



# DECS Asynchronous Server Design

Akshay Patidar and Ravi Patidar

# Index:-

- Tools
- Libraries used
- Database schema
- Approach to generate unique Request ID
- Approach to handle new connection request
- Approach to async evaluation
- Performance Analysis
- Comparison graphs

# Tools

- Database : PostgreSQL
- IDE : VScode
- Graphs: Pyplot
- Load scripting: Bash

# Libraries:-

- libuuid: Used to generate unique Request ID.
- libpq-dev: Used to connect C code to postgres.
- libpthread: Used for thread implementation.
- socket: Used for socket programming.

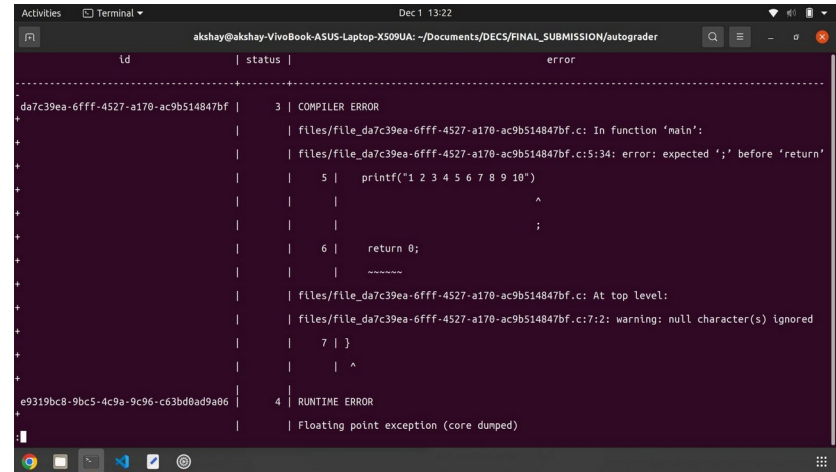
# How to run server

Run setup.sh script:

It will install all dependencies required for our project and compile the gradingServer.c file

# Database Design:

- Database: autograder
- Tables
  - Grading\_requests
    - id (UUID):- used to store unique id of each request.
    - status (int) :- status of request
      - 0:- not evaluated
      - 1:- in queue
      - 2:- in process
      - 3:- compiler error
      - 4:- runtime error
      - 5:- output error
      - 6:- pass
    - error (text, default NULL):
      - - used to store compiler , runtime or output error



```
Activities Terminal Dec 1 13:22
akshay@akshay-VivoBook-ASUS-Laptop-X509UA: ~/Documents/DECS/FINAL_SUBMISSION/autograder

-----
| id | status | error |
-----
da7c39ea-6fff-4527-a170-ac9b514847bf | 3 | COMPILER ERROR
| | | files/file_da7c39ea-6fff-4527-a170-ac9b514847bf.c: In function 'main':
| | | files/file_da7c39ea-6fff-4527-a170-ac9b514847bf.c:5:34: error: expected ';' before 'return'
| | | 5 | printf("1 2 3 4 5 6 7 8 9 10")
| | | | | ^
| | | | | ;
| | | 6 | return 0;
| | | | | ~~~~~
| | | files/file_da7c39ea-6fff-4527-a170-ac9b514847bf.c: At top level:
| | | files/file_da7c39ea-6fff-4527-a170-ac9b514847bf.c:7:2: warning: null character(s) ignored
| | | 7 | }
| | | | ^
e9319bc8-9bc5-4c9a-9c96-c63bd0ad9a86 | 4 | RUNTIME ERROR
| | | Floating point exception (core dumped)
```

## Approach to generate unique Request IDs:-

- Timestamp : When multiple request comes, it fails to generate unique requestID.
- Timestamp + random number: In multithreaded environment in some cases it fails to generate unique requestID.
- UUID: It generates unique ID for each request.

## Approach to handle new/status connection requests:-

- We maintained a queue which stores the socket ID of connection.
- We created a thread pool which take a socket ID from queue. And accept the file from client if the request type is new.
- Generate a unique RequestID for this request.
- Then store the file in secondary storage with filename file\_<uuid>.c.
- Sends the unique requestId to the client, and close the connection.
- If the request is for checking the status. Then it will fetch the status from postgres database. And inform the client.
- Close the socket connection to that client.



## Approach to async evaluation:-

- We maintained a eval\_queue. which stores the request IDs of requests for grading.
- We created a eval\_masterFunc which fetches the request IDs from postgres database where status=0 and will enqueue it in eval\_queue. It handles the case if server is down and then starts again then it will resume from the same state.
- And there are eval\_thread pool which pick a request ID from evaluation\_queue and process it.
- The evaluation\_threads will update the status of request in postgres database.
  - When it take the request from evaluation\_queue. It changes the status to 2 process is in thread.
  - After evaluation change the status according to the result obtained
    - Compiler error-3
    - Runtime error - 4
    - Output error - 5
    - Successful - 6
- And stores the results (error or successful compilation ) in postgres/autograder database..

# Fault Tolerance

Upon server initialization, any requests flagged with a status of 1 undergo an immediate status change to 0. Subsequently, we retrieve requests from the evaluation queue, specifically filtering for those now marked with a status of 0. This systematic approach ensures that, in the event of server termination, requests initially set to a status of 1 are seamlessly transitioned to 0 and subsequently reintegrated into the evaluation queue for processing.

# Communication between client and server

1. Client connects to the server to send a new request to the server.

Command: `./client new serverIP:Port sourceCodeFileToBeGraded`

The Server accepts the request and generates a unique identifier “Req ID” for that request. Server accepts the file send by client with the requestID generated for that client. Server informs client about the request acceptance and send Req ID to the client.

# Communication between client and server

2. Client connects to the server to check the status of its request

Command `./submit status serverIP:port requestID`

Now clients again connects to the server and sends the Req ID given by the server to check the status of its request. Server uses this Req ID to get its status from the database. If the request is still in the Queue, server gets the location of the Req ID and sends it to the client. If the server does not find any entry corresponding to the Req ID in the database it considers that Req ID as invalid and tells client to resend the request.

# Server Responses

**“Request accepted” and sends Req ID:** this is send when a client send a new request to the server and it is successfully accepted.

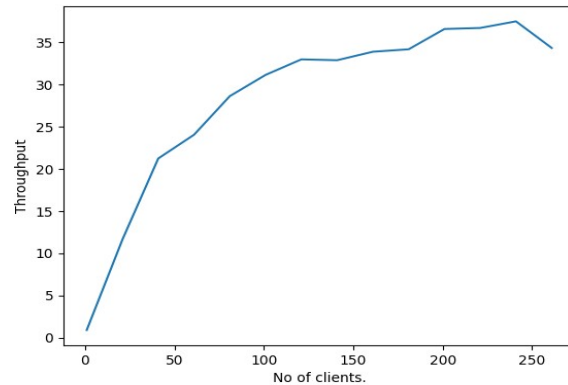
**“Request in Queue” and position in queue:** this is send when the client reconnects to check the status and the request is still in the queue and not picked by any thread.

**“Request in Progress”:** this is send when a thread picks the request from the queue and is working on it.

**“Request Completed” and sends the corresponding response:** when the request is completed and stored in databases and server sends the response as contents of that error column in grading\_request table.

# Performance Analysis

Throughput vs No. of clients

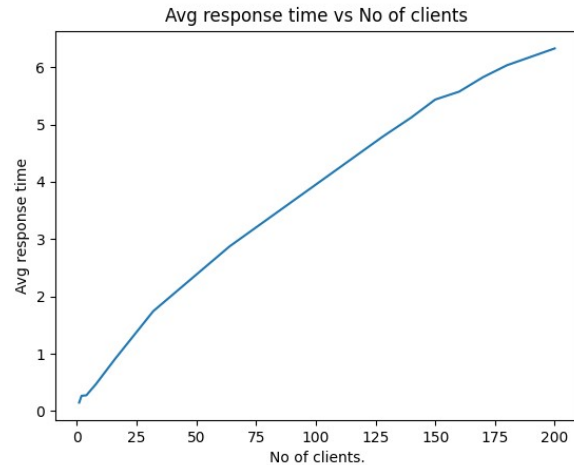


Compilation time of grading code: 0.110 sec

No. of cores: 4

Ideal max throughput:  $4 \times (1/0.108) = 36.36$

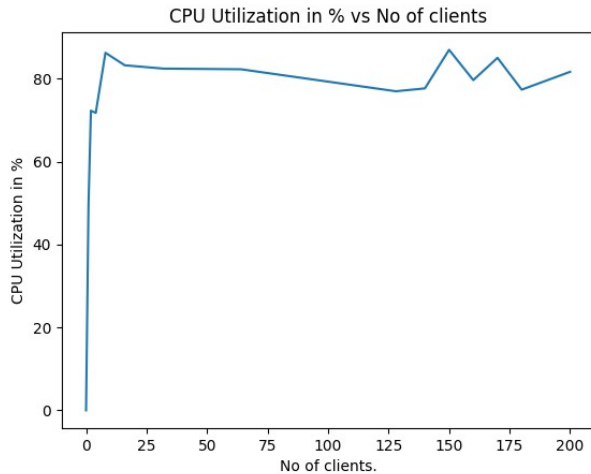
# Performance Analysis



Avg response time increases with number of clients as expected.

This can be attributed to resource contentions, queuing delays, caching ineffectiveness etc.

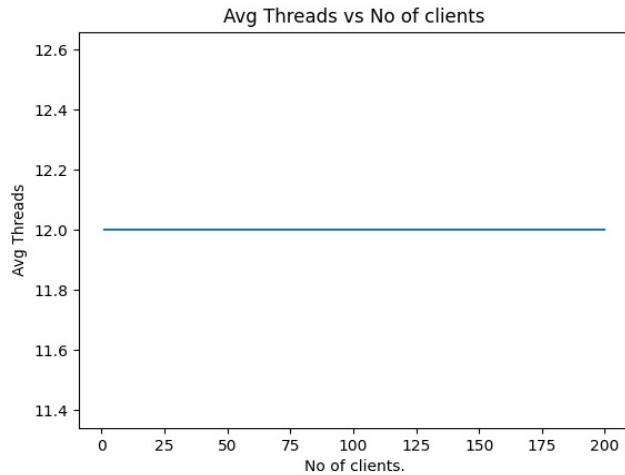
# Performance Analysis



Avg CPU utilization increases with number of clients as expected. This can be attributed to increased workload, context switching overheads, processing overhead for each request etc.

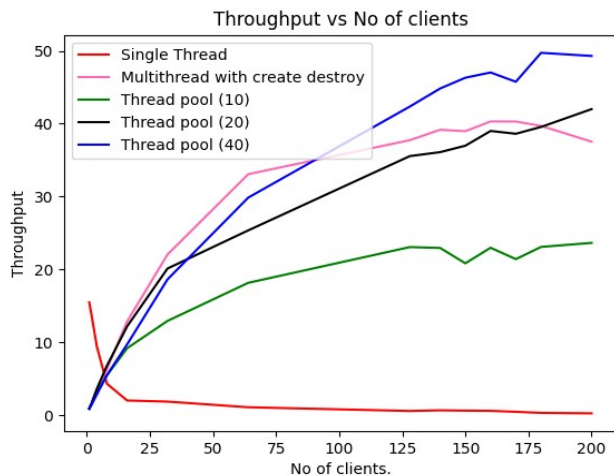


# Performance Analysis



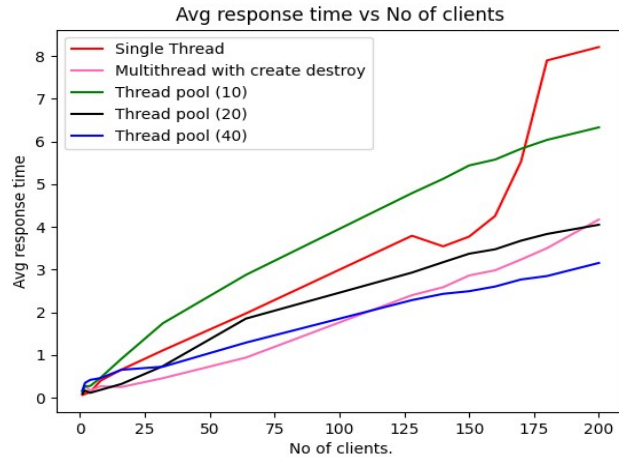
Number of Threads = 10 + 1 main thread + 1 grep command thread

# Comparison graphs



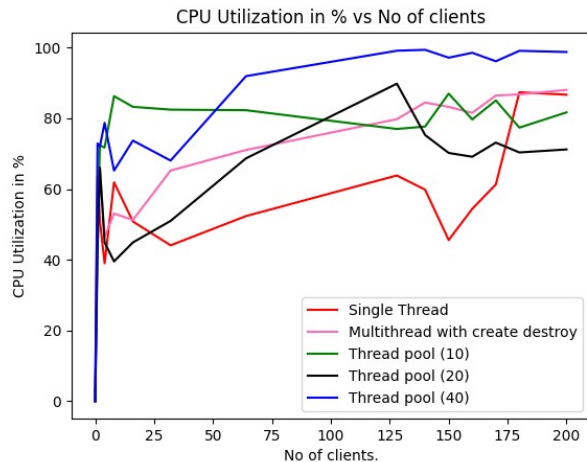
Single Threaded: Contention is due to the BACKLOG parameter, leading to reduction in throughput overall and stabilizing at a relatively low value. With increase in the number of threads in the threadpool, with the limitation of CPU cores, we see a reduced maximum throughput because of increased overhead, as mentioned in previous slides. As the number of request increases the throughput increase shows that out system is reliable with nearly 63%.

# Comparison graphs



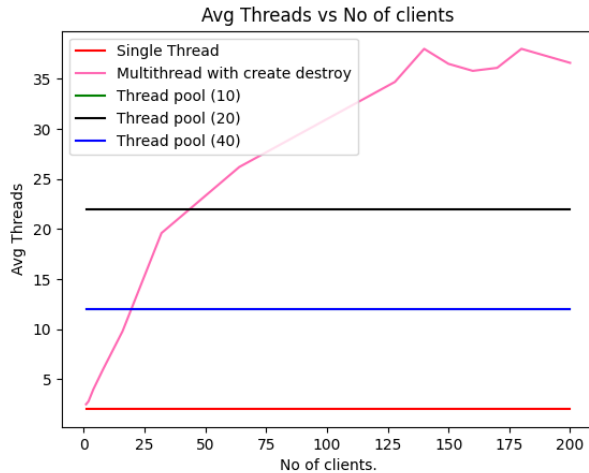
After sufficient number of clients, the service time stabilizes, although the number of requests keep on increasing, and hence we see a linear trend, because of linear waiting time. With the asynchrononous design the client wait time reduced to 93% as the client will get a request ID and can exit. Ans, leads to decrease in timeouts ~86%.

# Comparison graphs



The single threaded version stabilizes after a small value since the load becomes constant. Similarly, for the other threaded versions, the utilization stabilizes for a larger value, but shows a largely similar trend for all threaded versions. The CPU utilization improves nearly 62% as compared to single threaded server.

# Comparison graphs



For the threadpool versions, the number of thread is constant, while for the “New Thread per request” version, the thread count increases linearly with the number of requests. The number of threads grow when requests comes it shows scalability around 71%.

For single threaded version, it's stable at 1

Thank you