CIT 595: Computer Systems Programming, Spring 2021
Project (15% credit)
Due Date: April 27, 2021 11:59pm EST

Goals

- Concurrency: management of threads, use of mutual exclusion or semaphores to avoid deadlocks and resource mismanagement
- Networks: the creation of a client-server "socket" interface (TCP), networking a multi-threaded API
- Advanced topics: fault tolerance, user interface, more networking (RPC, HTTP)

Introduction

In this project, you will design and implement an electronic voting system, consisting of a client and multi-threaded server, using TCP sockets for networking.

Instructions

At a high level, you will be writing a client-server application that implements an electronic voting system. The client should be able to send requests to the server that simulates different voting actions; the server should accept and process those requests and send replies back to the client.

We've provided a Codio project with a Makefile to get you started. The implementation is open-ended, but your final submission should produce the server and client targets defined in the Makefile when `make all` is called.

Note: You may and probably should use any other software/service to collaborate with your team members including (but not limited to) Git, Drive, Dropbox, Box, *etc*. We strongly recommend using a version control system like Git and a hosting service like GitHub.

Server

The server should be multi-threaded—that is, the server should be able to handle multiple requests from multiple instances of the client concurrently. Each request is handled by a separate thread.

For milestone 1, it is sufficient for the server to receive requests through standard input much like your program from Homework 2 Part 1. For milestone 2 and beyond, the server should accept connections from clients through a TCP socket. While your code should be designed to work over the network, you can assume that the server will run locally using localhost or 127.0.0.1.

The server should also be persistent: *i.e.*, (1) it should work for consecutive elections, and (2) it should be possible to shut down and restart the server in the middle of an election without losing any information about candidates, voters, and votes. When the server is shut down, it saves its state to a data file with the format of your own choosing.

The server should support a few optional command-line options:

- `-a <password>`: This should set the password for the election administrator. Otherwise, the default password for the administrator should be "cit595".
- `-r <filename>`: The provided file should include the voting data from an election that was interrupted (i.e. the server was shut down in the middle of an election). If the user starts the server with this option, the server should read back in the data and continue the ongoing election. Otherwise, the server should start with a blank state.
- `-p <port>`: The server should accept connections on the specified port; otherwise it should use port 10000.

The server can be started with any of these options or none at all. For example, running `./server

`-a password -p 10001` should set the admin password to "password" and set the server port to 10001; since the -r flag was not used, the server should be started with a blank slate. To help with parsing the command-line options, you can use the *getopt()* function. If you receive invalid command-line options, you should print an error message and exit the program.

Client

The client performs none of the vote processing. It is instrumental in sending a request to the server via the TCP socket programming protocol.

The client must process two command-line arguments:

- `<port>`: This is the port used to connect to the server.
- **"**`<command_name> <arg1> <arg2> ... <argN>`**"**: This is the request to send to the server.

If the user does not include these arguments in the command-line, you should print an error message and exit the program.

Running `./client 10001 "add_voter 9999" ` should send the request "add_voter 9999" to the server at port 10001. The client should print the server's response to standard output and exit.

A separate client program is only expected for milestone 2 and beyond. For milestone 1, the server should read in its requests directly through standard input.

Supported Functionality

This section details the possible requests clients could send to the server. It is up to your team on how the client and the server programs exchange information, but you should expect the user to enter the commands in the format: "`<command_name> <arg1> <arg2> ... <argN>`".

The server should support the following commands for administrators through a client:

- start_election
  - Arg1: string *password*
  - Result: string
  - Start a new election in zero state with no candidates, no voters, and zero votes. Return "OK" if successful, "EXISTS" if an election is ongoing, and "ERROR" if there's any error.
- end_election
  - Arg1: string *password*
  - Result: string
  - Ends ongoing election. Returns the list of candidates with their vote count as well as the winner of the election in the following format:

    "<candidate1>:<count1>
    <candidate2>:<count2>
    ...
    <candidateN>:<countN>
    Winner:<candidateA>"
  - If there has been a draw, the last line of the returned string should be "Draw:<candidateA>,<candidateB>". If there is no winner, the last line should be "No Winner".
  - Return "ERROR" if the election has not ended or there's any other error.
- add_candidate
  - Arg1: string *password*
  - Arg2: string *candidate*
  - Result: string
  - Adds *candidate* to the store of candidates, voters, and votes. If an early voter

writes-in a candidate before the store is populated, the candidate vote should not be overridden. Returns “OK” if successful, “EXISTS” if the candidate already exists, “ERROR” if there is any error.

- shutdown
  - Arg1: string *password*
  - Result: string
  - Initiates shutdown of the server. If there is an ongoing election, the server should stop accepting new connections, allow all threads to finish processing existing commands, and save all data from the election to the file named **backup.txt**. The server should return “OK” before shutting down.

Note: The first argument for each of the admin commands should always be the administrator password; if the password provided is not correct, the server should consider it an error and return ERROR to the client.

Note: All commands result in the client printing the result

The server should support the following commands for voters through a client:

- add_voter
  - Arg1: int *voterid*
  - Result: string
  - Registers a new voter ID if it is in the set of valid voter IDs given by [1000, 9999]. Return “OK” if successful, “EXISTS” if the *voterid* is already present in the system, and “ERROR” if there's any error.
- vote_for
  - Arg1: string *name*
  - Arg2: int *voterid*
  - Result: string
  - Adds one vote to the total vote count of the candidate referred to by *name* if the *voterid* has not already voted. If the named candidate is not already present in the system, add the candidate to the system with a vote count of 1. Else, if the candidate is already present in the system, increment their vote count by 1. Return “EXISTS” if the candidate exists and was successfully voted for, “NEW” if the candidate didn't exist previously and was successfully voted for, “NOTAVOTER” if *voterid* is not in the list of registered / valid voters, “ALREADYVOTED” if the *voterid* had already voted, and “ERROR” if there's any error. In addition, this command returns a random number on a new line. We call this number the <*magicno*> (magic number). An example of the returned string would be

    “EXISTS
    <*magicno*>”
- check_registration_status
  - Arg1: int *voterid*
  - Result: string
  - Checks if the *voterid* is valid and has been registered. Return “EXISTS” if valid and registered, “INVALID” if invalid, “UNREGISTERED” if valid but hasn’t registered and “ERROR” if there is any error.
- check_voter_status
  - Arg1: int *voterid*
  - Arg2: int *magicno*
  - Result: string
  - Checks if the *voterid* has voted. Return “ALREADYVOTED” if voter has voted, “CHECKSTATUS” if voter isn’t registered, “UNAUTHORIZED” if *magicno* doesn’t match records and “ERROR” if there is any error.

The server should support the following commands for any user:

- list_candidates
  - No arguments
  - Result: string
  - Returns a list of candidates for the current election (but not their vote totals). Returns an empty list if the election has not started yet. The return format is as follows:

    “<candidate1>
    <candidate2>
    ...
    <candidateN>
    ”
- vote_count
  - Arg1: string *name*
  - Result: string
  - Returns the vote total for the candidate in string format referred to by *name*, or “-1” if the candidate isn't in the system or the election hasn’t started yet.
- view_result
  - No arguments
  - Result: string
  - Returns the list of candidates with their vote count as well as the winner of the election. Return ERROR if the election has not ended or there’s any other error. **Refer to the end_election command for return format.**

Extra Credit

The project has the following extra credit opportunities

1. Non-blocking server I/O **(20 extra-credit points)**
   a. Enhance your implementation by utilizing non-blocking I/O for implementing your TCP communication. The server would have an I/O thread that saves the requests, upon their arrival, into local files (e.g., based on the request type). These files would be processed asynchronously by the worker threads that can act on the requests concurrently with the I/O thread (HINT: use *select*).
2. Implement the client-server interface with RPCGEN **(30 extra-credit points)**
   a. You should use the RPCGEN package, write a server for the voting system and clients that use each of the RPCs you specified above. Your server and clients should follow the same specifications as the original project.
3. Recover from server failures **(30 extra-credit points)**
   a. A basic implementation of the project is not sufficient to handle server failures. In order to recover from failures, your server should perform more extensive checkpointing of the server’s state, so that upon failure, the server can recover from the exact state it was in before failure.
4. Client-side web user interface **(40 extra-credit points)**
   a. In addition to interacting with the server through a client program, users of this voting application should be able to make all the same requests through a browser. This will require your server to be able to parse HTTP requests and send HTTP responses. The UI can be as simple as static HTML for this extra credit.

Note: Extra credit is open-ended. If you have an idea you would like to implement, please propose it to your assigned TA and/or course instructors.

Milestones

Team Selection (Due Mar 23 at 11:59 PM)

You should form a team of 2-3 people and sign up for a Project Group in the People section in Canvas. You can use the team search feature on Piazza. If you are not converging on having a team, let the TAs/instructors know (before the deadline!).

First Milestone (Due by Apr 01) (10 points)

Create a single program (server-api) to concurrently handle command line requests. The server-api file should expect optional command line arguments of the form:

`./server-api <optional flag> <optional argument> `

Example:

`./server-api -p 10001 -a password -r filename.txt `

Commands are entered directly as standard input in the form

`<command_name> <arg1> <arg2> … <argN> `

Example:

` add_voter 9999 `

Your team will demonstrate all functionality of your code to your assigned TA on or before the recitation on Apr 01.

Second Milestone (Due by Apr 15) (15 points)

Create client and server programs interfaced with TCP. The server should expect optional command line arguments of the form:

`./server <optional flag> <optional argument> `

Example:

`./server -p 10001 -a password -r filename.txt `

The client should send requests to the server via terminal commands of the form

`./client <port> "<command_name> <arg1> <arg2> … <argN>" `

Example:

`./client 10001 "add_voter 9999" `

Then, the client should print the server's response to standard output and exit. Every request sent by a client (*i.e.,* every terminal command should be processed by a new server thread).

Your team will demonstrate all functionality of your code to your assigned TA on or before the recitation on Apr 15.

Third Milestone (Due Apr 27 at 11.59 pm) (75 points)

All content of the second milestone alongside extra credit functionality should be submitted by the above deadline. Your team's completed code should be submitted on Codio by one of your team members *(mention name of team member in the report)* and a brief report (1-2 pages) should be submitted to Canvas. Your report should contain the following:

1. Description of your completed work (including extra credit)
2. Description and brief justification of your design decisions
3. List of source files and a brief description of each
4. Instructions to run your programs
5. Team member responsibilities: whether all members contributed equally, or provide the relative weights indicating the contribution size of each member. The team should come to a consensus.

Recommended Roadmap

We recommend the following roadmap

1) First Milestone
    a) Create a single-threaded server-api program with necessary functionality
    b) Add to the above: creation of new thread for every new command line request. This should lead to a multithreaded server-api program.
2) Second Milestone
    a) Create client and server (single-threaded) programs networked with TCP
    b) Modify the above with: multi-threaded server API created in (1)(b)

3) Third Milestone
    a) One or more extra credit options can be added to the final output of the second milestone

Using Piazza...

Although you are encouraged to post questions on Piazza if you need help or if you need clarification, please be careful about accidentally revealing solutions.

In particular, please do not post questions that reveal your solutions to the assignment, e.g. how you used threads, how you implemented the blurring algorithm, etc.

If you think your question might accidentally reveal too much, please post it as a private question and we will redistribute it if appropriate to do so.

Academic Honesty and Collaboration

You must work on this project only with the members of your team.

You may discuss the project specification with other teams, as well as your general implementation strategy and observations, but you absolutely must not discuss or share code with another team under any circumstances.

Please see the course policy on academic honesty (posted in Canvas on the “Syllabus” page) for more information. Suspected violations of the policy will result in a score of zero for the assignment and/or be reported to the Office of Student Conduct at the instructor’s discretion.

Although you can, of course, look online for help with the C libraries and things like that, the work you submit must be your own. Copy/pasting code written by other people will be considered plagiarism and will be treated as academic dishonesty, even if you were “just looking at it to see how it’s done.”

If you run into problems, please ask a member of the teaching staff for help before trying to find help online!

Grading

This project is worth a total of 120 points:

- 10 for milestone 1
- 15 for milestone 2
- 75 for milestone 3
- 20 for no Valgrind/Helgrind errors

Please stick to the syntax of the commands described above and please implement TCP socket programming for any networking (apart from the RPC extra credit, which is done in addition to the TCP implementation). It is up to you how to structure your code.

For the first two milestones, you will be graded on your overall progress and design decisions.

For the third milestone, you will be graded based on the accurate final output, correct usage of threads, proper resource management, and the content of the report. Test rigorously with correct and malformed input.

We require your program to be free of any Valgrind / Helgrind errors, race conditions, and memory leaks. Together, these comprise 20 points of your grade.