THE AMERICAN UNIVERSITY IN CAIRO

School of Science and Engineering

Distributed Systems Project Report

Fall 2022

Mohamed Nasr 900182735

Youssef Kandil 900182405

Mazen Hassan 900191797

Introduction

In the process of the literature review, examining various algorithms related to load balancing and distributed election in distributed systems, there were many possible choices of implementations to choose from. However, simplicity was the main metric that was chosen as the filtering mechanism for said algorithms. Using this metric, the round robin algorithm was picked for the load balancing technique to be implemented, and the modified sandipan was chosen for the distributed election algorithm. The small scale of the distributed system being used allows for relatively simpler algorithms to be used, without needing to account for data being lost.

Firstly, the report will discuss the topology of the distributed system that is being used. Secondly, the report will discuss the implementation of the load balancing algorithm, as well as the distributed election algorithm. Thirdly, the report will discuss how the algorithms support fault tolerance in the distributed environment. Fourthly, the report will explain the engineering design choices and evaluate the effectiveness of the system, from the point of view of the client as well as the server. Lastly, the final section discusses the contributions of the group members in the project.

System Topology

The environment is set up as the following: there are two clients (that multithread into 500 clients each) sending out requests to the three servers that are set up. There is a middleware consisting of a set of agents where each agent is responsible for 500 clients (that is for our use case we have 2 agents in the middleware). An agent node is responsible for forwarding the relevant data to the relevant server, managing the round-robin algorithm, and collecting some reporting data. The three servers are to handle the incoming requests through the load balancing mechanisms (implemented in the middleware), as well as using distributed election to handle failure, and exhibit some fault tolerance throughout all this occurring.

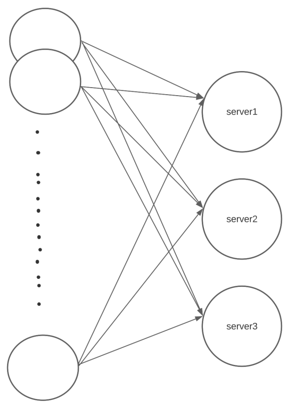
Algorithms

Load Balancing

For our load balancing technique, round robin was picked for its relative simplicity in implementation. To implement round robin, we took two approaches:

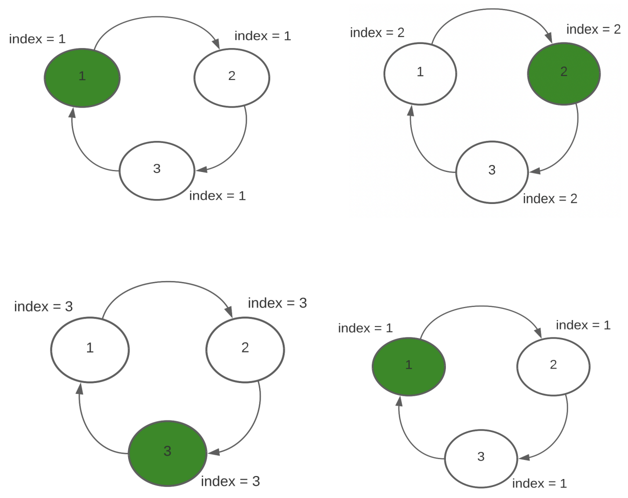
- Decentralized implementation on the servers' side
- Implementation in the middleware through the agents

For the first approach, each server was given a machine ID, ranging from 1 till 3. In addition, a round robin index globally updates in our middleware agent. This index indicates which machine needs to process the data. The middleware agent broadcasts the request to all the machines, then the machine with the relevant ID that matches the current value of the global round robin index is set to process the request.



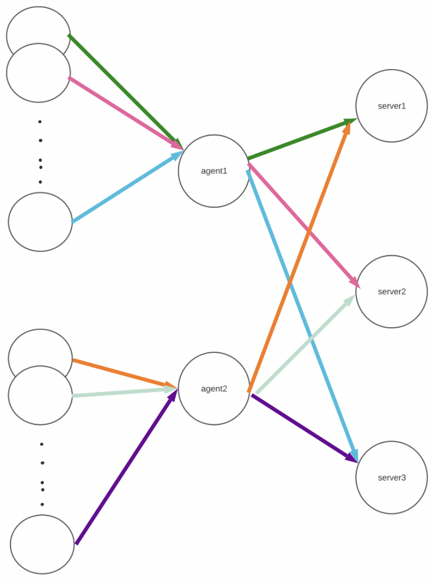
Multicasting from clients to servers

That is, if the ID of the machine is equal to the current round robin index, that specific machine proceeds to process the request. After the request is processed, the global round robin index moves on to the next machine, and once it reaches 3, it starts back at 1 and repeats until all requests are processed. This splits the load in three equally among all server PC's achieving load balancing in this system.



First approach for load balancing

For the second approach, the client agents which reside in the middleware will be the ones responsible for handling the load balancing algorithm. The clients send there request to their designated middleware agents, then the middleware agent keeps track of the server that has the turn to execute the request.

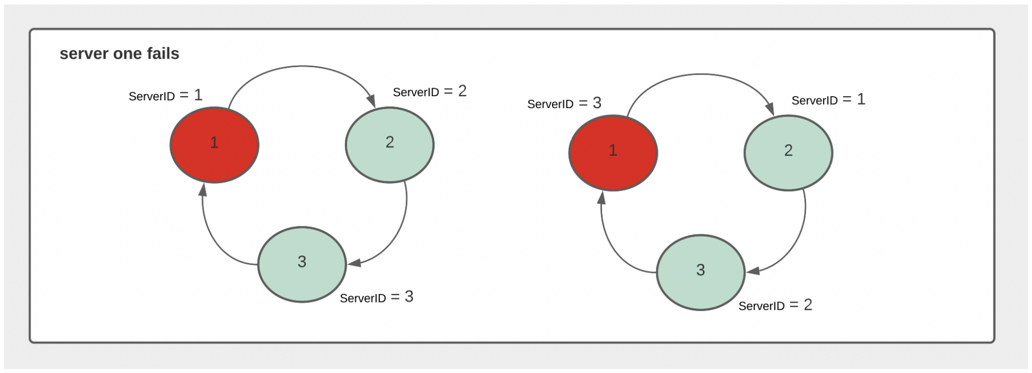


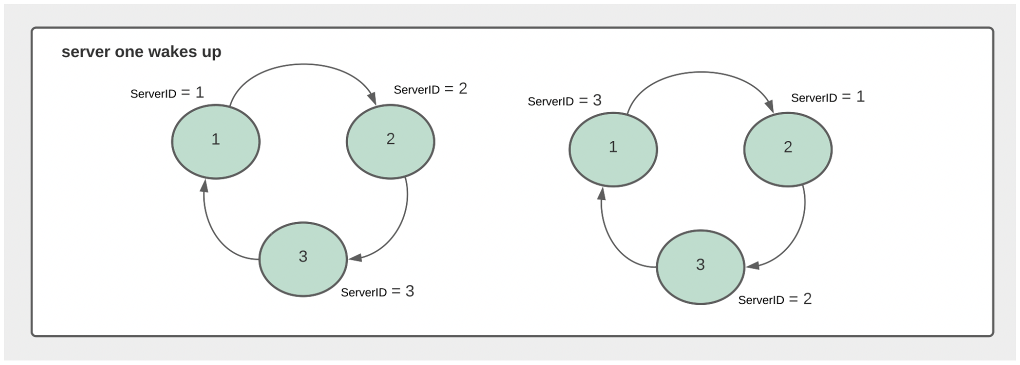
Second approach for load balancing

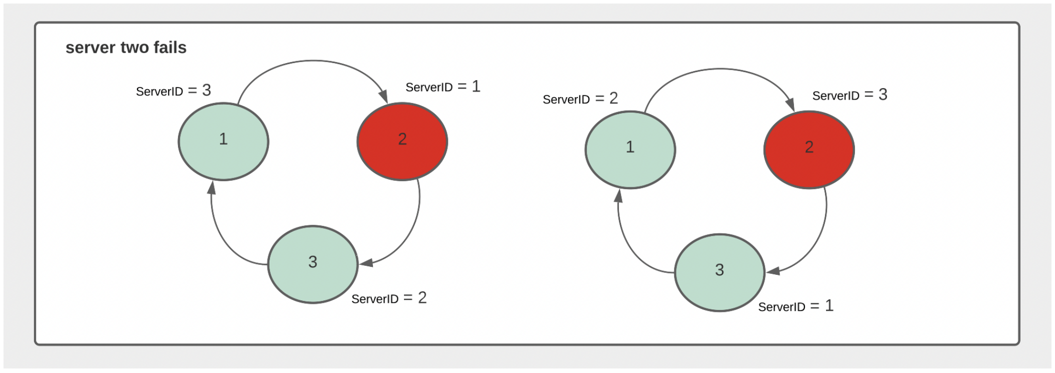
Distributed Election

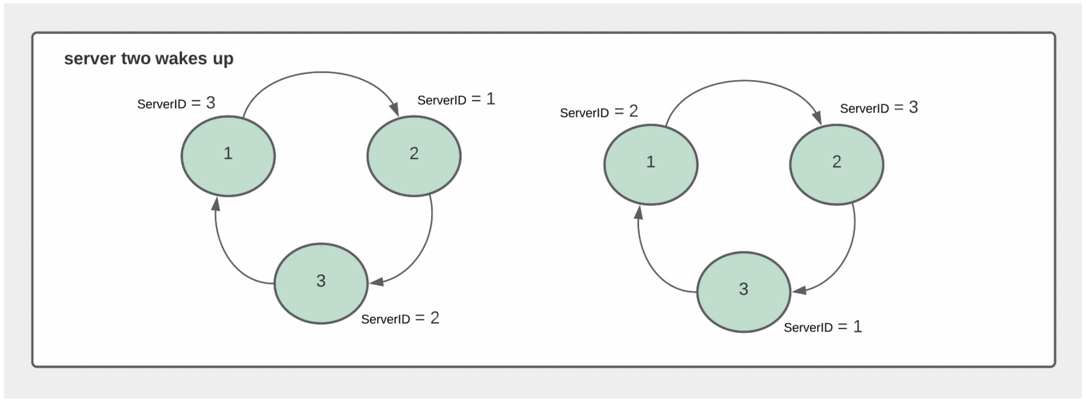
For distributed election, its main purpose in the project was to help with deciding which machine would have to fail. Since all the machines that were being used in this environment all have the same specifications and performance, the ID of each machine was used as a virtual metric to indicate the importance of each server. ServerID 1 is the weakest among them, while ServerID 3 is the strongest, with ServerID 2 being the intermediate value in terms of importance. Failure is always induced in the server that has ServerID 1, which means that the weakest server is set to fail. The sandipan' s algorithm keeps a table at each server node that keeps track of the election trait (which in our case is the ServerID). The ServerIDs are dynamically changing in our system to mimic a real case scenario. The tables are dynamically updated by all servers whenever the ServerIDs are changed, or a server fails.

Note that the sandipan's algorithm changed the main focus of the election algorithm from deciding which node has to fail at the time of the election, to updating the tables periodically and consistently such that whenever an election is called there would be no overhead of deciding which node has to fail, since all nodes would have a complete update of the table.









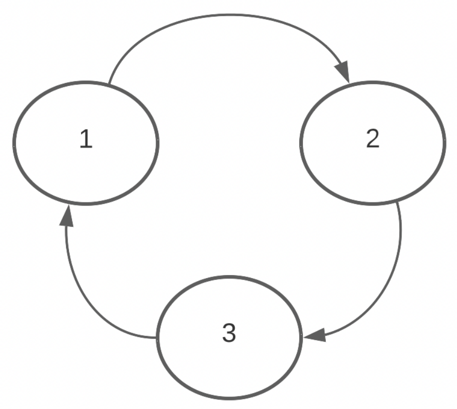
Sandipan’s modified algorithm + fault tolerance

Communication Mechanism

In order to facilitate the distributed election, load balancing, as well as metric gathering in this system, a communication system had to be set up in order to properly have nodes share relevant information with other nodes. For example, clients send requests to the middleware agent servers. The agents then proceeds to check the round robin index, and proceeds to forward the process that needs to be executed to the relevant node that has an ID that is equal to the current round robin index. Once a node failure is detected in one of the nodes, that needs to be communicated back to the middleware agents, in order for the round robin index range to be updated for only two machines. Once a node is done processing, it replies back to the client with a message to indicate that the process is now complete.

Fault Tolerance

To accommodate for a node going down, as a part of inducing failure in our system for testing, each node needs to be notified that a node has gone down. Each node is responsible for knowing that its proceeding node has fallen. For example: If node 1 falls, node 2 knows that node has fallen as it does not receive any signal from it. If node 2 falls, node 3 learns that node 2 has fallen when it does not receive any signal from it, and if node 3 falls, node 1 will learn that it has fallen once it stops receiving a signal from node 3.



Notification connections

Once a node knows that one of the nodes has fallen, it notifies the middleware agent , which then proceeds to change the ID's of both currently working nodes to 1 and 2, change the round robin maximum value to two, and have the nodes continue to process the incoming processes that are coming to them. Once the fallen node comes back, it gets assigned the ID 3 (indicating that it's the strongest node and thus the last to be chosen to fall) and the system continues to operate.

While one of the servers falling would result in packets falling that were otherwise going to be processed by said server, the system adjusts from distributing the processes from 3 servers, to 2 servers, accommodating for the packet that would have gone to the third node. The other two server devices continue processing the process's that are being sent to them. Thus the system handles the failure of a node , as the distributed environment is not completely shut down by the failure of one node and it adjusts to that failure, limiting the impact of said failure, thus exhibiting fault tolerance.

Engineering Choices

1. Since we have similarly capable servers, we decided to create a virtual trait that decides which one has to fail. As discussed before this trait is the ServerID. We also decided to give them random assignments initially, then wherever one fails the other nodes discover that and they will be assigned ServerID1 and ServerID2. Which means that we assume that their capabilities will decrease since they are now executing higher loads. We also decided that whenever a server wakes up, it will be assigned ServerID3. Which means that since it has just woken up then it has the highest capability of handling tasks. This also guarantees dynamic assignments of ServerIDs to the nodes.
2. We chose to handle round robin through a middleware agent, to avoid issues that can be caused by the desynchronisation of the round robin counter in a decentralized environment when failure occurs. When attempting to implement round robin through a decentralized global counter handled by the server nodes, server failure would cause desynchronization of the global round robin index, causing an unequal distribution of the load to occur once servers start failing, thus we would not be achieving an adequate form of load balancing. Thus switching to the middleware agent and giving the agent the functionality of maintaining and managing the round robin index would mitigate this issue.

3. Another concern we managed to handle is the congestion of the port at any node. When creating a connection for an agent, a client, or (more importantly) a server, we decided to create multiple sockets for receiving, each dedicated to a specific source. For example a server has a designated socket for the periodic communication with the servers, and another designated socket to receive the agent's requests. This decreases the congestion at each socket, provides larger buffer space to handle the upcoming request, also decreases the probability of losing packets. Consequently increasing reliability.
4. We decided to make all the sending sockets non blocking, yet all the receiving sockets blocking with varying TTLs from 10 to 100 milliseconds. This is to guarantee that with very high probability the system is able to receive most of the requests, yet insure the high speed of the system.

Evaluation

- Reliability and Response time
    - Requests sent by agent one: 1,029,082
    - Requests sent by agent two: 1,029,632
    - **Total requests sent**: 2,058,717
    - Requests processed by server one = 683259
    - Requests processed by server two = 685320
    - Requests processed by server three = 684667
    - **Total requests processed**: 2,053246
    - **Lost requests** = 5471
    - Total running time = 272 minutes
    - **Average response time** = 0.126 milliseconds.

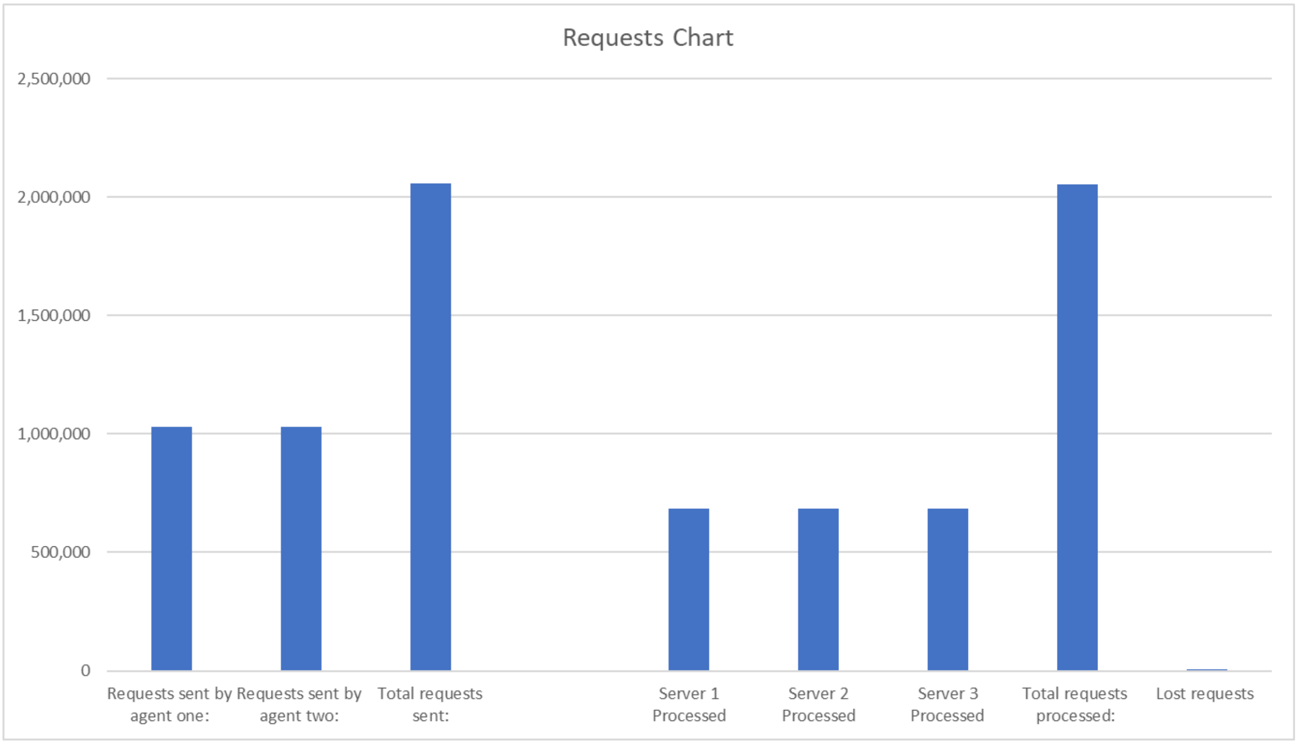


Chart indicating number of requests sent and processed by which node

- Workload Distribution
    - The variation among the three servers in the amount of process's computed stays relatively small to the total amount of processes computed, thus the system exhibits reliable load and workload distribution
- Scalability

  - In terms of scalability, the system was tested with a low number of clients as well as high numbers. The system operated the same in both conditions, highlighting that the system is in fact scalable
- Decentralization
  - If we consider the current size of the system, having two agents where each handles 500 clients, we cannot definitively state that the system is decentralized. However, when scaling the system large enough, having too many agents, each agent will only be managing a small portion of the client. Then we can say that the system is decentralized.
- Fault tolerance
  - As discussed in the fault tolerance section, when failure is induced in the system, the system adjusts to handle that failure and continues operating, thus fault tolerance is exhibited in our system.

- Transparency
  - We provide a middleware that consists of a set of agent that shields all the servers' details, load balancing algorithm, distributed elections algorithm, and the failure of servers from the clients.

Contributions

- **Mohamed Nasr**
    - **Second approach for load balancing**
    - **Middleware**
    - **Testing/Report**
- **Youssef Kandil 900182405**
    - **First approach for load balancing**
    - **Multithreaded clients, connection establishment**
    - **Testing/Report**
- **Mazen Hassan 900191797**
    - **Sandipan’s modified approach for distributed election**
    - **System integration**
    - **Testing/Report**