DISTRIBUTED SYSTEM PROJECT - REPORT

Supervisor: Dr. Amr El-Kadi

Basant Abdelaal (900192802)

Joseph Boulis (900182870)

Nehal Fooda (900183495)

TABLE OF CONTENT

INTRODUCTION

In this project, the main aim is to design a distributed system in which components and nodes are spread across multiple computers and coordinate their tasks and efforts to achieve the final goal of handling clients' requests with minimum loss.

The project focuses on two main components:

- Load balancing/ Sharing: Load balancing in distributed computer systems is the process of redistributing the workload among processors in the system to improve system performance.
- Distributed election: An election algorithm is used to find the coordinator to play a particular role in distributed environments, for example, managing the utilization of resources. Various algorithms have been proposed to elect a leader among peer processes. It is required to select a new leader if the existing one fails to respond.

The physical requirements include

- Three equally weighted servers responding to clients' requests and handling the failure of one of them.
- A large number of clients (min 100 clients) send requests to the servers and receive responses in case of the success of the request.

SUMMARY OF LITERATURE SURVEY

Regarding load balancing, the existing algorithms can be classified in two ways: static load balancing and dynamic load balancing.

For Static load balancing, prior knowledge of some resources involved in the system is a must, leading to determining the performance cost before the start of execution. On the other hand, a limitation of this technique is that processes can not move to other resources while running. To illustrate, Round Robin is a static load-balancing algorithm that iterates over available servers and assigns them the upcoming clients' requests with harmonious time slots. Other static load balancing algorithms, such as weighted round robin, Threshold Based Load Balancing, and Hash load balancing, follow the same idea of statically assigning the requests to the available servers with a slight adjustment in each algorithm.

Dynamic load balancing takes into consideration the server weights and state to ensure fair distribution of the traffic. It is not provided with prior knowledge of the servers that lead to the runtime load balancer system. That is why real-time communication between the servers is required. For example, The Least Connections load balance algorithm considers the servers handling the fewest number of tasks when assigning a new coming client request. Other algorithms that implement the dynamic load balancing technique are Throttled load balancing, Central Queue Algorithm, and Equally Spread Current Execution.

Concerning election algorithms and distributed election, the election of a new leader among peer processes requires that the elected process not be crashed and live. There are legacy algorithms that are Ring based and bully algorithms, and there are variants, such as the Modified bully algorithm.

CHOSEN ALGORITHMS

As discussed in the literature review, there are multiple existing algorithms for both load balancing and election distribution; thus, in this section, we will deliberate about our chosen algorithms, the reason for the choice, and their detailed descriptions.

In terms of the selected load balancing algorithm, we used **Round Robin**, which is a static load balancing algorithm that accommodates the project requirements and constraints; the system contains three equally weighted servers without adding the overhead of constant real-time communication between the servers that are required in the dynamic load balancing algorithm.

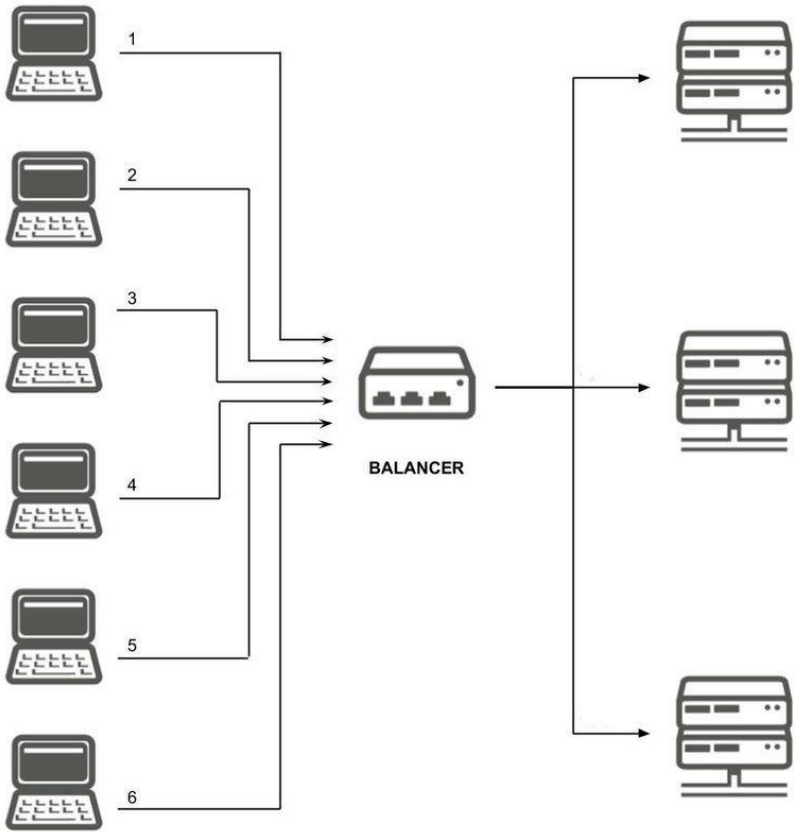
Algorithm Description:

Regardless of the workload on a given VM, the round-robin method distributes tasks to the next VM in the queue. Each process is given a predetermined time slot without any priority. Client requests are routed to available servers on a cyclical basis.

Features:

- It is simple because of the static nature of the algorithm that does not require dynamic changes during runtime.
- starvation-free because all processes get a fair share of CPU.
- Fair as each server is assigned client requests equally and gets an equal share of CPU.
- employs time-sharing, giving each job a time slot or quantum.
- It does not consider the resource capabilities, priority, and length of the tasks, which goes along with the project requirement as discussed earlier.

Architecture:



Src: [Analysis of the use of SDN for load balancing]

Moreover, **Asynchronous Modified Bully** is the chosen distributed election algorithm because it does not need a synchronous system, which is what our system needs. In addition, it has a low turn-around time - the number of message transmission times between the initiation and termination of a single run - and minor space complexity compared to other election algorithms. Therefore it suits our simple system with the targeted performance.

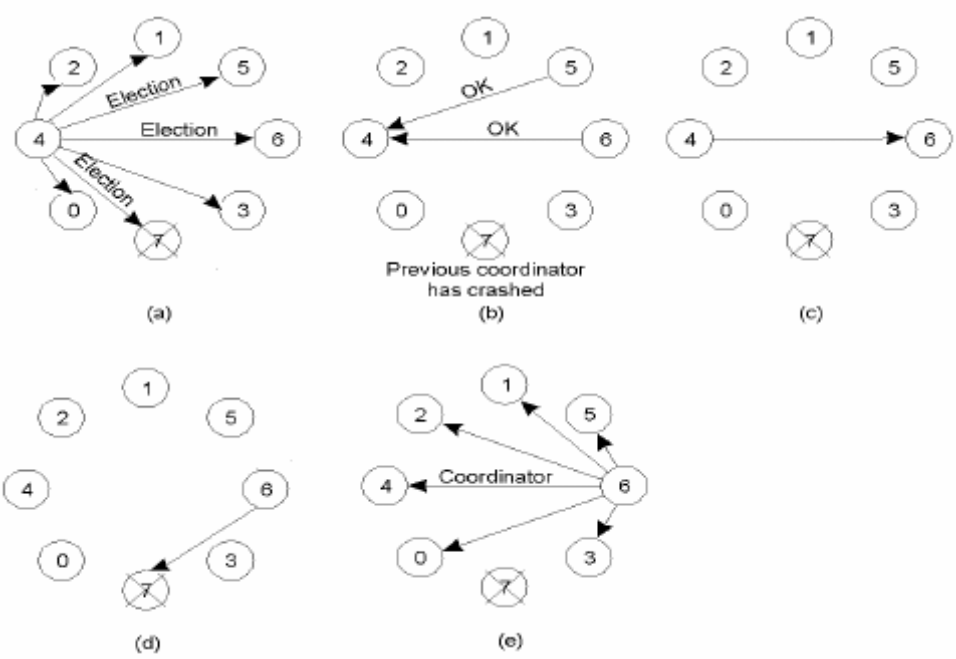
Algorithm Description:

There are three types of messages in the algorithm, an election message to announce an election, an ok message as the response to an election message, and coordinator messages sent to announce the new coordinator, among all other live processes. When any process finds that the coordinator process has crashed, it starts the run by sending an election message to processes with higher identifiers. Then, when a process receives an election message, it elects the higher processes. If a process receives no ok messages after a timeout N, it has the highest identifier. Therefore, it announces itself as the elected coordinator by sending coordinator messages to all processes with lower identifiers.

Features:

- It has low time-bound complexity O(1)
- Low space complexity of O(N), only the identifiers as their prior knowledge of other nodes is not needed.

Architecture:



Src: [Improved Bully Election Algorithm for Distributed Systems]

DESIGN CRITERIA AND DECISIONS

When designing the system, we defined some criteria that we want to achieve and assumptions for our system:

1) **Load Balancing:**

The first assumption we made is that the servers in the system are equally weighted. Therefore, as described in the previous section we operated the load balancing based on a simple Round Robin algorithm. Therefore, we expect that servers will only process requests assigned to them explicitly.

2) **Fault Tolerance:**

To have a fault-tolerant system we need to achieve two points:

1. No Single Point of Failure

Since the leader is responsible for distributing the load, we need to make sure that leaders are insured to be always present. This is achieved by the distributed leader election. Therefore, formally, we expect that for a certain maximum time period, a leader should be elected and fully operating.

2. Server Failure is Discovered Automatically

In real systems, servers may fail suddenly. Therefore, we do not expect to have any clues sent from the failed nodes that it failed. Namely, there is no “Failure Message” that the leader receives. This is because our aim is to design a system that is as close to what happens in real life. Therefore, there should be a **dynamic technique for discovering the live nodes**.

Unlike the leader, that always needs to be present, follower servers may be dropped. And in that case, the only effect we expect to have is that the leader becomes aware of the live nodes and excludes the failed nodes from the system. Namely, to operate the load balancing only among the live nodes at any point in time.

3) **Location Transparency**

In order to have a dynamic system, we need to support the automatic discovery of new nodes registered to the systems and failed nodes. Therefore, we wanted to achieve a somewhat tough requirement which is not using IP addresses directly. Instead, the only

assumption we had is that the whole system is present under the same network and subscribes to a certain topic to be able to communicate with other nodes. Our motive behind this is:

- Clients will not need to know the IP addresses of the servers. Therefore, the topic will act as our naming service and the only thing that clients need to know to be able to send requests.
- New servers are easily added to the system. They only need to declare themselves as servers in the beginning. And also, failed servers are discovered dynamically as previously discussed.

4) **Minimal Logic on the Client Side**

To simulate what happens in real life, clients need to only have application logic. Therefore, they should not be aware of what servers they connect to, they only need to know the name of the Topic/Channel they subscribe to and broadcast their requests.

ARCHITECTURE

Given the discussed criteria, our system architecture is as follows:

- There are two roles for nodes/servers in the system:
    - **Leader:**

Leaders distribute the load of client requests among follower servers.

Leaders only forward the requests, so there is no overhead of processing the request - no application logic here.

Leaders should be aware of all the server followers and have an updated list of live nodes. Therefore, the list should be dynamically updated by excluding failed nodes or adding new nodes.

- **Follower:**

Followers respond to requests of the clients assigned to them.

The architecture can be represented in the following diagram:

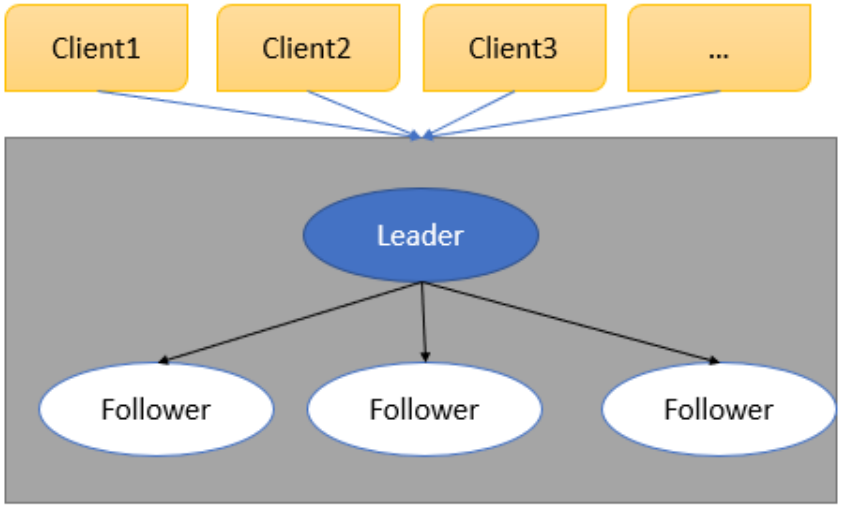


Fig. 1 System Architecture

Messages for Distributed Election:

There are two messages for election in the system:

- *Initiate Election:*
  When a server times out it sends an election round initiation
- *Leader Announcement:*
  The same server who initiated the election inspects the servers in the system and selects the one with the highest id. It then broadcasts the leader announcement message to all servers.

Architecture and Messages for Client Requests Load Balancing:

There are two messages for handling and load balancing client requests in the system:

- *Client Request:*
  The clients sends client request. Those are only handled by the leader.
- *Client Response:*
  The leader sends a client response message that contains all the data in the client request. In addition, it contains the field *handled* set to false and the *server_id* that handles this response.
  The assigned server then receives this unhandled response and processes it by marking the field *handled* with true and sending the response back to the client.

IMPLEMENTATION DETAILS

Communication

For communication, we chose to base our architecture on peer-to-peer networking. This is due to its simplicity, decentralized nature, ability to broadcast, and, finally, to be able to discover live nodes dynamically through inspecting listeners on the channel.

Regarding the implementation, we relied on some helpful libraries in Rust that perform basic peer-to-peer networking using TCP. The essential libraries were **libp2p** which has functionality for **mdns**, **floodsub**, **networkbehavior**, and **swarm**. They are also responsible for marshaling and unmarshalling data for RPCs.

The initialization of the communication all happened at the beginning of the main function.

```
behaviour.floodsub.subscribe(TOPIC.clone());

let mut swarm = SwarmBuilder::new(transp, behaviour, PEER_ID.clone())
    .executor(Box::new(|fut| {
        tokio::spawn(fut);
    }))
    .build();

Swarm::listen_on(
    &mut swarm,
    "/ip4/0.0.0.0/tcp/0"
        .parse()
        .expect("can get a local socket"),
)
.expect("swarm can be started");
```

Distributed Election

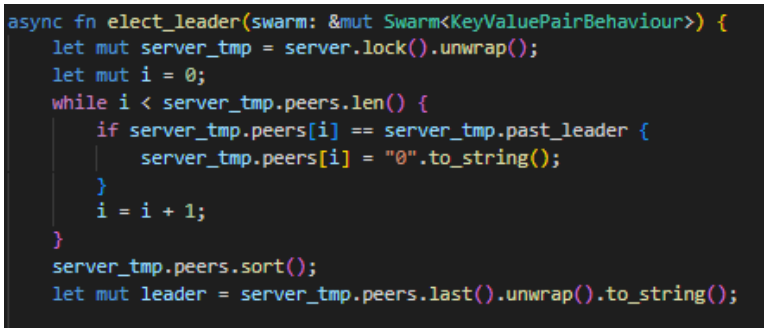
For the distributed election, as discussed, we chose the asynchronous bully algorithm. We based our algorithm on constant heartbeats that are sent from the leader and associated timeout from the followers' side.

Whenever the timeout passes in a server with no received heartbeats, the server starts an election round and elects the server with the highest id. It then populates the result to all the servers in the system, and they change their leader accordingly. That said, they compare the winning leader with their id and recognize themselves as a leader if the id matches.

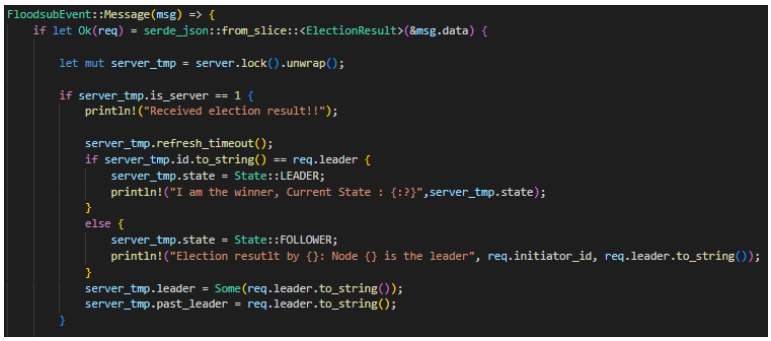
This mainly happens in the function broadcast_heartbeat from the leader side. And from the follower side, it is in **handle_timeout**, **elect_leader**, and handling the **election result** in the NetworkBehaviour as illustrated below.

```
async fn broadcast_heartbeat(swarm: &mut Swarm<KeyValuePairBehaviour>, resp: ClientResponse) {
    if resp.sender_id != "None" {
        println!("Sending data");
        let json = serde_json::to_string(&resp).expect("can jsonify request");
        swarm
            .behaviour_mut()
            .floodsub
            .publish(TOPIC.clone(), json.as_bytes());
    }
    else {
        println!("Sending heartbeat!");
        let id = server.lock().unwrap().id.to_string();
        let req = Heartbeat {leader_id: id,};

        let json2 = serde_json::to_string(&req).expect("can jsonify request");
        swarm
            .behaviour_mut()
            .floodsub
            .publish(TOPIC.clone(), json2.as_bytes());
    }
```



```
async fn elect_leader(swarm: &mut Swarm<KeyValuePairBehaviour>) {
    let mut server_tmp = server.lock().unwrap();
    let mut i = 0;
    while i < server_tmp.peers.len() {
        if server_tmp.peers[i] == server_tmp.past_leader {
            server_tmp.peers[i] = "0".to_string();
        }
        i = i + 1;
    }
    server_tmp.peers.sort();
    let mut leader = server_tmp.peers.last().unwrap().to_string();
```



```
FloodsubEvent::Message(msg) => {
    if let Ok(req) = serde_json::from_slice::<ElectionResult>(&msg.data) {

        let mut server_tmp = server.lock().unwrap();

        if server_tmp.is_server == 1 {
            println!("Received election result!!");

            server_tmp.refresh_timeout();
            if server_tmp.id.to_string() == req.leader {
                server_tmp.state = State::LEADER;
                println!("I am the winner, Current State : {:?}",server_tmp.state);
            }
            else {
                server_tmp.state = State::FOLLOWER;
                println!("Election resutlt by {}: Node {} is the leader", req.initiator_id, req.leader.to_string());
            }
            server_tmp.leader = Some(req.leader.to_string());
            server_tmp.past_leader = req.leader.to_string();
        }
```

Load Balancing

For the load balancing, as discussed, we implemented an equally weighting algorithm through a simple Round Robin that the leader performs. The leader here acts as an agent who is aware of all the servers in the system and distributes a load of incoming requests accordingly. This is done through **discovered_nodes** and handling client requests in the **network behavior**. The following are screenshots of the relevant parts of the code.

```rust
impl NetworkBehaviourEventProcess<MdnsEvent> for KeyValuePairBehaviour {
    fn inject_event(&mut self, event: MdnsEvent) {
        match event {
            MdnsEvent::Discovered(discovered_list) => {
                for (peer, _addr) in discovered_list {
                    self.floodsub.add_node_to_partial_view(peer);
                }
            }
            MdnsEvent::Expired(expired_list) => {
                for (peer, _addr) in expired_list {
                    if !self.mdns.has_node(&peer) {
                        self.floodsub.remove_node_from_partial_view(&peer);
                    }
                }
            }
        }
    }
}
```

```rust
} else if let Ok(req) = serde_json::from_slice::<ClientRequest>(&msg.data) {
    let mut server_tmp = server.lock().unwrap();
    if server_tmp.state == State::LEADER {
        let handler_id = server_tmp.requests_cnt % (server_tmp.peers.len() as i32);
        let handler = server_tmp.peers[(handler_id as usize)].clone();
        server_tmp.requests_cnt = server_tmp.requests_cnt.clone() + 1;
        println!(" Received request id {} from {}",req.request_id,req.sender_id.to_string());
        forward_client_request(self.response_sender.clone(), req, handler.to_string());

    }
```

Client

The client logic was simple requests that each has the client's and the request's id.

The assigned server from the load balancing responds to the request by sending a client response that includes the same fields plus the field done. The main work of the client happens in the **broadcast_client** method

```rust
async fn broadcast_client(swarm: &mut Swarm<KeyValuePairBehaviour>, request_id: i128) {
    println!("Client Request is working!!!");
    let req = ClientRequest {request_id : request_id, sender_id: PEER_ID.clone().to_string(),};

    let json = serde_json::to_string(&req).expect("can jsonify request");
    swarm
        .behaviour_mut()
        .floodsub
        .publish(TOPIC.clone(), json.as_bytes());
}
```

EXPERIMENTS AND STAGES OF IMPLEMENTATION

Experiment 1: Communicating different servers

In this stage, we implemented peer-to-peer networking between servers.

All servers communicate with each other using **broadcasting on TCP 0.0.0.0**

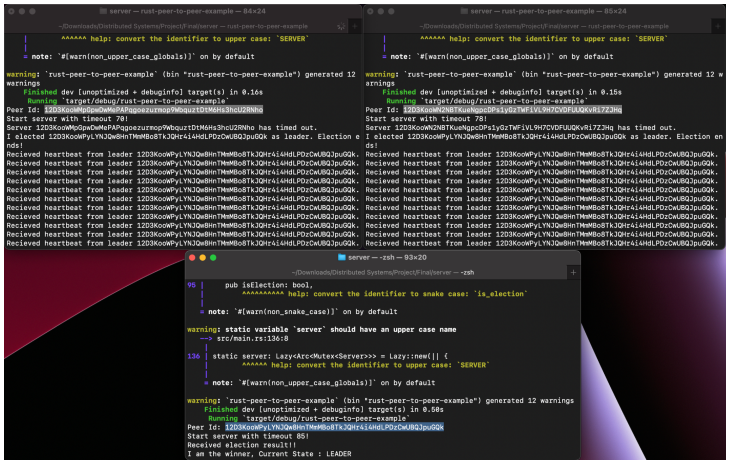
We verified that servers on the same PC (different terminals) and different PCs were discovered and communicated the same way. This allowed for **location transparency**.

Experiment 2: Distributed Election

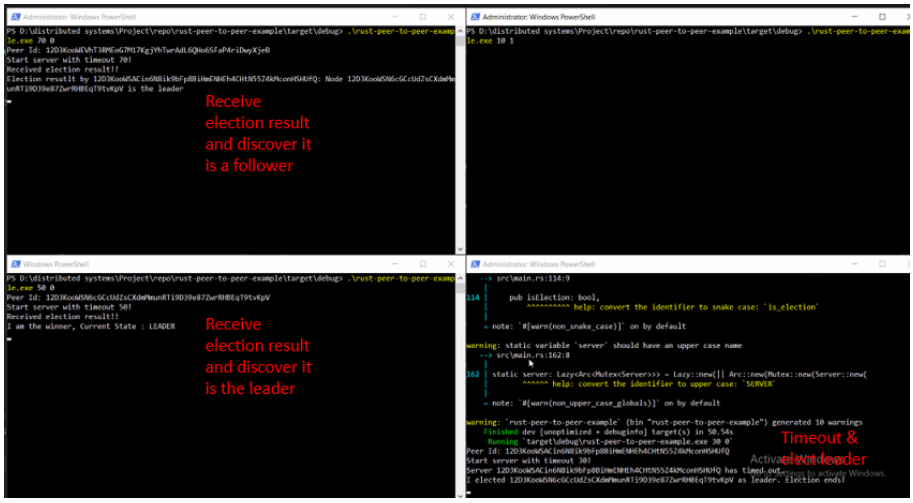
In this experiment, we developed the aforementioned leader election algorithm, “Asynchronous Bully.” We verified that it is working on two steps:

1. **Initial Election:** Start communication between the three servers and check whether they will be able to elect a leader among them or not.

   And one of the servers elected the leader with the highest id successfully, as shown below.

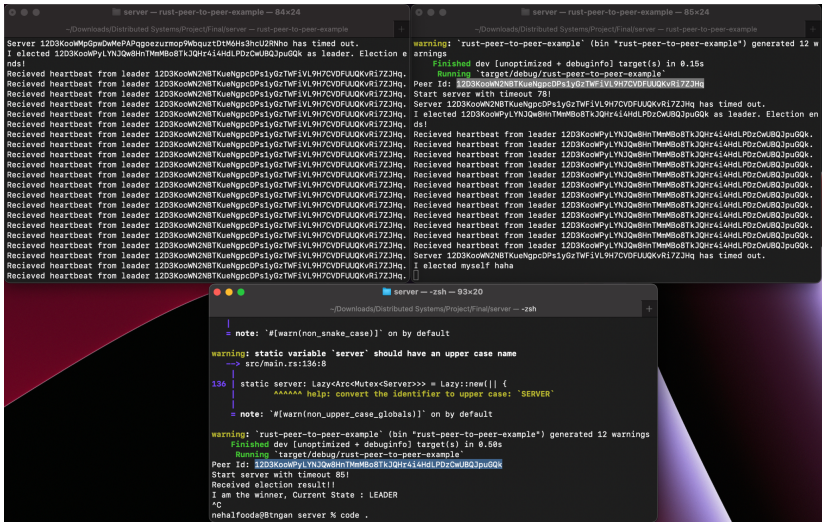


Another screenshot shows a server starting the election (downright) and the result of the election was populated to the other two servers (up and down left):



2. **Drop the leader and wait for timeout and election**: We dropped the leader server by killing it (Ctrl + C) and checking whether the remaining active servers will start a new election and elect a new leader.

   As shown below, it was successful, and a new leader was elected.

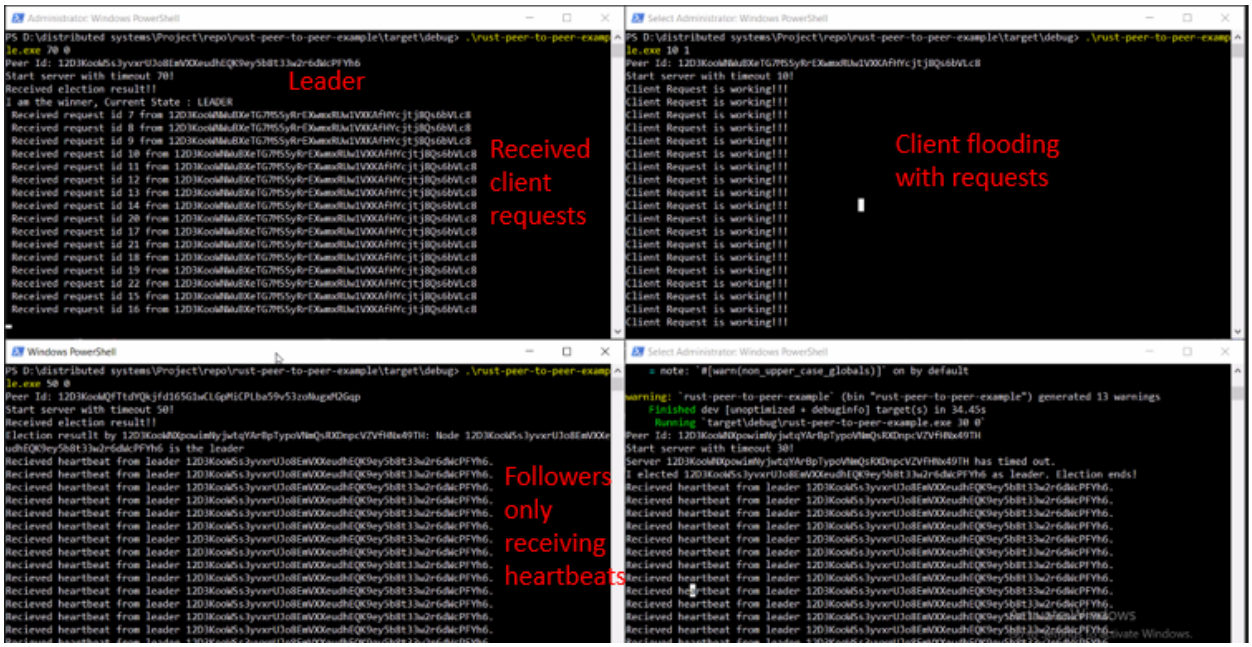


Experiment 3: Create Client Logic and Process Requests by Leader

In this experiment, we tested the performance of the following:

- Create a client that floods the server with requests through broadcasting.
- Only the leader server can handle the requests and forward them according to the load-balancing algorithms (next step).

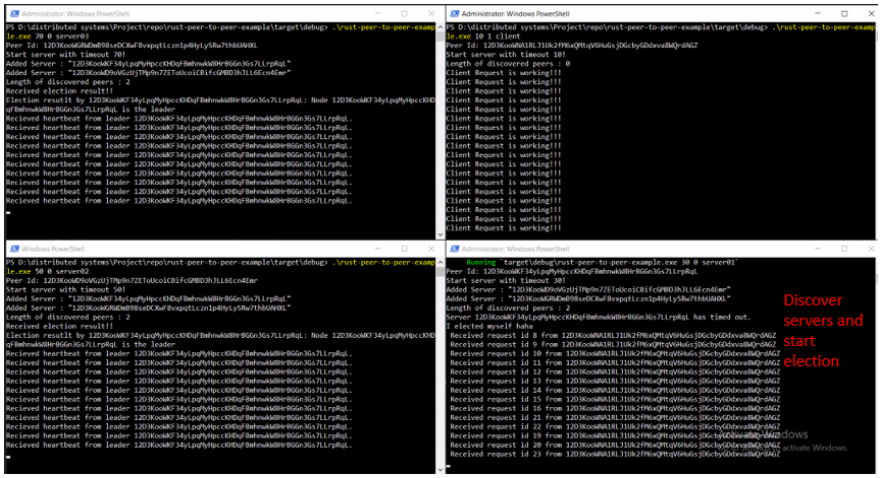
Results are shown and annotated:



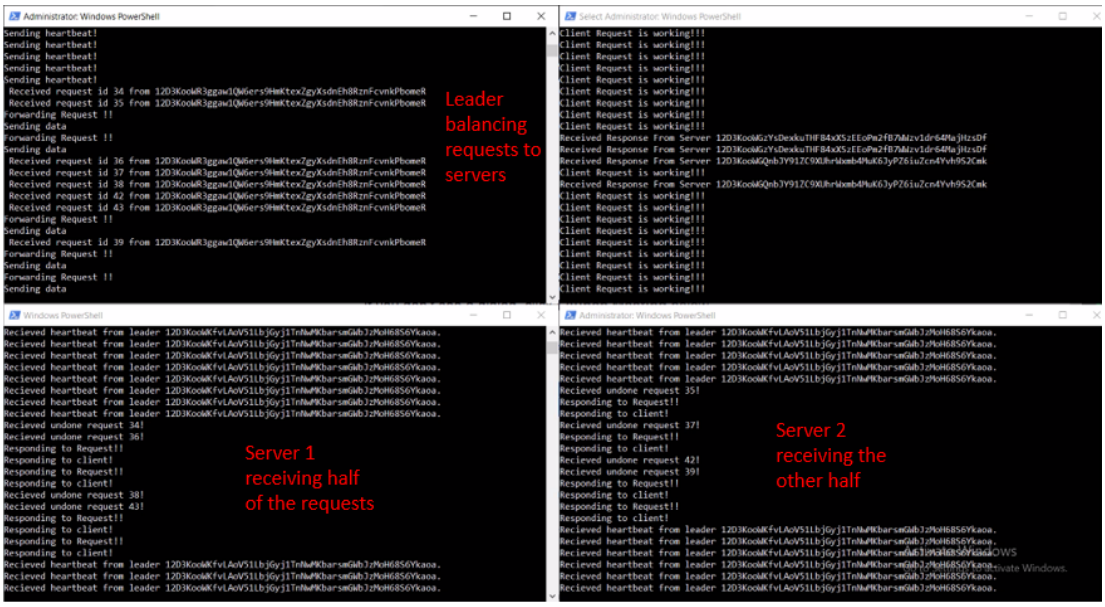
Experiment 4: Create Load Balancing:

In this stage, we added load-balancing logic dynamically to the elected leader. According to the requirements and to simplify the logic, the load balancing algorithm is just a simple Round Robin that equilibrates the requests among the live servers.

- Step 1: Make the leader aware of the existing live servers



- Step 2: Balance the incoming requests among the discovered servers.

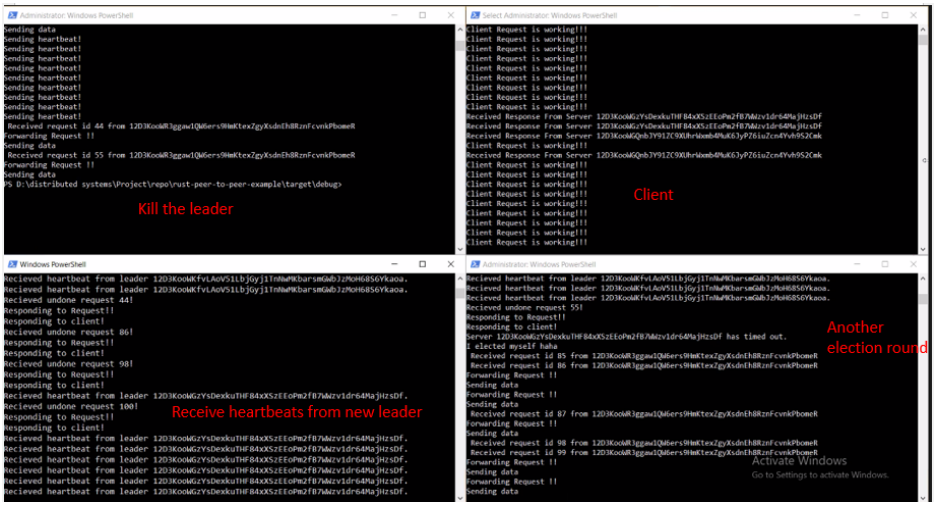


Experiment 5: Test Load Balancing and Distributed Election together:

Last but not least, we ensure that everything runs smoothly and that the load balancing and distributed election are working together. We did that by just killing the leader and observing another leader is elected, and the load balancing is still operating.

Finally, we dropped one of the follower servers and made sure that the leader

updates its active list of followers and redistribute the load among the new list.



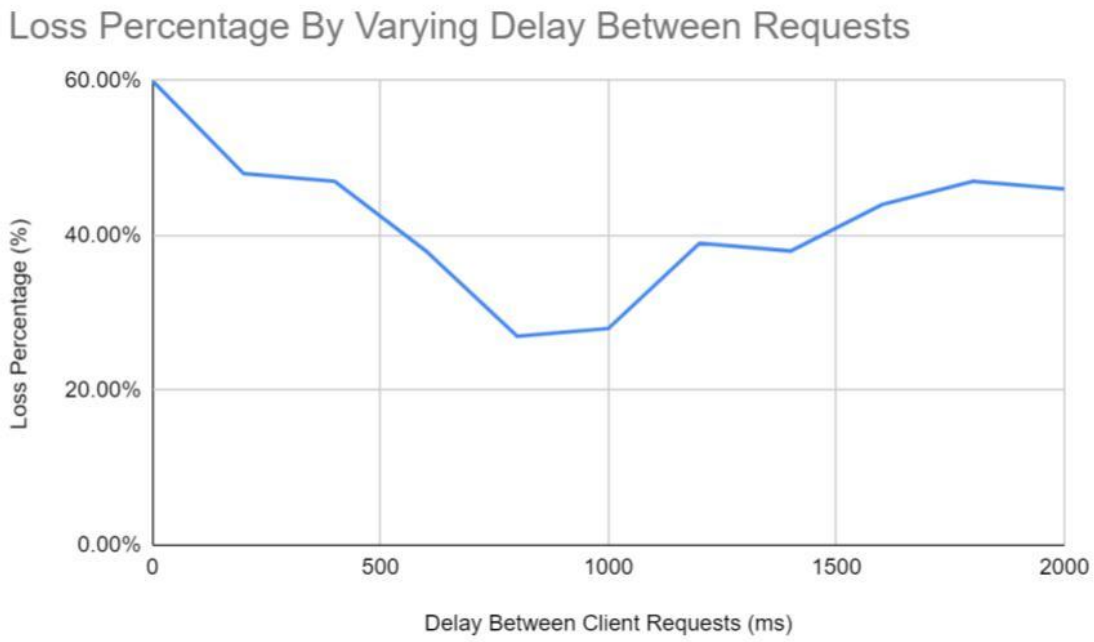
Experiment 6: Stress load balancing and gathering statistics

The last experiment was to create clients that all flood requests to servers (1 leader and three followers) and collect some statistics, which are discussed in the next section.

STATICAL RESULTS AND DISCUSSION

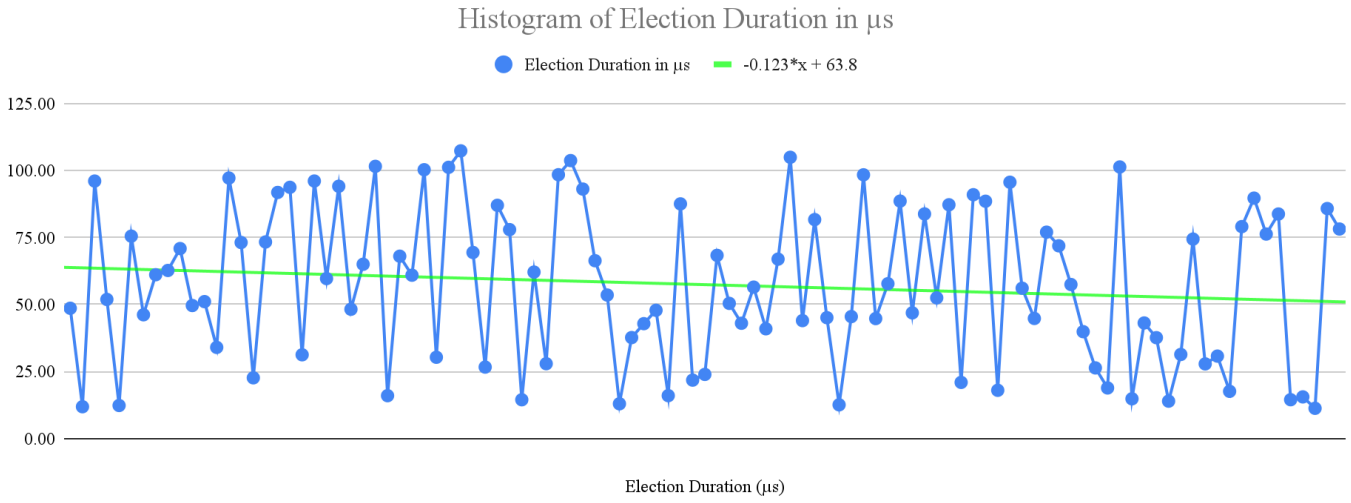
Testing Packet Loss Percentage by Varying Client Requests Dealy

In this graph; we experimented with the effect of varying the delay between two consecutive requests made by a single client to the servers. The delay ranged from 0 to 2 secs. As seen in the graph, the more we increased or decreased the delay from 800 ms, the more the packet loss rate we have. We infer that the less delay we have between two consecutive requests incurs a higher sending rate, leading to network congestion. Hence, more packets compete to be sent over the shared medium, resulting in multiple request losses.



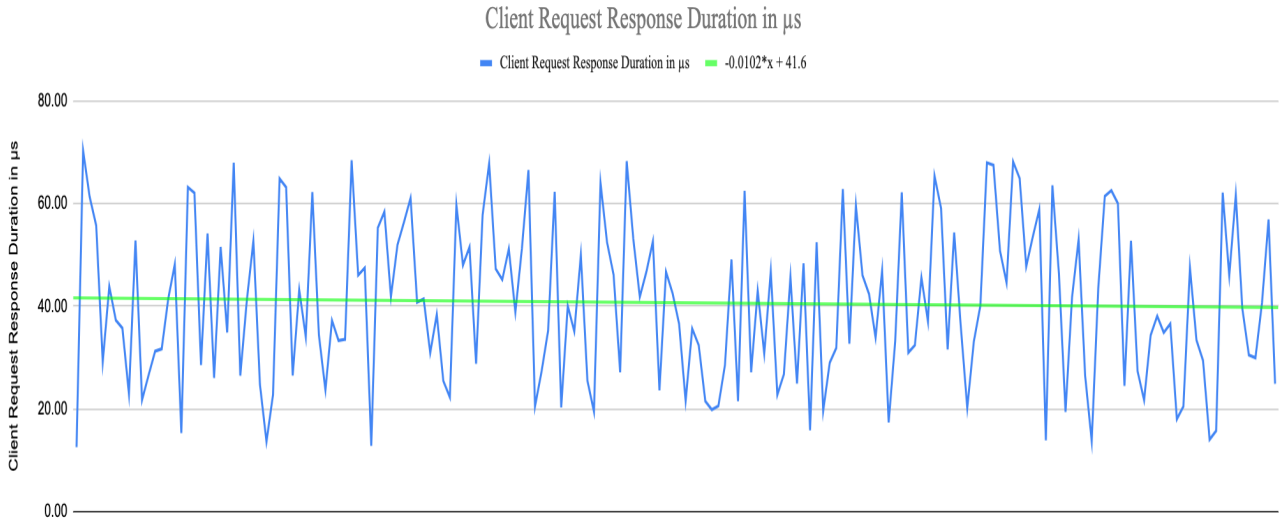
Election Duration

In this histogram, the election duration of 105 leader elections was recorded and displayed. The distribution of the graph data shows that the max election duration was around 107 µs and the min election duration was around 12 µs. Therefore, it is displayed in the graph that the average election duration is about 60.8 µs. The standard deviation for the represented data is 28.6 µs which indicates that the election duration values are clustered around the average. The system can be considered stable because there are no considerable fluctuations in its data and measurements.



Client Request Response Duration

In this histogram, the time the server takes to respond to a client request was recorded for 184 successful clients' requests and displayed. The distribution of the graph data shows that the max response duration was around 70 µs and the min response duration was around 13 µs. Furthermore, the average request response duration is about 41.6 µs. The standard deviation for the represented data is 18 µs which indicates that the time duration the servers take to respond to clients' requests are clustered around the average. Therefore, in addition to the election duration data represented above, the system shows more stability.



Success Rate:

The final average success rate for the tested clients was **87.43%.**

As shown in the graph above, we have varied the number of requests per client and spawned a different number of clients in each experiment, then calculated the average number of requests successfully received which is our success rate.

The result is very good given the complexity of the system and using broadcasting and Topic subscription instead of connecting to IPs.