CS 744: Design and Engineering of Computing Systems
Autumn 2023

Project DECServer

Clarifications/Reminders

In this section we'll add running clarifications of issues that emerge after students start implementing the labs building towards the final project.

- **Queue implementation in Multi-threaded-with-thread pools.**
  - You are most welcome to use C++ STL library for the queue (or in C, if you know of any equivalently good library). This is not a data structures course and from-scratch implementation of 'your own queue data structure' is not required.
- **Timeout Issue.** There seem to be two ways to do this:
  - Using setsockopt system call in the client and set a SO_RCVTIMEO option on the socket FD used by the client. In this option you must check for errno after you exit the recv call to see if it is set to EWOULDBLOCK or EAGAIN. Example code fragment:

```
l = recv(...)
if (l < 0)
{
     if (errno == EWOULDBLOCK || errno == EAGAIN)
     {
          // Timeout, handle it somehow
     }

     break;   // Exit receiving loop
}
```

  - *The client can start another thread for handling the entire request-response. The main thread uses **`pthread_timedjoin_np`** to wait for the child to finish, otherwise it timeout and terminates the thread. See the man page for details.

    I actually prefer this option because it better represents an overall user level timeout. A human user will simply timeout sometime after she issues a submit, whether it's a connect timeout or send timeout or recv timeout will not make a difference. This option is also easier to get working.
- **Should the load generating client send requests over the same socket connection.**

    - **NO.** The ‘loop’ implemented for sending multiple requests should be over the entire process of creating a socket, connecting, sending a request, receiving the response and then closing the connection.
- **Git**
    - You can use (are in fact encouraged) to use git.cse.iitb.ac.in
        - However if you area an active github user and want to add this to your existing account that is also ok

  Share the account with your project TA. **In either case, DO NOT MAKE THiS PROJECT REPOSITORY PUBLIC, NOW NOW, NOT EVER.** This project may be a regular feature in CS 744 - students shouldn’t just find the code in github.
- **PLEASE KEEP ALL VERSIONS OF THE CODE (even if you did not submit any - i.e. the V1, V2… etc that I am defining).** In the final project submission, a *comparative performance analysis of all 5-6 versions will be required, and viva may be conducted on any of these versions.*
    - Each version should implement the main paradigm correctly. *Do not add arbitrary unapproved ‘sophistications’.* As a reminder the progression so far is **strictly:**
        - V1: Single threaded server **(Multithreaded will not be accepted)**
        - V2: Multi threaded with create-new-thread-per-request paradigm (**thread pools will not be accepted)**
        - V3: Multithreaded with thread pools and request queue. **(No need to add any CPU pinning, any containerization, etc etc)**

You are most welcome to play with all these things, however for the purpose of submission and grading please stick to the sequence given.

In this project we will step-by-step build a scalable program autograding server (and its client).

We can assume that the purpose of the submitted program is simply to print the first ten numbers:
1 2 3 4 5 6 7 8 9 10

If the submitted program prints this output, it has passed, else it has failed. You may assume the programming language to be either C or C++. Do not assume python.

The autograding client and server themselves should be written **highly preferably in C++,** however C is also ok. **PYTHON IS NOT ALLOWED.**

In all versions, the server will always be run as follows,

```
$./server  <port>
```

and the client will always be run as follows,

```
$./submit  <serverIP:port>  <sourceCodeFileTobeGraded>
```

and will get back one of the following responses from the server:

1. PASS
2. COMPILER ERROR
3. RUNTIME ERROR
4. OUTPUT ERROR

In cases 2,3,4, the server should additionally send back the error details:

- For compiler error, the entire compiler output should be sent back to the client
- For runtime error, the error type should be sent back to the client
- For output error, the output that the program produced, and the output of a 'diff' command should be sent back to the client

The following will be the step-by-step 'versions' through which we will build the server.

Version 1 (Lab 06): Single threaded Auto-grading server (due 30th Sept)

In this version the server is a single threaded server, that does the above functionality. I.e. on receiving the source code file, it

1. Compiles it. If there is a compiler error, it sends the message and info back to the client else does the next step.
2. Run the executable. If there is a run-time error, it sends the message and info back to the client else does the next step. If it ran successfully, the output should be captured.
3. The output is compared with desired output. If not matching, send back the error and diff output.

Use the simple single threaded client server code discussed in class as the starting point.
Submission instructions:

1. gradingserver.c or cpp: the server code
2. gradlingclient.c or cpp: client code

Just submit these files one by one (no zipping required).

Lab 07: Performance Experiment Setup for All versions (due 7th Oct)

The purpose of performance experiments is to quantify the 'capacity', 'scalability' or 'responsiveness' of your server. We will learn more about these in our performance concepts lectures next week. For this week, you should just build the tools needed to measure performance.

To measure performance of a server with increasing load, typically we do a 'load test'. In the load test, we simulate multiple simultaneous clients, each in a 'request-response' loop with the server - i.e. The client sends a grading request, wait's for response, then sleeps for some time, then sends the next request, and continues like this. This loop with some M clients is run for a few iterations. The aim is to measure the performance of the server as a function of the increasing 'load' (load here = number of simultaneous clients). By 'performance' here, we mean the following metrics:

1. Number of successful requests/second in the experiment - this is called *throughput*
2. Average *response time* of a request in the experiment.

You can do this step by step as follows (this is just a **suggested** design. You can do it some other way):

Step 1:

First, convert your client into a 'load generator' client - i.e. let it take 2 more arguments:

- Number of iterations of the request-response loop it should do
- Sleep time

```
./submit <serverIP:port> <sourceCodeFileTobeGraded> <loopNum> <sleepTimeSeconds>
```

For this, first figure out how to measure the time taken by your client to get the grading response from your client. I.e .measure the *response time* of the client.

- All you need to do in this case, is to read the current local time just before sending the request (Tsend), then read the current local time just after getting the response (Trecv), and calculate the difference: Trecv - Tsend.
    - Make sure you use finer granularity than seconds. **gettimeofday is better than time (in C)**
- This will give *one sample* of response time
- Note that this part you need to do inside the client code.
- Check that this is working
- Now add the loop with the sleep timer added *after* the client gets the response.
    - In each iteration of the loop you will get a response time, accumulate it into a sum
    - Additionally keep track of how many successful responses you got.
- Output **the average response time at the end, the number of successful responses, and the time taken for completing the loop.**

Now your client is ready to be a part of a set of ‘load generating’ clients.

Step 2
Write a shell script that takes as an argument how many clients you want to start, the number of iterations each should do, and the sleep time between response and next request.

./loadtest.sh <numClients> <loopNum> <sleepTimeSeconds>

The script

- starts some M clients (in the background), each with output redirected to a different filename (do this cleverly using some counter index).
- waits until all clients are done (‘wait’ bash command maybe useful, check man page).
- Then based on the outputs in the file calculates overall throughput and response time as follows
    - Overall throughput = sum of all individual throughputs of the M clients
    - Average response time = Sum_i (N_ i * R_i)/Sum_i (N_ i)
        - Where R_i is the average response time for client *i*
        - N_i is the number of response time samples measured by client *i* (= number of successful responses received).

Now run this script for a varying number of M clients - increase M until the performance begins to feel ‘bad’ (we will learn more about this, later). Plot throughput and average response time vs M.

Submit the load generating client code, the driver shell script,and a pdf of the above two graphs.

Lab 08 Autograder Version 2: Multi-Threaded Server with Create-Destroy threads (Due Sat 14th Oct)

NOTE: PLEASE KEEP VERSIONS, YOU WILL NEED EACH FOR LATER COMPARISONS

1. Add multithreading capability to the server. I.e. now you have a listener thread that accepts grading requests and creates a worker thread to process each request. A thread should be created for each request and should exit after serving one autograding request. The thread should directly write the response back to the client and then exit
2. Additionally, update the client to add a **timeout** (you can give it as one more command line argument)

```
./submit <serverIP:port> <sourceCodeFileTobeGraded> <loopNum>
<sleepTimeSeconds> <timeout-seconds>
```

3. Upgrade your performance measurement setup now to measure **CPU utilization** and **Average number of active threads**.
    a. For each load level, just before you start the experiment, start taking 10 second snapshots of CPU utilization using ***vmstat***, then stop after the experiment. (Needless to say, save this to a file.) Similarly use **ps eLf** or any other tool you like to take snapshots of **NLWP**
    b. Calculate the average CPU utilization and average number of active threads (ensure you do not use 'warm up' and 'cool down' values).
4. Now do the same performance experiments as before. First do these experiments by keeping the number of cores small (2 cores). Ensure the load is such that you are able to drive the CPU to 100%.
    a. Now for each load level (i.e. number of clients), apart from throughput and response time, also calculate request sent rate, and timeout rate (so request sent should be = throughput + timeout + error rate if any)
    b. Do the experiments for various load levels and plot these curves vs M:
        i. Request rate sent
        ii. Successful request rate (we will now call this **goodput)**
        iii. Timeout rate
        iv. Error rate if any (connection refused etc)
        v. CPU utilization
        vi. Average number of active threads
    c. Compare the performance metrics with the Version 1 metrics (plot them on the same graph)

The experiments should be done from a 'low load' level to a 'saturation' level. This means that you should see the throughput curve flattening (or even starting to go down). Choose the load level steps such that not more than 20% of your curve is in the 'saturation' region.

Submit the update code in separate code files and the experiment results in a PDF.

Lab 09 Autograder Version 3: Thread Pools

Due Thursday October 26th.

Now update the grading server to a 'thread pool' design. In the previous version, a thread was created and destroyed for every request. In this design, first of all there is an overhead of thread creation and destruction, and secondly there is no control on how many threads are created. Your Lab 08 experiments would have shown that indiscriminately creating threads would result in throughput degradation after server saturation at high load. Thus, you will now implement a *thread pool* design.
To understand this concept in more detail, refer to Prof Mythilli's slides and lecture video for an overview of multithreaded server design.

Specifically: update your server to accept a thread pool size which is the number of threads the server has active. This thread pool size is fixed.

```
$./server  <port>  <thread_pool_size>
```

- At the beginning, the server main thread creates the thread pool of size `thread_pool_size`. These threads will start and wait for grading requests.
- You will now need a shared queue in which the main thread queues grading requests (connections). Whenever a worker thread is free it should pick up the next grading request from the shared queue. After that the functionality is the same as before, the thread directly sends the grading response to the client.
- You will need **mutexes** (for thread-safe access to the shared queue) and **condition variables** so that the threads are not constantly checking the queue in a 'busy loop'.
- **From this submission onwards, your server code will get large enough that it should be split into multiple files. This is also the *only way* for multi-person teams to write code for a single application. This will also help you learn how to write C/C++ code in multiple files.**
    - **Start a habit of writing the code author name in the comments.**
- **Also create a git repository, create a project for this server, share it with your project TA (details will be shared later).**

Refer to this tutorial for git usage basics.

REPEAT THE SAME PERFORMANCE EXPERIMENTS AS DONE IN LAB 08. Add one more metric: Average number of requests in the queue.

For metrics that were calculated earlier also - PLOT ON THE SAME GRAPH AS DECSERVER VERSION 1 AND 2 AND COMPARE THE PERFORMANCE. Add your observations with each graph.

Submit

1. The updated code and scripts in separate code files. Server code must be in separate files now.
2. A readme with your git repository details and a description of the division of **coding work. (** Division of work along coding/experiments boundary is *not acceptable.*).
3. The experiment results in a separate pdf

Lab 10 Autograder Version 4: Asynchronous grading architecture. ONLY DESIGN DOCUMENT due Sat Nov 4th.

Working Implementation will be due post-endsem.

So far we improved the performance of our server by going from single threaded (which has a thread bottleneck), to multi-threaded with a thread-per-request model with no limit on the number of threads (which can lead to a hardware bottleneck in terms for excessive contention for CPU and memory), to multi-threaded with *pools* so that we limit the number of threads. With an appropriate setting of the number of threads, this can be satisfactory.
However, the autograder application has a unique design problem: the student code submitted for grading can have a completely unpredictable runtime. This can result in clients timing out during high load conditions. The only solution to this is that the clients should not *block* on the grading request - i.e. it should not wait for the entire compile-run-grade process to be done. The grading can be done *asynchronously.*

Thus the goal of this version is to turn the server architecture into an asynchronous grading architecture, as follows:

When a grading request arrives, the server first simply *queues the request for grading* and sends a response to the client stating that request is accepted and is being processed for grading. This results in the client's 'recv/read' call getting a quick response, and the client considers this request-response cycle as complete. The client then sends a 'check status' request to the server later, to check if the grading request is complete. Whenever this request is sent, the server sends some progress response.

Note that each time the *client continues to send a blocking request.* The way we are making this 'asynchronous' is that the client is no longer waiting for the *grading task* itself. The grading task is what is happening 'asynchronously'.

For all this to now work in connectionless, stateless setup, we need a way to identify *which grading request* is the client asking status for. For this we will implement a concept of a 'request ID', which the client uses, when it enquires status. The program usage is updated as follows:

```
./submit  <new|status> <serverIP:port>
<sourceCodeFileTobeGraded|requestID>
```

Now the 'submit' program takes one more argument: the string 'new' or the string 'status'. If the argument is 'new' the client is sending a new request and the 3rd argument should be the filename. If old, the client is checking the status of an old request and the 3rd argument should be the requestID. Suppose a new request is sent as follows:

In either case, the server will send one of the following responses:

- `Your grading request ID <requestID> has been accepted and is currently being processed.`
    - This response is sent if the request has been picked up by a thread for processing.
- `Your grading request ID <requestID> has been accepted. It is currently at position <queuePos> in the queue.`
    - This is if it is still in the queue.
- `Your grading request ID <requestID> processing is done, here are the results:`
    - `//send the same results as before.`
- `Grading request <requestID> not found. Please check and resend your request ID or re-send your original grading request.`

The server architecture now needs to be updated in various ways

- You will need some way to generate a unique request ID.
- You will need data structure and storage design to
    - Associate the request ID with the grading request to be able send the correct status response of queue position, or 'in process'
    - Save the results - preferably not just in server process memory, but into secondary storage, so that even if the server restarts, it can retrieve the results of the request.
        - You then need to associate the stored results with the request ID so that they can be sent back when the status inquiry comes.

You may think of a design where you have some status information of recently done requests in memory, but older ones in a log file (your design should be resilient to the situation that a client gave a grading request a long time back and then 'forgot' about it and queried much later for it).

Performance Experiments

Since your client is now 'asynchronous' your measurement infrastructure needs to be updated.

- A successful response now is only the one which says '`processing is done`' - the others will be successful only in getting *some* response but not the 'grading done' response.
    - After the 'submit new' you will have to wrap the 'submit status' request in some loop with some 'polling interval' after which the client keeps 'polling' the server for status until it gets the 'done' response.
- Response time should be time from sending the 'submit new', until getting the 'done' response. Note that this will become a function of the polling interval that you choose to keep checking with the server.

Write up the details of the entire server design in slides form (GOOGLE SLIDES ONLY) and submit it on the due date.

---

*After this: **Work in progress** - this is just a teaser of possibilities, if you start this, note that some specs may change. Start at your own risk 🙂*

```
//Possibly containerization - will flesh out later
```