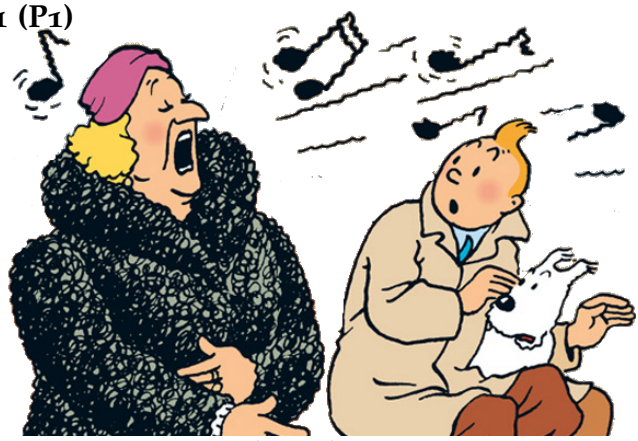
CS447-00x : Networks and Data Communications

Programming Assignment #1 (P1)

Total Points: 100



Assigned Date : Thursday, September 11, 2025
Due Date : Thursday, October 02, 2025 @ 01:59:59 p.m.

Overview

Your first programming assignment is to **implement a basic client/server application** using the socket interface. The learning objectives are to:

a. gain hands-on experience with socket programming fundamentals, mastering the order of socket API primitives;
b. become familiar with Linux system calls;
c. gain a basic understanding of network protocols; and
d. set yourself up for the rest of the course.

1 Back Story

Frustrated by the chaotic, manual course registration process, **Madame Castafiore**, the esteemed Dean of Student Affairs at **S.S. Calculus Academy**, has demanded a more elegant solution. She has enlisted the brilliant **Professor Calculus** of the Computer Science department to compose a new, network-based system. This digital platform will allow students to navigate their academic careers with grace, from browsing the course catalog to managing their enrolled classes, all from a remote terminal.
Professor Calculus has outlined the core functionalities of his online course registration system, categorizing them into three distinct modes:

1. **`CATALOG` Mode**
   - Search for courses by subject, instructor, or course-code.
   - Display details for a specific course, including prerequisites and availability.
   - List all available courses, with optional filters.
2. **`ENROLLMENT` Mode**
   - Enroll in a course.
   - Drop a course.
3. **`MYCOURSES` Mode**
   - View a list of enrolled courses.
   - View grades for completed courses.

Technical Requirements

Your task is to build a client-server application that fulfills the needs of the S.S. Calculus Academy's new registration system. The application must meet the following technical requirements:

- The server should support concurrency, allowing more than one **TCP** `client` to use the registration system simultaneously.
- Your protocol interaction should adhere to the following specifications:
- **Client Commands**[‡]**:**

1. **`IAM <student-name>`** - This is the **first** command issued by the `client` to initiate the connection and identify the student. The `server` reply code is `200`.
2. **`HELP`** - This command can be issued at any time after **`IAM`**. It requests a list of available commands. If issued within a specific mode, it should only list mode-specific commands and available modes. The server reply code is `200`.
3. **`CATALOG`** - This command enables clients to access the course catalog. Success is acknowledged by server reply code is `210`.
   (a) **`LIST [filter]`** - The `LIST` command lists all available courses, optionally filtered by **subject**, **instructor**, or **course-code**. The server replies with `250` and the list of courses, or `304` if no courses are available.
   (b) **`SEARCH <filter> <search-term>`** - The `SEARCH` command searches for courses by a specified `<filter>` (subject, instructor, or course-code) and `<search-term>`. The server replies with `250` and a list of matching courses, or `304` if none are found.
   (c) **`SHOW <course_code> [availability]`** - The `SHOW` command displays details for a specific course. When the optional `[availability]` argument is included, the server should only list the course's availability status and the number of available seats. Without the optional argument, the server should provide the full course description. The server replies with `250` and the requested details, or `404` if the course is not found.
4. ~~**`*ENROLL`**~~ **`ENROLLMENT`** - This command allows clients to enroll in or drop courses. The server's reply code is `220`.
   (a) **`ENROLL <course_code>`** - This command enrolls a client in a course. The server replies with `250` on success, `403` if the course is full, or `404` if the course is not found. **Prerequisites** for a course are considered met if prerequisite course(s) are listed in the current enrollment history.
   (b) **`DROP <course_code>`** - This command allows a client to drop a course. The server replies with `250` on success or `404` if the course was not enrolled by the client. Dropping a course removes it from the student's active enrollment.
5. **`MYCOURSES`** - This mode provides clients with functionalities to manage their academic schedules. The correct server reply code is `230`.
   (a) **`LIST`** - This command displays the student's current enrollment *(and by that virtue the history)*. The `server` replies with `250` and the list of courses, or `304` if no courses are found.
   (b) **`VIEWGRADES`** - This command shows the student's grades for completed courses. The `server` replies with `250` or `304` if no grades are available.
6. **`BYE`** - This command closes the connection and requests a graceful exit. The `server`'s reply code is `200`.

- **Server Reply Codes:**

1. **`200/210/220/230 Command Success.`** These reply codes indicate that the command was successful.
2. **`250 <data>`** - Indicates successful command execution with data returned.
3. **`304 NO CONTENT`** - This reply code indicates that the server successfully processed the request, but there is no content to send back (e.g., no search results).
4. **`400 BAD REQUEST`** - This reply code indicates that the server could not understand the request due to invalid syntax or missing parameters.
5. **`403 FORBIDDEN`** - This reply code indicates that the server understood the request, but refuses to fulfill it (e.g., trying to enroll in a full course).
6. **`404 NOT FOUND`** - This reply code indicates that the server did not find the requested resource (e.g., a course with a specific ID).
7. **`500 INTERNAL SERVER ERROR`** - This reply code indicates that the server encountered an unexpected condition that prevented it from fulfilling the request.

[‡]**Note**: Arguments enclosed in angle brackets (< >) are required, while arguments in square brackets ([ ]) are optional.

Functional Requirements

1. **Starter Code & Helper Library**
   - The provided starter code must be used. Submissions using other base code will be considered invalid.
   - A Helper Library (`p1_helper.h/p1_helper.cpp`) and a sample course database (`courses.db`) are provided. The library functions are designed to help you manage an in-memory database of courses on the `server`.
   - Students are expected to use the provided `p1_helper.h` and implement the stub functions in the provided `p1_helper.cpp` file. The server logic should call these helper functions as needed, and students should not implement or duplicate their functionality within `server.cpp`. The implementation for the `load_courses_from_db` function is already provided.
   - The `p1_helper.h` file **must not** be modified.
   - The course database (`courses.db`) is a text file that can be modified for testing purposes as you see fit without changing the file formatting.
   - The starter code is provided to you as a compressed tarball. You can use the following command to uncompress it:

     ```
     tar -zxvf starter-code.tgz.
     ```

2. **Network Setup**
   - A `server.conf` file must be used to configure the port number at which the server is listening. Once compiled the server's runtime signature should look like follows: `./server server.conf` **Note**: The provided starter code has a hard-coded port number, which you need to fix to meet the required runtime signature.
   - The client side of this assignment does not require a separate program. You can use standard command-line tools like `telnet` or `netcat` to connect to and interact with your server. The runtime signature should look like:
     - `telnet <server-ip> <server-port>`
     - `nc <server-ip> <server-port>`
   - The server must acknowledge the client's IP and connection type. *(See sample interaction below)*

3. **Client-Server Interaction**
   - Client-server interaction must adhere to the protocol specifications (client commands, server reply codes, etc.) detailed in the previous section.
   - The `client-server` connection must use **TCP**.
   - The `server` must be capable of handling multiple simultaneous client connections. The provided starter code uses C++20's `std::jthread` for concurrency. It's strongly recommended to familiarize yourself about `std::jthread`, if this is a new construct for you.
   - Clients must exit using the **`BYE`** command. The server may be terminated forcefully.

4. **Testing & Deployment**
   - Test your implementation with at least 2 (more is better) simultaneous client connections, preferably using separate zone server containers. Test for potential interference by concurrent clients.
   - Ensure your code compiles and runs on a Linux machine. Provide a `README` with clear manual compilation instructions in case your `Makefile` fails, and any additional software dependencies.

5. **At the end of your implementation, you should be able to**
   - Compile and run your code using multiple **`zone server`** instances.
   - Run your `server` program first.
   - Use `netcat/telnet` to interact with the `server`
   - Perform course registration functionality while meeting the technical requirements mentioned above.
   - Exit the client(s) gracefully.

6. Below is a non-comprehensive sample interaction. Assume the client's IP address is `192.168.0.11`, and the server is running on the `tgamage-0` zone server instance.

```
                    Client        Server
                IAM Jane Doe   →
                               ←   200 WELCOME Jane@192.168.0.11
          *ENROLL ENROLLMENT   →
                               ←   503 Bad sequence of commands.  Must enter a mode
                                   first.
                     CATALOG   →
                               ←   210 Switched to CATALOG Mode
      SEARCH subject Chemistry   →
                               ←   250 <list of Chemistry courses>
  SHOW MATH250 availability   →
                               ←   250 Availability:  Open, Seats:  25
                  ENROLLMENT   →
                               ←   220 Switched to ENROLLMENT Mode
               ENROLL CS201   →
                               ←   403 FORBIDDEN. Prerequisites not met.
              ENROLL ART205   →
                               ←   250 ENROLLMENT SUCCESSFUL.
                  MYCOURSES   →
                               ←   230 Switched to MYCOURSES Mode
                       LIST   →
                               ←   250 Enrollment History:
                                       ART205
                 VIEWGRADES   →
                               ←   304 NO CONTENT. No grades found.
                        BYE   →
                               ←   200 BYE
```

Instructions

- **Start early & backup often:** This assignment requires careful planning and implementation. Don't procrastinate, and remember to save your progress frequently.
- Implement your solution in C++. The sample code is written for C++20. Write clean, readable code, adhering to a recognized style guide like Google's C++ Style Guide `https://google.github.io/styleguide/`.
- Your code must compile and run flawlessly on a standard Linux environment. Thoroughly test your implementation on the **`zone server`** using command-line tools.
- Submit a `.tgz` and a `.pdf` report. See specifications below.
- Start with a single client interaction to a baseline implementation and then gradually scale up to handle parallelism. Plan your data structure usage. This is not a trivial task.
- Focus on core C++ socket and I/O functionalities. Avoid using external libraries unless explicitly permitted.
- DEADLINE: **Thursday, October 02, 2025 @ 01:59:59 p.m. (for both sections)** through Moodle. Email submissions or requests to "*review*" your implementation before submission are not honored.

Deliverables

A complete solution comprises of two attachments:

1. Report (**`pdf`**): The report doesn't carry a lot of points on its own but serves as crucial evidence for partial credit. It's your only way to earn points if automated testing fails or your submission has bugs. Use it to clearly document the functionality you implemented and validated (with screenshots!) and, just as importantly, to state any features that are not working or were not implemented. There's no penalty for not submitting a report, but remember, if your code doesn't work, you will receive zero points without one.
2. Compressed Tarball (**`siueID-p1.tgz`**):
   - Source Code Directory: Include your `*.cpp` and `*.h` files. The provided `p1_helper.h` is not needed as its meant to be immutable.
   - `Makefile`: A Makefile is required. The instructor should be able to compile your code without errors or warnings by simply typing **`make`** on the terminal.
   - `README`: Provide clear instructions and any information on how to compile and run your code. Include manual compilation instructions in case your `Makefile` fails to work as expected. Also, list the directory structure and the file contents of your tarball once it's uncompressed. A `README` should be exactly what it sounds like: a complete set of instructions and information for a would-be user of your code.
   - `courses.db` and `server.conf` files do not need to be included in your submission. The instructor will use his own versions for testing.
   - **ONLY** include files listed above. Submissions with deep folder structures, and/or hidden meta files and folders (from perhaps your IDE or version control e.g. `.code`, `.git`) are subject to penalties.
   - Use the following command to create the compressed tarball:

     ```
     tar -zcvf siueID-p1.tgz p1/.
     ```

     Replace `siueID` with your actual SIUE ID and `p1/` with the name of your source code directory.
     e.g. `tar -zcvf tgamage-p1.tgz p1/`. Run this command from the immediate parent directory to avoid unnecessarily deep folder structures. **You are fully responsible for ensuring your tarball is not empty or corrupt**. Submitting an unusable file is not an acceptable excuse for a re-submission.

This assignment is designed to push your boundaries and foster growth. The goal is to master these concepts, not just complete the task. **Plagiarism, whether from classmates, online sources, or AI tools, stifles learning and compromises academic integrity**. The instructor actively uses MOSS (http://theory.stanford.edu/~aiken/moss/) to check for software similarity, and plagiarism has severe consequences as outlined in the course syllabus. **There will be no exceptions**.
Your code should be a testament to your own abilities. I encourage you to collaborate but not copy. Learning from peers is a valuable part of the process, but simply copying someone else's code (whether from a classmate or online) is strictly prohibited. If you use external resources for ideas, you must cite them and then implement those ideas in your own way.

Some Useful Resources

- Linux Man pages – found in all Linux distributions
- Beej's Guide to Network Programming – A pretty thorough free online tutorial on basic network programming for C/C++ https://beej.us/guide/bgnet/.
- Linux Socket Programming In C++ – https://tldp.org/LDP/LG/issue74/tougher.html
- The Linux HOWTO Page on Socket Programming – https://www.linuxhowtos.org/C_C++/socket.htm
- Learn Makefiles With the tastiest examples – https://makefiletutorial.com/