You need to design your own file format for storing discussion group information. You must have initially 15 to 20 discussion groups.

You also need to design your own way for storing all the posts for each group at the server. You may also pre-create a small number of posts (3 to 5) for each discussion group before starting the server. All discussion groups as well as all posts within each group are stored and maintained by the server. Once a new post is composed, it should be sent to the server immediately.

Concurrent server and concurrent state updates Multiple users may access the server at the same time. Your server should be able to support multiple client-server communications at the same time. This can be implemented using multi-processes, as seen on Slide 3-12, or using multi-threading, as seen on Slide 3-13.

Multiple users may update a discussion group's information at the same time. E.g., by submitting new posts to the same discussion group at about the same time. You need to make sure these updates to the same data store is properly handled so that multiple simultaneous writes are performed in serial instead.

In this project, you will need to read about concurrency implementation based on the programming language that you choose. See the section "Development" below for more information.

Timely state updates There is an important state update requirement when multiple users are subscribed to the same discussion group at the same time. When a new post is submitted by an author to this group, all other active subscribers of this group in the system should be able to see the information about this new post as quickly as possible. E.g., when a subscriber is in the `sg` mode, the next time a list of N groups is shown, an alert message should be displayed indicating a new post is added for a specified group. Similarly, if a subscriber is in the `rg` mode, the next time a list of N posts is shown, the new post should be added to the top of the list.

Thread of posts You do not need to implement a "reply to a post" function. Rather, one can submit new posts with the same subject line. These posts are considered to be in the same thread. The time stamps can be used to distinguish different posts within the thread. For simplicity, in the `rg` mode, we do not display the posts within the same thread together. Instead, all posts are displayed in the reverse-chronological order based on the post submission time stamps, regardless of their threads.

Persistence and expiration of posts A user's history information includes which groups she has subscribed to, and which posts she has read in each group. As described above, such user history information must persist (be remembered) across multiple user login sessions, instead of being lost after a login session ends.

We assume that all posts will expire after a certain period of time, say 6 months or 3 years. You do not need to implement a "delete post" function. For simplicity, you do not need to implement the post expiration either.

Communication protocols Each command defined above may invoke message exchanges between the client and the server. The protocol for the communication between the client and the server, including the types of messages, the syntax and semantics of each type of message, and the actions taken when each type of message is sent and received, needs to be designed and specified by you. You can refer to the HTTP protocol (RFC2616) and use formats similar to the HTTP request and response messages (Slides 2-26 to 2-31) for the types of messages that you define for this project. For example, you can use methods such as "LOGIN", "AG", and "SG", etc. in the method field of the message that is sent to the server. A command argument can be put in a subsequent field of the same message.

Each message sent to the server should be properly acknowledged by way of response messages. Status codes and status phrases similar to that in the HTTP response messages should be used. Some examples are 200 OK and 400 Bad Request, among others. The protocol specification that you define needs to be clearly stated in the written documentation described below.

Bonus task: Building a multi-server system

If you have finished the project and still have time, you may build a multi-server system. In such a system, each server serves a separate groups of users. Based on its user preferences and characteristics, each server selects a subset of all discussion groups out there, and only serve these selected groups to its users. As a consequence, one discussion group may be selected by multiple servers. Indeed, very popular discussion groups may be selected by all servers.

When new posts are submitted by a user in a discussion group, the server serving the user needs to "push" the new post to all other servers who have selected this group, so that users of those servers can receive up-to-date discussion posts. You need to design and define new protocols and additional components involved for the server-to-server communication. Include a description of all of them in your documentation.

In the multi-server system, all client-server interaction such as the set of commands are identical to the single-server case.

Development

You may implement the project using either Python, Java, or C. Your programs must run on the *allv* Linux environment that we used in Labs 2 and 3. Your server program should print adequate messages on the standard output, to convince the TA that it has implemented required functionalities.

Python socket programming was covered in Labs 2 and 3 as well as the textbook. For Java and C socket programming, some simple example programs are provided to you on this page. You are also free to read online tutorials.

As to *concurrency programming*. For Python, you can refer to the presentation slides and code samples at An Introduction to Python Concurrency, note that you only need to be concerned with Parts 3 and 4 in this tutorial. For Java, you can refer to Java tutorial for concurrency. For C, you may refer to Keir Davis et. al's book chapter, which contains information on concurrent server design. Additional concurrent programming pointers can be found again here.

The project demo will be held shortly after the due date. Each group is given approximately 6 min. A sign-up process will be announced and followed to determine the demo slot for each group. Every member must attend the demo. You are free to develop your code at home, but you must be able to give the project demo in the lab environment at school.

You are responsible to clean up any processes of yours that might be left around running as you develop, debug, and test your programs. Orphan processes consume extensive resources, slowing down both others' work and yours. Under Linux, to kill a process, type "`ps aux | grep your_id`", where `your_id` is your login id, to find the process id(s) of the orphan processes (in the second column of the `ps` command output), and type "`kill -9 pid`" to kill the process with process id `pid`.

Moss and academic integrity As stated in the course outline, all submitted individual work must be your own. All submitted group work must be your group's own. Your submitted code will be checked using the Stanford Moss package - *A System for Detecting Software Plagiarism*, to detect cheating. Source code from you and your classmates will be checked against each other by this package.

What to submit

Submit the following two materials separately before the due date:

(a) Well-documented source code files, together with a README file.

- Each program should have proper program documentation in it. Program documentation is the comments that appear in the source code to aid in the understanding of the program. They tell what effect the code will have or to elucidate a complex statement. Do not reiterate in English what is obvious from the code such as "variable x is incremented".

- You can **only** submit the source code. No executable or object file is accepted. This means before you submit, you must make a clean submit directory that has only the required files in it. Name all Python program files with .py suffix, all Java program files with .java suffix, and all C program files with .c suffix.

- Submit a single .tar or .zip file instead of multiple files. Include in your README file instructions on how to obtain the source program files from the archive or compressed file. This must be able to be performed in the lab environment as well.
- If you implemented the bonus task, you may choose to submit one code that has both bonus and non-bonus functions, or submit two versions, one for the base and one for when the bonus part is added.

(b) Written project documentation in a single file that includes the following sections:

- Overview: indicate how much of the project you have completed. A brief description of the major functionality in each part that you have implemented.
- User documentation: it contains everything that the user of your program (and the grader) needs to know to properly use the program to obtain desired results. This section does not describe how the program works, but only what it does and how to use it. You should describe the syntax and parameters used to run your programs and the program output (including possible error messages).
- System documentation: system documentation is for people who need to know what is going on inside the program so they can perhaps change it or add new features. It describes all design decisions; the communication protocol(s) that you defined for this application; how concurrent operations are handled; all major data structures and file formats; all program limitations such as the maximum #clients at a time and the maximum post size; and how to compile and generate the executables. Try to be complete but concise.
- Testing documentation: include a list of testing scenarios to convince the grader that the program works. Explain what expected output and actual output are.
- The written documentation should be type-set. Only pdf is accepted.

How to submit

Both the source code archive file and the documentation file are to be submitted via Blackboard Assignment. Blackboard Assignment submission instructions are at: http://it.stonybrook.edu/help/kb/creating-and-managing-assignments-in-blackboard. You must read the submission instructions very carefully, and check to make sure that your project has been submitted correctly before the deadline. You can only submit once. Only click on "Submit" after you have checked and are certain that all requirements are followed.

Due date

The project due date is 23:55pm, Tuesday, December 13. No late submissions will be accepted.

Tentative marking scheme

- 30% of marks will be allocated to the documentation, within which 20% will be on logically structured, well documented, and easy to understand code, 80% for a clean written documentation.
- 70% of marks will be allocated to the correct implementation of the specified functionality.

Total: 100pts (The bonus tasks account for 12pts extra. Since the project is 15% of the course grade, the bonus task does not add too much to your course grade. You should place your priority on the final exam instead.)