Decs LabAutograder

Team Member Names:
Vivek Pawar(23m0769)
Jayesh Bangar(23m0805)

Version 1:Single Threaded Server

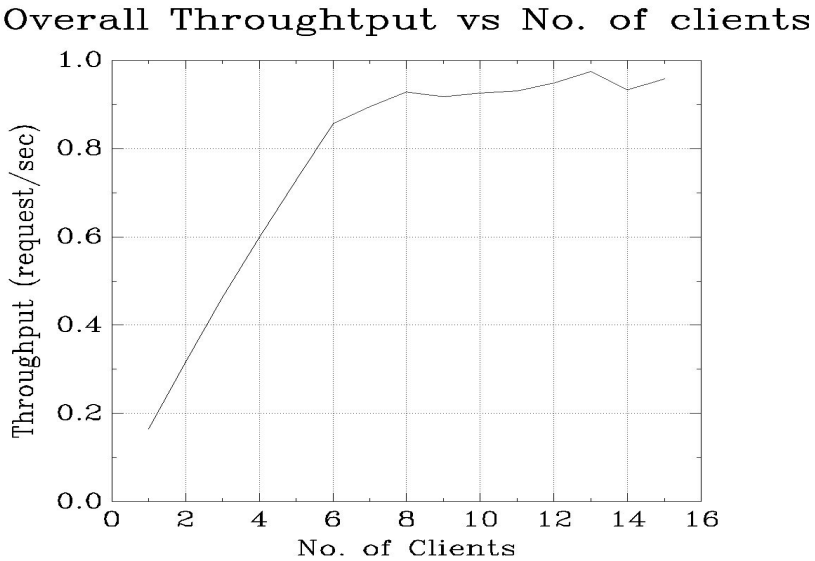
- In Version 1 of the Single Threaded Server code, a basic server functionality is implemented to listen for incoming client sockets.
- Clients create a socket to connect to the server, sending a file for grading.
- Upon receiving the file, the server grades it and sends the result back to the client.
- The program is compiled and executed using the system command, and the output is compared to the expected output.
- If a difference is detected, the server uses the diff command to identify and send the discrepancies between the actual and expected output files to the client.
- After a successful response is received, the client sleeps for some time, which is the think time.
- We’re calculating the No of successful responses, total timeouts.
- We’re also calculating the Average response time, the request sent rate, the goodput, the timeout rate and the error rate in order for us to perform performance analysis on our server.

Version 1:Single Threaded Server

- **Design choices**
- The server also creates multiple directories in order to store the various intermediary outputs in separate directories.
- Then server assigns a unique id to each grading request using the rand() function and appends the id to the various intermediary files and saves it to the respective folder for the next step.
- Example, the server uses the following method to create a unique filename for a grading request.
- The same filename will be used for compiling, executing the request.

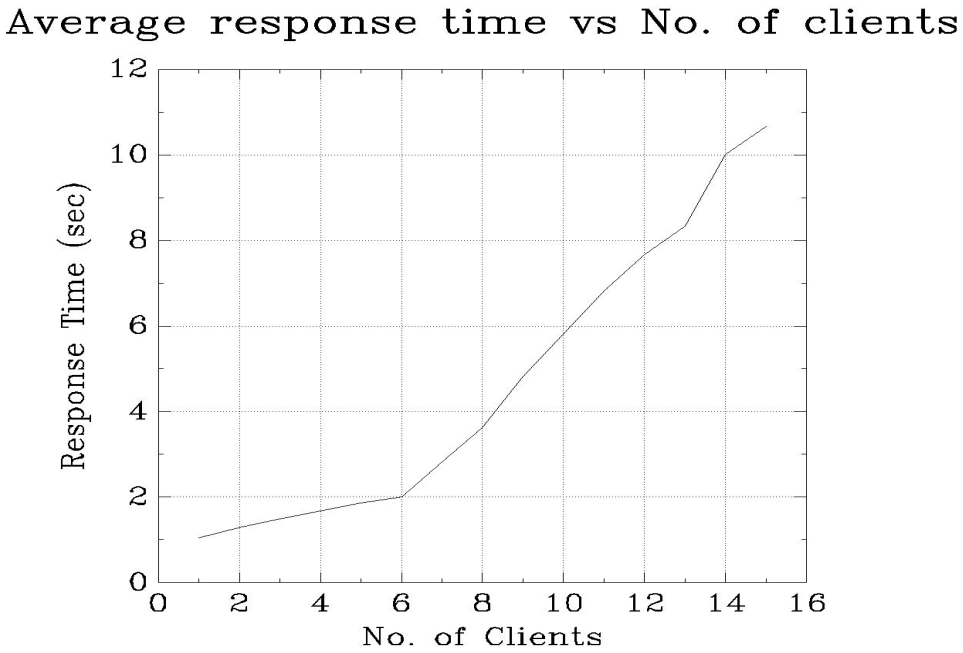
```
string filename = SUBMISSION_DIR + to_string(id) +"file.cpp";
```

Version 1:Single Threaded Server



For a single response, avg response time is 1.05 sec. Therefore max throughput can be achieved is 0.95 Our system is achieving throughput close to that number.

Version 1:Single Threaded Server



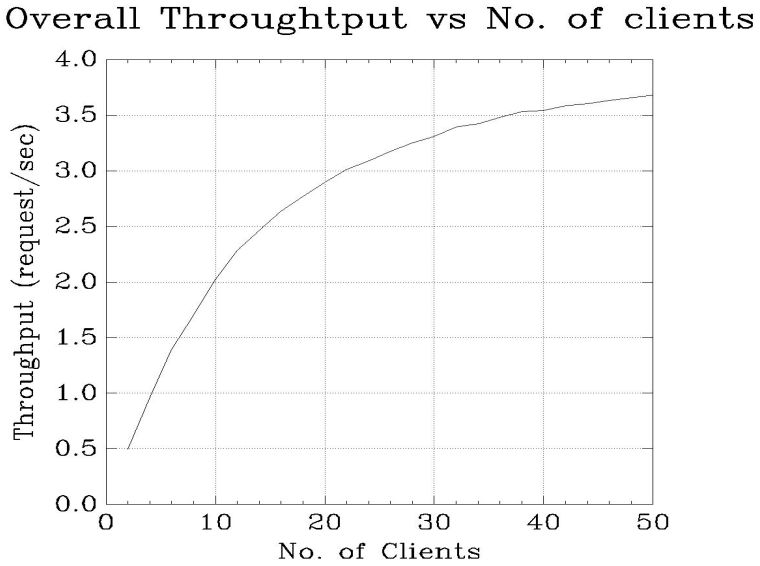
Version 2:Multi-Threaded Server with Create-Destroy threads

- In the enhanced version of the server (Version 2), multithreading capability has been introduced.
- The server now includes a listener thread responsible for accepting grading requests.
- For each incoming request, a worker thread is dynamically created to process the request independently.
- Each worker thread handles one autograding request, directly writing the response back to the client upon completion.
- The client has been updated to include a timeout mechanism, enhancing its robustness during communication with the server.
- This implementation improves overall system efficiency by allowing concurrent processing of multiple grading requests.

Version 2:Multi-Threaded Server with Create-Destroy threads

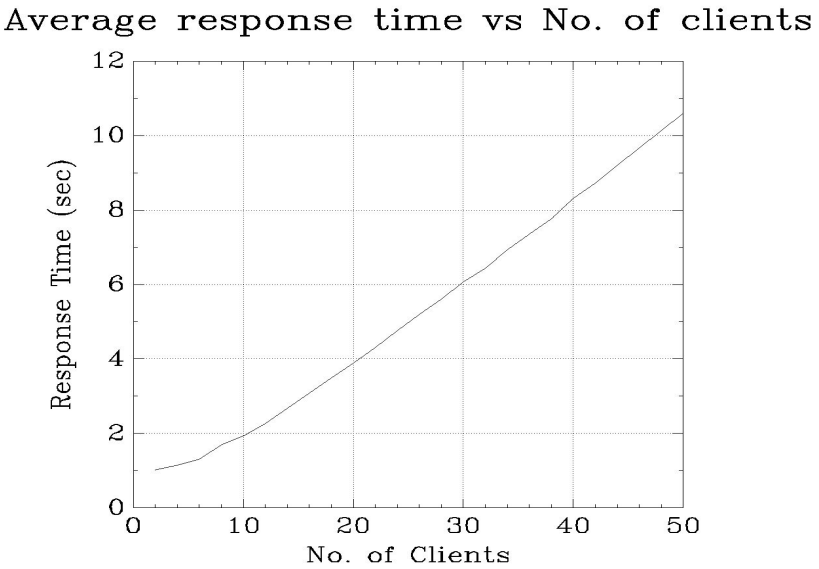
- **Design choices**
- The entire client logic remains the same as the Version 1.
- The Server now accepts every request in a while loop and creates a client_socket for it.
- It also creates a thread specifically for handling every grading request and the grader function executes the grading of the specific file.
- In order for releasing the threads system resources back to the system without the need for another thread to join and clean it up, we’re using pthread_detach() function.

Version 2:Multi-Threaded Server with Create-Destroy threads

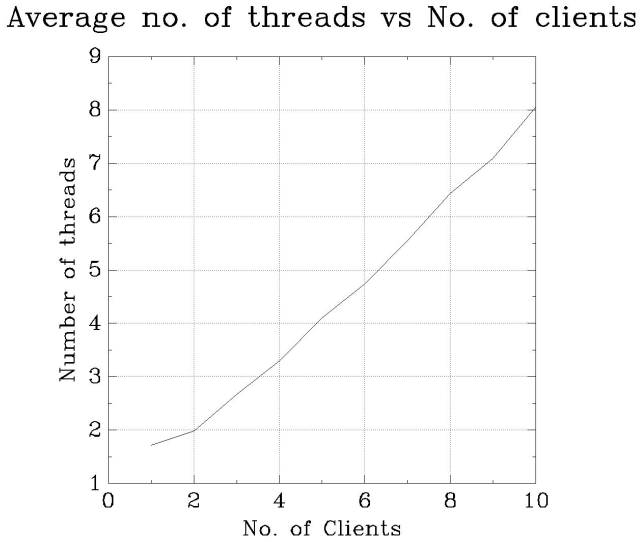


As no of clients increases the throughput stars increasing , and reaches a saturation level. In the multithreaded server, were unable to achieve the maximum throughput due to the context switching overhead at the cpu level

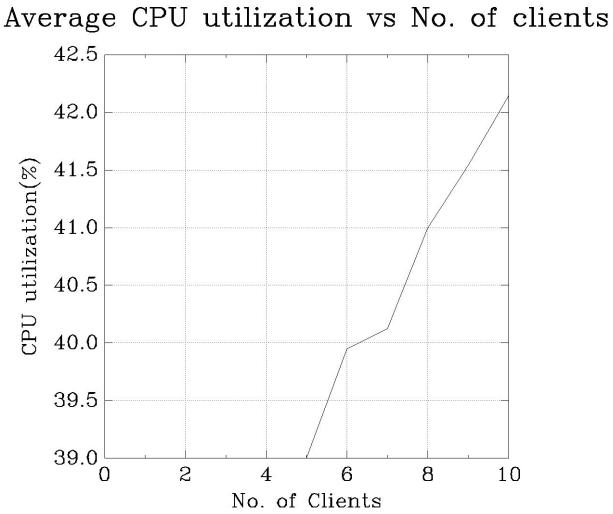
Version 2:Multi-Threaded Server with Create-Destroy threads



Version 2:Multi-Threaded Server with Create-Destroy threads



Version 2:Multi-Threaded Server with Create-Destroy threads



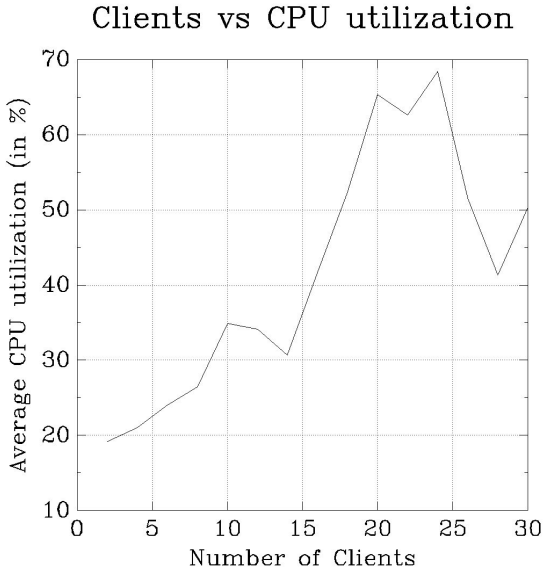
Version 3: Thread Pools

- Now, a thread pool is created by the main thread at the beginning, with a specified size (thread_pool_size).
- These threads start and wait for grading requests. To facilitate communication between the main thread and worker threads, a shared queue is introduced.
- The main thread enqueues grading requests (connections) into this shared queue.
- Mutexes are utilized to ensure thread-safe access.
- Additionally, condition variables are employed to avoid the inefficiency of constant checking of the queue by worker threads in a busy loop.
- Worker threads can wait on a condition variable, and the main thread signals them when a new grading request is enqueued.
- This is a signaling mechanism.

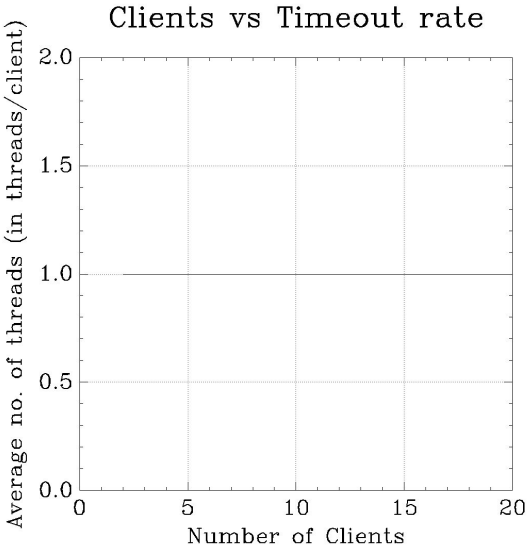
Version 3: Thread Pools

- **Design choices**
- The client remains the same from the previous versions.
- In the server code, we're asking for a thread pool size from the user and creating a thread pool of worker threads of the given size.
- We're creating a Shared queue.
- Then we're accepting incoming client connections on a server socket.
- Before accessing the shared queue, a mutex lock (pthread_mutex_lock) is applied to ensure thread safety. This prevents multiple threads from accessing the queue simultaneously.
- After enqueueing the client socket descriptor, the mutex is unlocked (pthread_mutex_unlock).
- A condition variable (pthread_cond_signal) is used to signal that a new item has been added to the queue.
- The worker thread, after waking up from the condition variable signal, proceeds to dequeue an item from the shared queue using the dequeue function. After dequeuing, the mutex is unlocked (pthread_mutex_unlock).

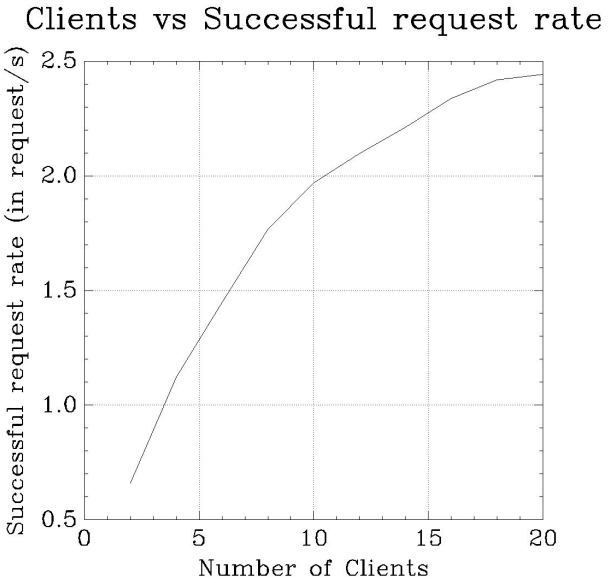
Version 3: Thread Pools



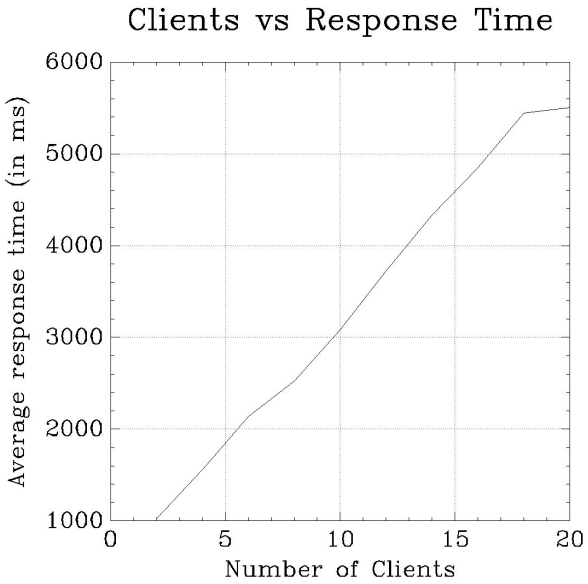
Version 3: Thread Pools



Version 3: Thread Pools



Version 3: Thread Pools



Working of the client

1. **Argument Validation**:
    - Client checks for valid arguments: "new" or "status."
    - If "new" is chosen, the client must include the source file name.
    - If "status" is chosen, the client must provide the requestID from the server.
2. **Submitting a New File for Grading**:
    - If "new" is the chosen status, the client:
    - Establishes a new socket connection with the server.
    - Sends the file to be graded to the server.
    - Receives a "File sent successfully" message from the server.
    - Obtains a requestID from the server for tracking the grading request.

Working of the client

3. **Checking Grading Status**:

    - If "status" is the chosen status, the client:
    - Sends the requestID to the server.
    - Server processes the request, retrieves grading status or results.
    - The server sends the grading status or result back to the client.

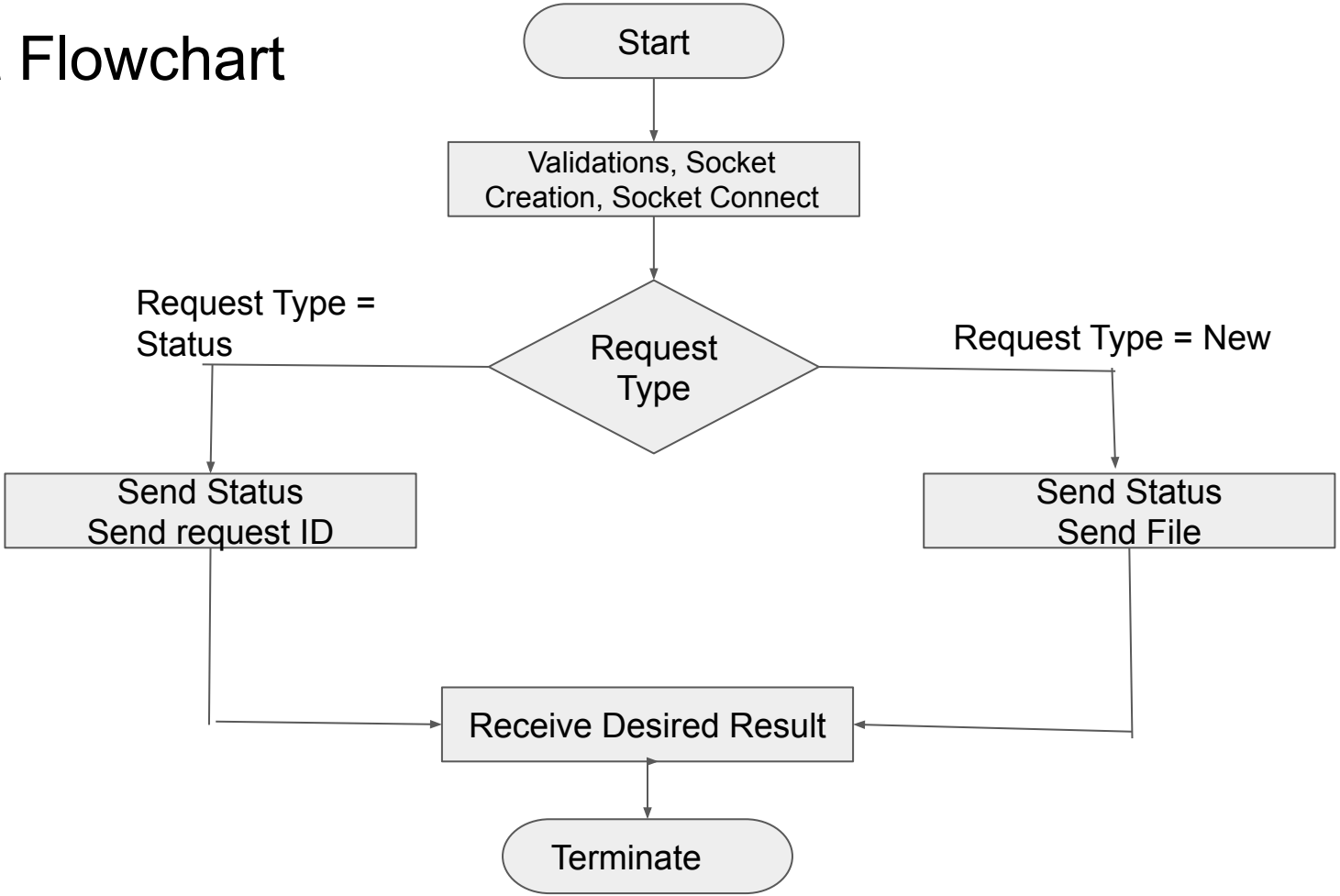
Autograding Server

For this iteration of the autograding server, we've implemented an asynchronous server approach. It receives client requests for evaluation. Instead of keeping the client waiting for an immediate response, the server now provides the client with a unique request ID. This request ID allows the client to later check the status of their submitted request.

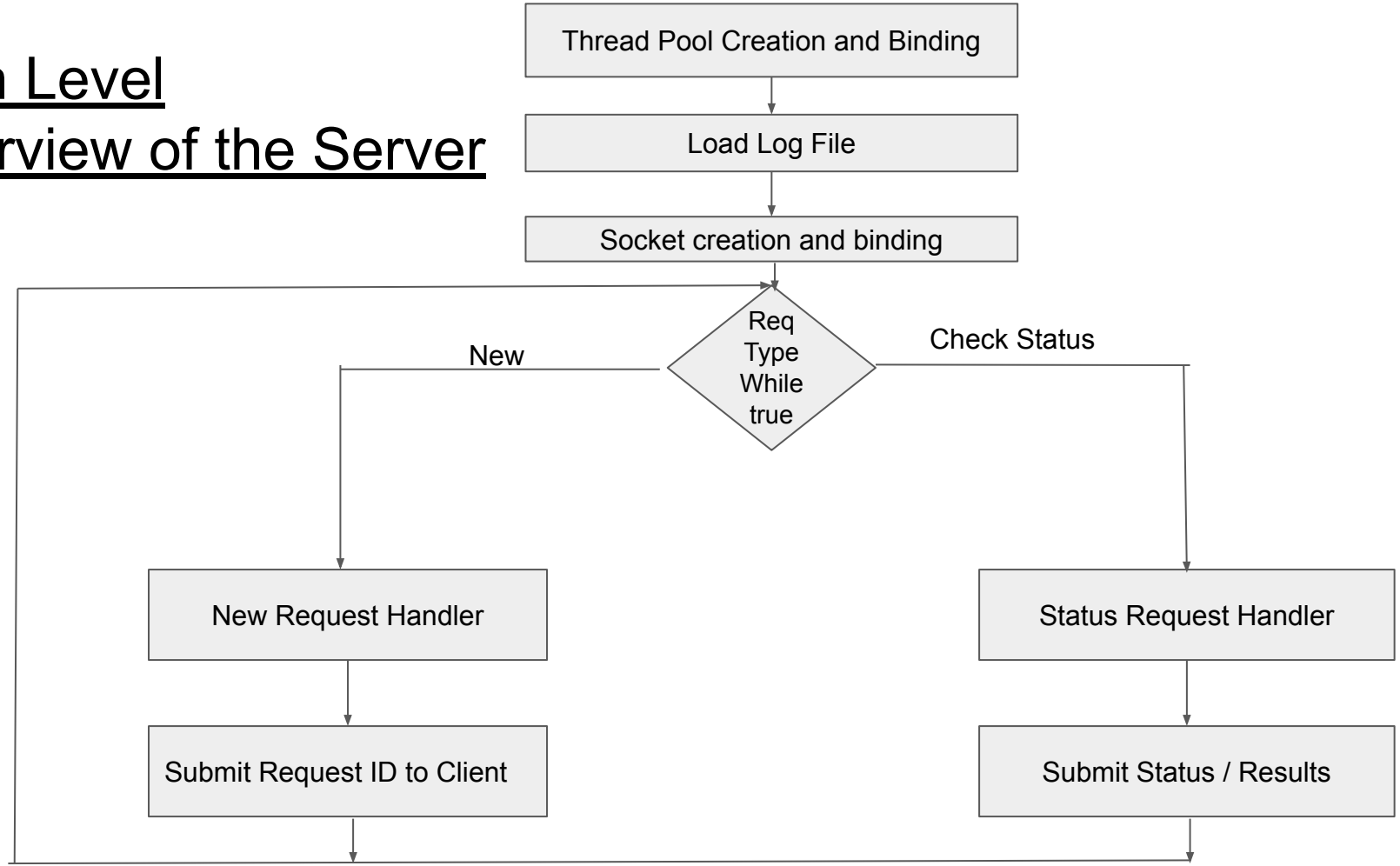
In this setup, the server maintains the request IDs and their corresponding statuses in a hashmap and a log file. Additionally, our server is equipped with specialized modules to manage new requests, handle status inquiries, and oversee the grading process.

The provided flowchart offers a high-level overview of the grading server's operation. Please note that the flowchart does not include details about the grader module. The grader module operates in the background and manages requests as they are encountered in the request queue.

Client Flowchart



High Level Overview of the Server



Thread creation and Initialization module:

The primary purpose of this module is to establish a thread pool for managing grading requests efficiently. It involves the following key functionalities:

1. **Thread Pool Creation:** The module is responsible for creating a thread pool to handle grading requests.
2. **Thread Initialization:** During the server's startup, the module initializes and creates a specified number of threads. The number of threads to be created is typically provided as input when starting the server.
3. **Resource Management:** In addition to thread initialization, the module also initializes lock variables and conditional variables. These synchronization mechanisms are essential for maintaining thread safety when multiple threads access shared resources.

Load Requests Logs Module:

The module is designed to address server reboots and handle pending requests.

The module loads log entries from a log file stored on the server's secondary storage into an in-memory hashmap.

1. **Log Entry Creation:**

Entries are added to the log file whenever there is a change in the request status.

2. **Loading Entries:**

Entries from the log file are loaded into the in-memory hashmap.

Concurrently, requests that have already sent their results to clients are removed from the map.

Load Requests Logs Module:

3. Additional Processing:

- Requests with a status of 'queued' are enqueued into the request queue of the grader.
- Requests with a status of 'processing' are scheduled for processing, rather than being enqueued.
- Requests with a status of 'processing done' but with unsent results are retained in the hashmap.
- Requests with a status of 'processing done' and results already sent to clients are removed from the hashmap.

The Load Request Processing Module enhances the server's resilience by managing request states during server reboots.

New Request Handler Module:

The primary purpose of this module is to manage and process incoming grading requests received by the server.

1. **Request Handling:**

   This module serves as the initial point of contact for newly received grading requests. Whenever the server receives a new request, it is directed to this module for processing.

2. **Unique Request ID Assignment:**

   Upon receiving a new request, the module assigns a unique request ID to it. This unique identifier is crucial for tracking and distinguishing individual requests, ensuring their unique identification within the system.

New Request Handler Module:

3. **Submission File Management:**

   Following the assignment of a request ID, the module receives the submission file from the client. The module stores the submission file in secondary storage, ensuring that it is readily accessible for later use by grader threads during the evaluation process.

   To facilitate seamless matching and identification, the module associates the submission file with the unique request ID.

4. **Status Update:**

   Once the submission file is completely received and stored, the module updates the status of the request in both the hashmap and the log file concurrently. This ensures that the request's status is accurately reflected in the system, and the log file maintains an up-to-date record of request changes.

Grader Module

The primary purpose of this module is to perform the evaluation of submissions and save the evaluation results.

1. **Evaluation of Submissions:**

   This Grader thread is responsible for the evaluation of submitted content. It repeatedly checks the status of the request queue created by the server to maintain the requests in the system.

2. **Request Queue Monitoring:**

   The Grader thread continuously monitors the request queue. If the thread finds the request queue to be empty, it enters a sleeping state. The thread can be awakened from this sleep state when a signal function is called on the conditional variable associated with the thread.

Grader Module

3. **Processing Requests:** When the Grader thread identifies a request in the queue, it dequeues that request and updates the request status in the hashmap and logfile to 'processing.' The thread then retrieves the corresponding submission from the secondary storage and proceeds to evaluate it.

4. **Result Storage:** After the evaluation is complete, the Grader thread stores the evaluation results in a result file. This result file is associated with the unique request number, ensuring a link between the request and its evaluation outcome.

5. **Status Update:** Once the evaluation is finished and the result is stored, the Grader thread modifies the status of the request to 'completed' in both the hashmap and the log file simultaneously. This keeps the system's records up-to-date.

6. **Client Response:** The evaluation result can be sent to the client upon request. The Status Request Handling Module handles this task and provides the result to the client whenever the client initiates a request for it.

Status Request Handler Module

The module takes a RequestID as input.

1. **Checking RequestID**: It checks if the RequestID exists in a Hashmap containing request statuses.

   If not found, it sends a message to the client: "Incorrect Request ID, Please send a valid requestID."

2. **Queue Status**: If the status is "queued," the module returns the client's queue position.
3. **Processing Status**: If the status is "processing," it sends a message to the client: "The request has been picked up by a thread."
4. **Completed Status**: If the status is "completed," it sends the grading results back to the client.

The module handles status requests for a given RequestID, checking if it exists, and responding with the appropriate status or messages based on the status of the request.

Request Id Generation

1. The unique request ID is generated by appending the following two components:
   a. Current Timestamp: This provides the date and time information, down to a precise level, ensuring uniqueness.
   b. Thread ID: This is the identifier of the thread that is handling the grading request.
2. By combining the timestamp and thread ID, the resulting request ID is 12 digits long, ensuring it is unique for each grading request.
3. This unique request ID is then provided to the client, allowing them to track the status of their grading request and enabling the system to manage and identify individual grading requests accurately.

Design and Implementation of HashMap:

The Hashmap is structured with RequestID as the key and a corresponding Status Flag as the value. The Status Flags have the following meanings:

- If the value is 1, it means the request is still in the queue, and the module will determine its queue position and return it.
- If the value is 2, it indicates that the request has been taken up by a processing thread.
- When the value is 3, it signifies that the processing is complete, and the module will return the grading result from a file with a unique name formed by concatenating the RequestID and "results.txt."

Key(Request ID)	Value(Status Flag)
1217282	1
1268271	2
1727512	3