

Distributed Project - Report

### Abdullah Nashat Salman - 900143140

### Ahmad Shaaban - 900171600

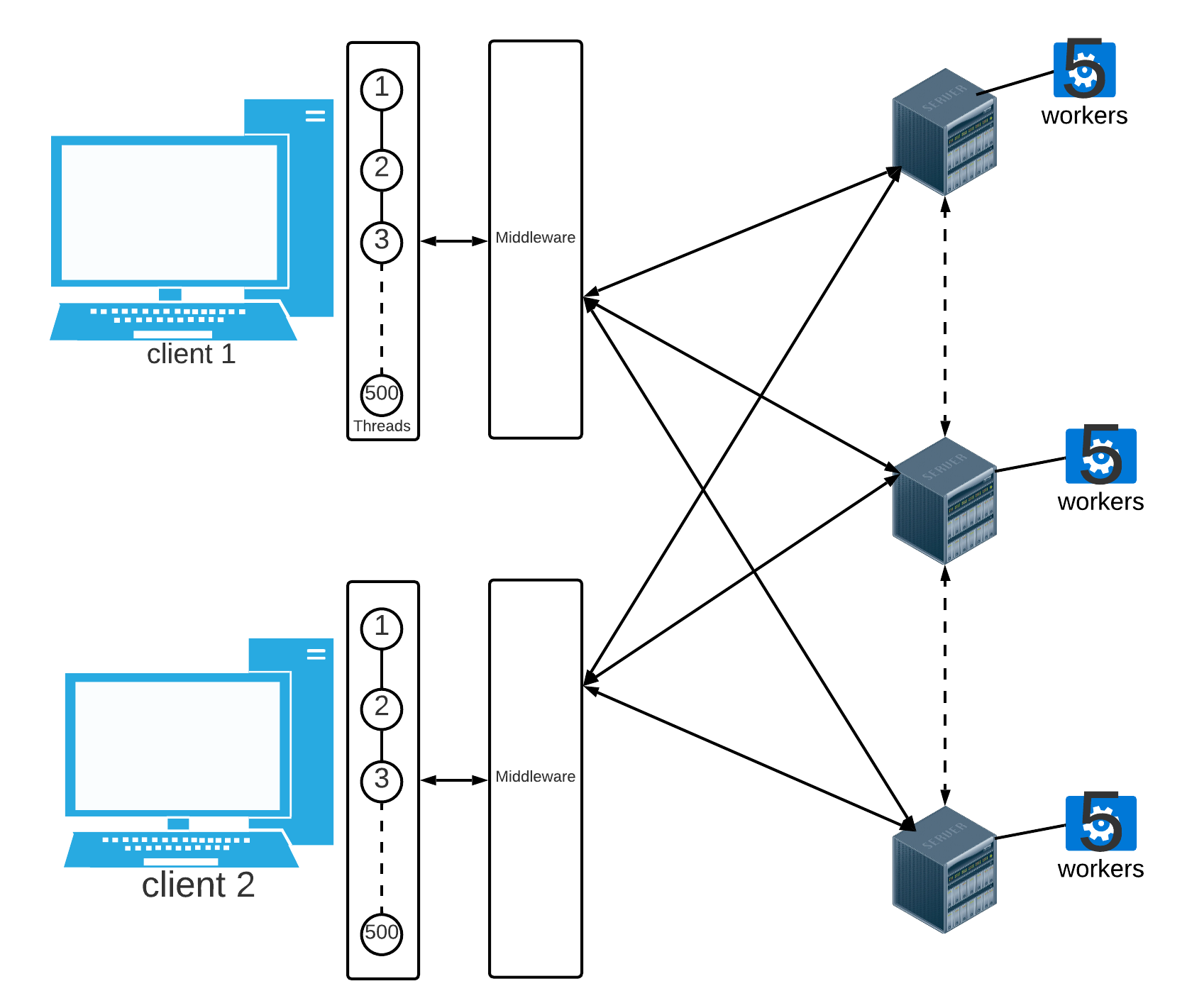
### Abdelrahman Reda Abdelmonem - 900192706

### 

### 

# 

# Design



A Simplified diagram describing our system showing the different nodes & their connections.

### 

### **Client Side:**

The Client is implemented as a **single process that creates 503 threads**.

* *500 threads* represent the different client applications,
  + Each thread executes an infinite loop that sends a message to the middleware thread and waits/blocks for a reply.
  + Every 20 seconds, a thread will send a message to the logging thread with its info such as (number of requests it made, latency, and so on ..).
* *2 middleware threads*
  + The First thread accepts incoming messages from the client threads.
    - This thread sends requests and waits for a reply from a server (using round robin).
    - Timeout is implemented in the case that the server doesn’t reply within 3 seconds.
    - After receiving a reply from a server, the middleware thread replies to the client thread. In case there is a timeout, the middleware thread replies to the client thread with a failure message (-1).
  + The Second thread accepts incoming messages from other middleware agents (servers).
    - This is specifically made in case there is an announcement that a server is down. In which case this thread updates a list of which servers are available and which are down.
* *1 logging thread*
  + This thread is responsible for logging in the thread metrics and calculating averages and total sums, it writes this data to a file.
  + This operation is done whenever this thread receives updated info from a client thread which is done every 20 seconds.

All communication between the different threads is done by rust native objects called Channels. Channels allow for synchronous and asynchronous communication between the different threads. It also allows for multiple producers and a single consumer (senders can be cloned to other threads).

### **Server Side:**

The Server is implemented as a single process that has n+3 threads

* *n threads* represent the worker threads spawned at start time
  + N is a constant that defines the number of workers, we used 5
  + Each worker executes an infinite loop that blocks on a channel where jobs are queued. When it gets a job it executes it then returns to blocking on the channel.
  + The receiving end of the channel is shared between all workers so it has a mutex lock on it. One worker will block on the channel waiting for a job while all other workers will block on the mutex to get the channel.
* *1 listener thread (main)*
  + This is the main function
  + Blocks on get\_request function, waiting for requests
  + When request is received, it is pushed onto the channel for the workers to execute
  + Special case request for down-announcement
    - Announcement message includes ID of down server
    - If my\_ID = down\_ID (sleep for SLEEP\_TIME)
    - Stops the election trigger thread to assure only 1 server can be down at any time
  + Measures load as requests/sec by counting received request in a time window
  + In threshold load balancing, multicasts to servers to update load
* *1 election trigger thread*
  + Attempts to start election every TRY\_START\_ELECTION\_FREQ msec
  + Has START\_ELECTION\_PROBABILITY to start election
  + If election started, sends an election message to next server to start election
* *1 logging thread*
  + This thread is responsible for logging the server load and when it goes down
  + This operation is done whenever this thread receives updates from the listener about the current load which, under normal operation, happens every 1 sec.
  + The log is stored in a file for each server as lines each stating the time of this log and the load
  + The log also contains entries about when the server goes down
  + The log holds only the 100 most recent entries and old entries are overwritten

Some variables (ie., server ID, load, state of all servers) are necessary for some threads to do their job. Therefore, we define these variables with Ack pointers to allow access across different threads. We also use a mutex on the Ack pointer if any threads will write to this variable.

# 

# 

# Algorithms

## Load Balancing

#### **Round Robin**

We implemented a round robin algorithm for load balancing & sending messages. This was done at the beginning for rapid testing, simplicity of implementation and it seemed appropriate in our project case since we are mostly dealing with homogeneous systems in terms of hardware and software.

Round robin is quite straightforward, each client middleware thread has an index which loops over the server list. If the server is available (by checking its local list of available servers) it will send the request, otherwise it will move on to the next server.

#### **Round Robin + Threshold**

After implementing round robin, we implemented a threshold algorithm on top of the round robin in order to make a more intelligent system. The idea of the algorithm is as follows: if a server receives a request and is overloaded, it will forward that request to another server which is not overloaded. In case it can’t find any server it will handle the request itself.

Each server node can have one of three load levels (0,1,2) indicating how overloaded the server is. The load of the server is measured by the number of requests per second it receives. Where:

* 0 underloaded 0 - 5000 req/sec
* 1 medium 5000 - 7500 req/sec
* 2 overloaded 7500 - inf req/sec

The server every one second will check and update its local status, in case its status has changed, it will multicast a message of type load balancing to all other servers of its new status. This way each server node has a somewhat updated local list of the server status.

* This list can get outdated in the case that a server is down, its load will not be changed or updated till it wakes up again and sends to the other servers its new status.

Distributed Election

#### **Ring Algorithm:**

To choose which server will go down, we implemented the ring distributed election algorithm. We ordered the servers in a logical ring with each server having a “next” server. The election trigger thread on each server has a set chance START\_ELECTION\_PROBABILITY to trigger an election process every TRY\_START\_ELECTION\_FREQ msec. If an election is triggered, the election message starts traveling along the logical ring with each server appending its ID and a randomly generated value.

When a server receives an election message where the ID of the first entry is its own ID, this server will know that this election message has traveled along the entire ring and collected the IDs and values of all servers. It then finds the maximum values and its corresponding server ID. This ID is then multicast to all servers and clients as an announcement message.

When an announcement message is received by a server, it sleeps the election trigger thread to assure that only one server can be down at any point in time. The server will then check if the ID included in this announcement message is equal to its own ID. If so, It will sleep the listener for SLEEP\_TIME.

# Assumptions & Limitations

* Client:
  + Each client app/thread will send to the middleware thread and block for a reply.
  + Socket Receive Timeout = 3 sec. If the timeout is reached, the request will be considered a failure.
* Server:
  + Server has n=5 workers.
* Threshold:
  + There are three load levels: underloaded, medium, and overloaded
  + The load is defined as requests/sec
  + Down servers cannot multicast their load
  + Requests cannot be forwarded to downed servers
* Election:
  + Any server could start an election depending START\_ELECTION\_PROBABILITY and TRY\_START\_ELECTION\_FREQ
  + Only one server can be down at a time.
  + When a server is down, its listener stops listening to requests; however its thread workers will still reply to any present requests in the pool.
  + The downed server will be down for SLEEP\_TIME = 10 sec.
  + Announcement messages are broadcasted to every node (servers and clients.)

# Changes & Updates

### **Client Side:**

* Simple client application that sends and receives messages from a server using UDP.
* Implemented round robin to communicate with multiple servers.
* Implemented a request reply protocol on top of UDP by adding a struct for messages and doOperation method and marshaling and unmarshalling of messages
* Added Timeout and error handling in case of sending requests and receiving replies from a server.
* The client became multithreaded,
  + Created 500 threads for client applications,
  + Two middleware threads for communicating with other middleware agents.
    - A thread for sending requests and receiving replies from servers.
    - Another thread for receiving announcements from other agents such as the election announcement message.
  + Added another thread for logging performance and metrics and writing to a file.

### 

### **Server Side:**

* Created a simple server that handles requests and sends replies using UDP.
* Implemented request reply protocol on top of UDP, by adding handel\_request(), getrequest(), and sendreply() methods and the same struct for messages used by the client which is used for marshaling.
* Made the Server Multithreaded, 1 listener process and a worker pool with a jobs queue and 5 worker threads to handle requests.
* Added Election & Announcement logic, each server now has a separate thread that can trigger an election, sending election messages to the next server on the ring. It can use the election algorithm to decide which server will go down and send announcement messages to all nodes including the client agents.
* Added Threshold logic on the server side. By measuring the number of requests per second and updating its status and multicasting its status to other servers on a change in load.
* Added Logging code to log server status every 1 sec.

# Issues & Obstacles

We faced many issues & problems while working on the project, some were design related but a lot more were related to the code and difficulty of working with a new language especially rust. Rust seems to be a very strict language, this is done because Rust as a language is designed to prevent “bad” code that could crash the program or corrupt the memory (data races, null pointers, and so on …). While this is good as it encourages the developer to write more safe and cleaner code, it also makes the language appear to be very rigid and not flexible resulting in a not so friendly learning curve and making the process of writing a simple functionality more tedious and taking longer than it should.

**Design Issues:**

We had issues with how to implement 500 different client applications that communicate with a middleware agent. The first design/trial was to use two different processes (a client process with 500 threads) and (a middleware process). However, dealing with IPC in rust seemed to be more trouble than it was worth. We found that Rust has built-in objects that allow communication between threads of the same process, so we made the client a single process instead.

However, we faced another issue when we wanted the middleware to reply to the client threads. Using rust channels doesn’t allow for a single producer and multiple consumers, so we had multiple options. In the end, we decided to create 500 channels (senders and receivers), each thread will have a receiver and the middleware thread will have 500 senders, it will use the suitable sender when replying to a specific thread.

**Technical/Code Issues:**

* Global variables,
* Error handling, and underflow and overflow problems
* Server which keeps replying when it's down
* Marshaling
* Getting current time.

# Metrics & Performance

Both the clients and the servers log some performance matrices

* Server logs load every second, and when it goes down
* Client logs number of sent, fulfilled, and failed requests. It also logs the latency and the duration

The table below lists some test cases for our system, listing the logged data for each case. We compared the performance of the server using the basic round-robin load balancing and the threshold load balancing. We also test cases where there is only one client middleware working and cases where there are two working.

| Test cases | latency | Throughput | Requests failed | Avg server 0 load | Avg server 1 load | Avg server 2 load |
| --- | --- | --- | --- | --- | --- | --- |
| 2 clients Round-robin | 61 | 8108 | 1 | 4997 | 6630 | 7225 |
| 1 clients round robin | 44 | 11140 | 1 | 5681 | 6100 | 3989 |
| 2 clients threshold | 66 | 7459 | 13 | 19791 | 12932 | 6466 |
| 1 clients threshold | 44 | 11082 | 1 | 4645 | 4200 | 4400 |

## 

## Discussion

*1 vs 2 clients case:*

* 1 client case has less latency and higher throughput. This is expected as this client has the full attention of the server instead of competing with another client. This shows us that the servers are put under load when serving 2 clients at the same time because the latency and throughput per client get worse

*Threshold vs Round Robin:*

* This system is very simple, with just 3 homogenous servers. This means that an advanced load-balancing algorithm such as threshold would not improve the performance of the system.
* The cases with servers running threshold have slightly higher latency and lower throughput than the round-robin cases. This is most likely because of the overhead that comes with threshold load balancing. If the system is complex enough, the benefits of this load-balancing algorithm would outweigh this overhead. However, for our simple system, the overhead results in a decrease in performance.
* The threshold algorithm has the effect of increasing the load on the servers for the same throughput. This can be seen in the table above in the “2 client threshold” case. This is partially responsible for the decrease in performance.
* Thus, we have decided to include both versions of the server in our final submission, one with round robin and the other with threshold. This is because the round robin gave better performance. On the other hand, threshold is the more advanced algorithm that would yield better performance in more advanced systems.

# Member Contributions

## Ahmad Shaaban:

* Implemented thread pool on server
* Implemented ring distributed election
* Implemented election announcement handling
* Implemented threshold load balancing
* Implemented server logging

## Abdullah Nashat:

* Initial research on rust basics
* Created simple server
* Implemented round robin
* Implemented Request-reply protocol on server
* Implemented marshaling

## Abdelrahman Reda:

* Created simple client
* Implemented Request-reply protocol on client
* Implemented client timeout
* Implemented multithreaded client
* Implemented client logging