CS 744: Design and Engineering of Computing Systems
Autumn 2023

Project DECServer

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*

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.

---

*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* 🙂

1. Version 2

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. There should be configurable limit of how many worker threads the server can have active.
Do the same checks 1-4 as above. Do some what-ifs with the number of worker threads.

Additionally, check what happens if some of the submitted programs are too long (add some dummy computation to some of the programs)

2. Version 3

Turn the server architecture into an asynchronous architecture, as follows:

- When a grading request arrives, the listener thread does one of the following:
    - If a worker thread is free, it gives it this request to grade. It also generates a unique request ID. It does not block on this thread. Instead it immediately responds back to the client:

```
$./submit  <serverIP:port>  <sourceCodeFileTobeGraded>
Your grading request ID <requestID> has been accepted and is
currently being processed.
```

    - If no worker thread is free - it simply queues the request into a common queue, gives the request a unique ID, and responds back to the client as follows:

```
$./submit  <serverIP:port>  <sourceCodeFileTobeGraded>
Your grading request ID <requestID> has been accepted. It is
currently at  position <queuePos>  in the queue
```

- The worker threads should now write the result into some file (instead of sending it back to the client).
- Write a separate status-check client which is invoked as follows, and gets a response as above if the grading request is not done, or the usual grading response described earlier.

  I.e. if the request is still in queue:

```
$./checkStatus  <serverIP:port>  <requestID>
```

- Your grading request ID <requestID> has been accepted. It is currently at position <queuePos> in the queue

  If the request is being processed:

```
$./checkStatus <serverIP:port> <requestID>
```

- Your grading request ID <requestID> has been accepted and is currently being processed.

If the request is done:

```
$./checkStatus <serverIP:port> <requestID>
Request <requestID> processing is done, here are the results:
   1. PASS
   2. COMPILER ERROR
   3. RUNTIME ERROR
   4. OUTPUT ERROR
```

With the additional information as described earlier.

Do the same experiments as described earlier. Do some what-ifs with the number of worker threads.

3. //Possibly containerization - will flesh out later
4.