CS371 PA1: socket programming and epoll

Problem description: Socket programming is a fundamental concept in network programming that allows applications to communicate over a network using TCP/IP protocols. In a typical client-server model, the server listens for incoming connections, while clients establish connections and exchange data.

Traditional socket programming uses blocking I/O, where each connection is handled sequentially, leading to inefficiencies under high loads. To address this, Linux provides the **epoll** API, which enables efficient handling of multiple connections in an event-driven manner. **epoll** allows a server to monitor multiple file descriptors asynchronously, reducing CPU overhead and improving scalability.

In this assignment, you will implement a client-server application using socket programming and leverage **epoll** for efficient multiplexing of server application.

Objective: The objective of this assignment is to implement a simple client-server application in **C** using **TCP sockets**. The server will use a single-threaded, event-driven model with epoll to handle multiple client connections, while the client will use multiple threads to establish connections and communicate with the server. The client should also measure round-trip time (RTT) and request rates.

Note: I have provided a skeleton code (**pa1_skeleton.c**) that you can just follow the "**TODO**" in the code to complete the required tasks (described on page 3) without worrying about the structure of the code.

Grouping: You can form a group of no more than three students in this class to complete the programming assignment together. You can also choose to complete the programming assignment individually, but the scoring criteria are independent of the group size.

For students who need to earn honors credits through this course, please complete the "Bonus Challenge" separately. For other tasks in the programming assignment, you are free to choose to work with group members.

Tasks

Server Implementation

- The server, which runs as a single binary process, should be single-threaded and use a run-to-completion event loop.
- It should listen for incoming client connections and handle data transmission, both using "**epoll**".
- Upon receiving a message from a client, the server should echo the message back to the client.

Client Implementation

- The client, which runs as a single binary process, should support multiple threads, each establishing a separate TCP connection to the server.
- Each client thread should use an independent "epoll" instance to monitor server responses.
- The client should measure the round-trip time (RTT) for each message and collect the following metrics:
    - The average RTT for messages sent by an individual client thread.
    - The overall average RTT across all client threads.
- The client should also track the request rate per thread and compute the total request rate across all threads.

Key Implementation Requirements

- Use the "epoll" API for efficient event-driven programming on server side.
- Use TCP sockets ("SOCK_STREAM") for communication.
- Ensure proper synchronization and avoid race conditions in multi-threaded client implementations.
- Handle error cases gracefully (e.g., failed socket connections, epoll failures, etc.).
- Properly close the file descriptors of sockets and epoll.
- Collect and report statistics (RTT, request rate) at the end of execution.

For students who need to earn honors credits through this course, please refer to "Bonus Challenge" for additional tasks.

Skeleton Code

We provide a skeleton code to simplify the development of server/client tasks. You can follow the "TODO"s in the skeleton code to complete the corresponding task. The client thread sends a **16-Bytes** message every time. Link to download the skeleton code:

https://gist.github.com/ShixiongQi/923caf880eab090dd329a6cd1da3cd19

Command-Line Arguments

The skeleton code is a mixture of both server and client programs. It accepts the following 5 command-line arguments:

- "<server|client>" – decide whether the skeleton code will run as a server or as a client program (see example usage below)
- "server_ip" - The IP address of the server.
- "server_port" - The port number of the server.
- "num_client_threads" - The number of client threads to launch.
- "num_requests" - The number requests that each client thread will send.

Note: Please DO NOT change the command-line argument (except for Bonus Challenge). We will deduct 50 points from your final score if you change the Command-Line Arguments

How to compile after you complete the TODOs in the skeleton code

Note: the instructors of this course stick to Ubuntu 22.04 for development and evaluation of this programming assignment (for details, refer to "Development and Evaluation Environment").

```
gcc -o pa1_skeleton pa1_skeleton.c -pthread
```

Example usage:

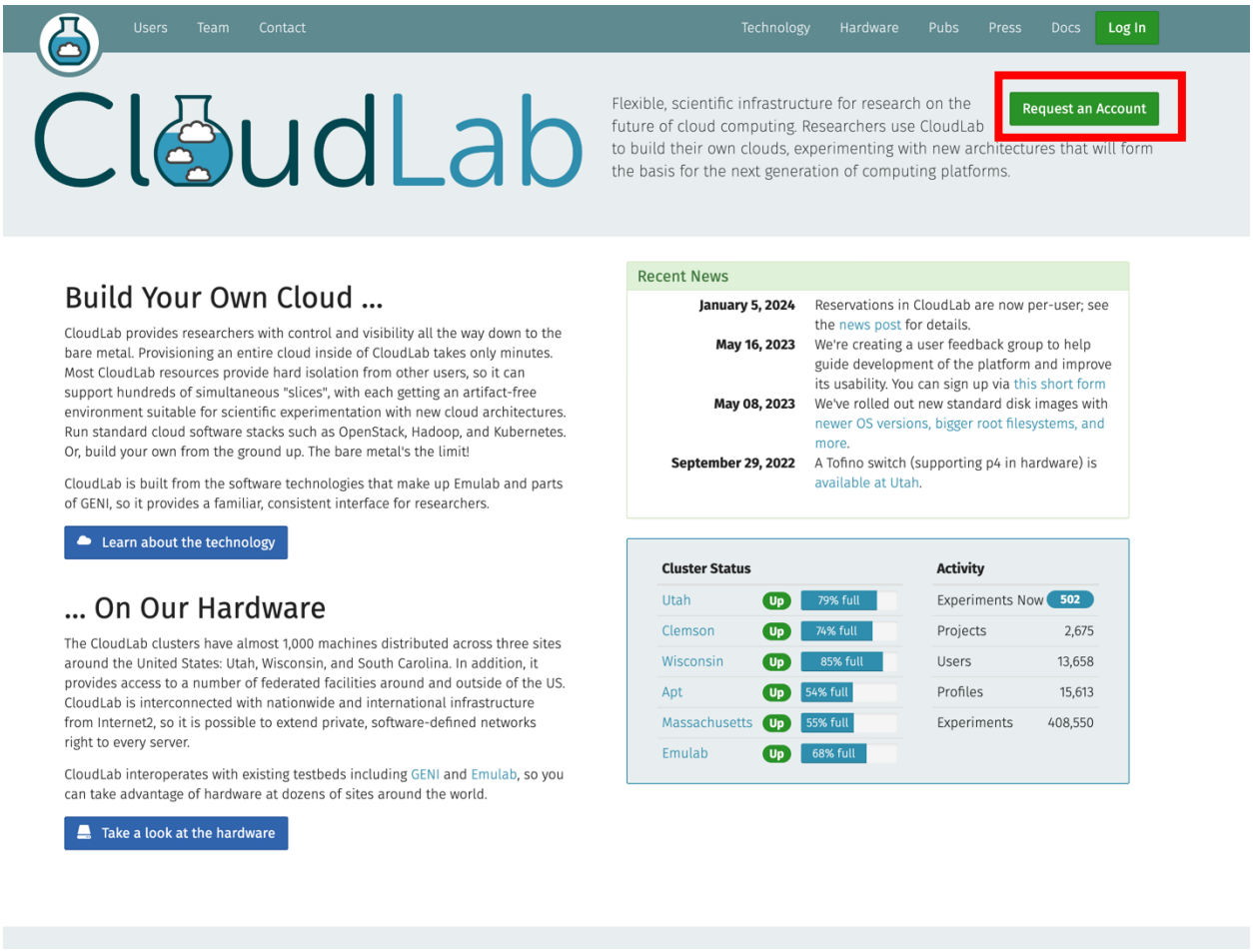
```
# Start the server first with designated IP and port number
./pa1_skeleton server 127.0.0.1 12345
# Then run the client with 4 threads, each thread sends 1000000 messages
# The client threads will connect with designated server IP and port number
./pa1_skeleton client  127.0.0.1 12345 4 1000000
```

Development and Evaluation Environment

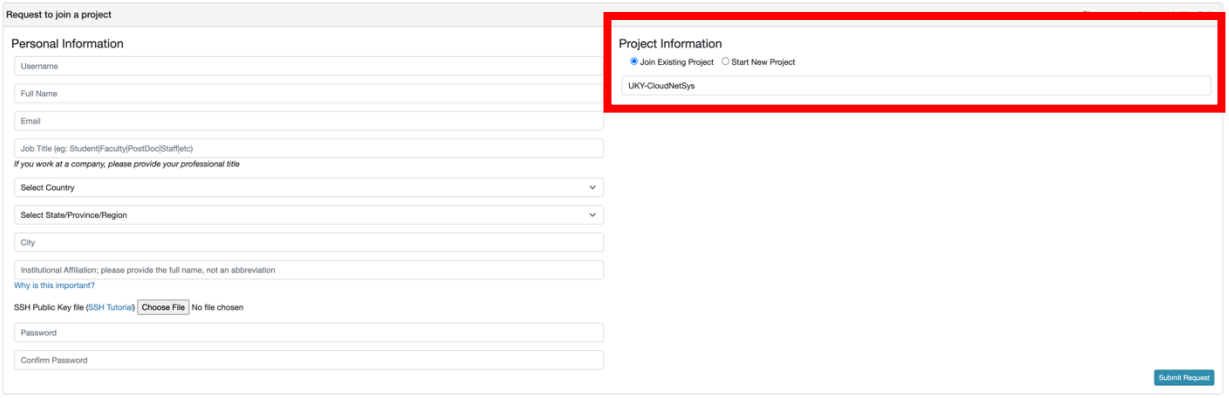
The skeleton code was developed on Ubuntu 22.04 server, using x86 CPUs. We will stick to this setup, using the x86 Ubuntu 22.04 servers on NSF CloudLab for development and evaluation of this programming assignment.

We encourage all students register an NSF CloudLab account and join the "**UKY-CloudNetSys**" project, where you can get exclusive access to an Ubuntu 22.04 bare-metal server for a certain period (it's free!). Please follow the steps below for **CloudLab Registration**:

1. Go to https://cloudlab.us/ and click Request an Account



2. Fill in your personal information, select **Join Existing Project** and enter **UKY-CloudNetSys** then Submit and wait for approval**.**



3) This video provides an introduction to cloud lab:

Creating a New Experiment in Cloudlab and How to Extend an Experiment

https://uky.zoom.us/rec/share/DXfu7qXihjnpYJgGM3sicsEwwZgN6qHGKewR_9sINU7w4PrMNXewakfRkgjuwoA6.tnzG03d22xZkRtTA

Useful references for Cloudlab usage:

- Resource availability on Cloudlab: https://www.cloudlab.us/resinfo.php
- Hardware specs on Cloudlab: https://docs.cloudlab.us/hardware.html

Important Note: You can also use the Ubuntu virtual machine to develop work on your own laptop, but please test the code on NSF CloudLab before submitting. You are responsible for ensuring that the submitted code can run on the x86 Ubuntu 22.04 machine on NSF CloudLab.

If you do not comply with this policy, your submission will be counted as 0 points because we cannot evaluate your submission.

Submission Instructions

Please upload your programming assignment to Github and submit the link to your Github repository for your programming assignment to Canvas.

Other submission methods will not be considered and will be scored **zero**.

Please follow the requirements below when submitting your programming assignment. Your Github repository for your programming assignment must contain the following files:

- Source code (modified "**pa1_skeleton.c**" file).

Please note that we will use the last commit before the due date to evaluate the programming assignment. Any commits after the due date will not be considered.

Grading Policy

Evaluation Criteria

The total score is 100 points. You need to ensure that the client and server can run seamlessly, following the requirements of the "TODO" in the skeleton code. If neither the client nor the server can run, the submitted score will be **zero**. The following score breakdown is based on the client and server being able to run, that is, the client can send messages to the server, and the server can send messages back to the client without any errors or crashes:

- Implementation of client-side logic (**35 pts**)
- Implementation of server-side logic (**30 pts**)
- Collect and report statistics (RTT, request rate) at the end of execution (**20 pts**)
- Properly close the file descriptors of sockets and epoll (**5 pts**)
- Handle error cases gracefully (**5 pts**)
    - e.g., failed socket connections, epoll failures, etc.
- Code readability (**5 pts**)
    - Use consistent indentation (e.g., 4 spaces) to visually structure code block
    - Use consistent spacing around operators, keywords, and other code elements

Important Notes:

- Please use TCP sockets ("SOCK_STREAM") for communication
    - Your submission will be counted as 0 points if you use other socket types.
- Please DO NOT change the command-line argument (except for Bonus Challenge).
    - We will deduct 50 points from your final score if you change the Command-Line Arguments
- Please follow Submission Instructions on page 6.
    - Other submission methods will not be considered and will be scored zero.
- If you do not comply with the policy of Development and Evaluation Environment, your submission will be counted as 0 points because we cannot evaluate your submission.

Bonus Challenge (20 pts)

This is only required for students who need to earn Honors credit

- Extend the server to be multi-threaded and enforce the load balancing between client threads and server threads.
- Talk to the instructors if you need help.

References

- Practical TCP/IP Sockets in C, Second Edition
    - https://cs.baylor.edu/~donahoo/practical/CSockets2/
- Linux epoll documentation
    - https://man7.org/linux/man-pages/man7/epoll.7.html
    - https://linux.die.net/man/4/epoll