Fundamentals of Distributed Systems

(CSCE 4411)

Instructor Name: Dr. Amr El Kadi

Project Report

Fall 2022

Mohamed Ayman 900182267

Yassin Walid 900183310

Nada El-Shenawy 900182782

Introduction

The aim of this project is to simulate the interactions between servers and clients in a distributed system. We implemented this project using the language Rust. Our system consists of two client machines and three server machines. Our project implements two main concepts: load balancing and fault tolerance. Load balancing is concerned with the distribution of client requests over the available servers at any given moment, such that the loads on the servers would be near equal. The fault tolerance mechanism is concerned with the ability of the system to continue operating in the case of a server failure. In this report, we will discuss our design choices, implementation logic and the overall performance of the system.

Design & Implementation

Network Protocol

For the choice of network protocols, we used UDP (User Datagram Protocol). UDP is a connectionless protocol; which means that the request packet is sent directly, without the need to establish a connection first (handshake). On the other hand, the TCP (Transmission Control Protocol) is a connection-oriented protocol, where a connection must be established before sending request packets. The advantages of using UDP over TCP include its simplicity, higher speeds, lower latency, avoiding retransmission delays and the overhead of establishing the connection.

Clients

The clients are the end-users of the system. In our network, we have 2 client machines. Each machine has multiple threads that are continuously sending requests in an infinite loop. There are 2 agents; 1 agent for each client machine. The client threads send the requests to the corresponding agent for their machine.

Client Implementation

Each client machine creates 500 threads. Each thread loops infinitely, continuously sending requests to the corresponding agent for this client machine. The client waits for a

response for each request sent. In case a reply is not received, the thread times out after 3 seconds, and proceeds to send the next request, ignoring the failed one.

Client Agents

The agents act as a middleware between the clients and the servers. We have one agent for each client machine. Each agent receives the requests from the clients and propagates them to the server. This is where our load balancing algorithm takes place.

Load Balancing

We implemented a round robin load balancing algorithm to fairly distribute the load among the servers. Round robin is a static scheduling algorithm that chooses the server to send the request to according to a predefined order. In other words, the servers are listed in a specific order and client requests are sent to them in that order. When the list is exhausted, the load balancer returns to the beginning of the list.

Agent Implementation

The agent has two processes: a parent process and a child process. In the parent process, a thread is created for each client request it receives. A separate thread is responsible for concatenating the client port number to the client request data. This is done to be able to send the reply back to the same client that sent the request. After adding the port number, this thread adds the modified request data to a queue. Another separate thread then sends each request in the queue to the server, according to the round robin load balancer. The load balancer only considers the currently active servers; it excludes any servers that are down at the moment of sending the request. The parent process gets the information regarding which servers are active or inactive from the child process.

The child process has a thread that receives the replies from the servers and places them into a queue. This thread also receives the “Server down” and “Server up” messages (mentioned in the message formats table below). Upon receipt, the thread updates the list of active servers, and sends it to the parent process through an OS pipe. Another thread in the child process then takes each reply from the queue, removes the extra concatenated client port number and sends it back to the client. The usage of queues in the agent does not create an overhead, as sending/receiving through the socket happens sequentially.

Servers

Network Topology

As mentioned above, our system has 3 servers. We have designed our servers to have a ring topology, where each server knows the addresses of the servers on its “right” and “left” sides. The main advantage of using a ring topology is that there is no need to store global information about all the servers in the network, which is efficient in terms of memory usage. This global information could be saved in every server in the network, which creates a huge storage overhead, especially in large networks. Also, this can cause inconsistency issues, in regards to updating the global tables in all the servers. To avoid the issues of inconsistency, this data could be stored in a centralized control unit, however, this creates a single point of failure for the system.Taking into account the aforementioned limitations, we chose to implement the ring topology for the servers.

Server Implementation

Each server can exchange messages with its 2 neighboring servers and with the agents. To test the capability of the system to tolerate server failure, we used a distributed election algorithm to periodically elect a server to go down for a certain amount of time. There is a separate thread in the server that is responsible for starting the election process every 60 seconds. The server contains an infinite loop that is continuously receiving messages from both the agents and the neighboring servers. The servers keep track of the addresses of the agents that have sent requests to that server so far. When a message is received, the server first checks the source address of this message. If the source address is not a neighboring server address, then this message is a client request. Therefore, the server first updates its local list of agent addresses, then proceeds to process the request. The client requests are of the format [1,2,3,xxx,xx], where xxxxx represents the client port number. Processing the request in our simulation is basically reversing the array [1,2,3] that was received, sending [3,2,1], leaving the client port unchanged. So the reply is of the format [3,2,1,xxx,xx]. The reply is sent back to the source agents.

The other case is that the message source is one of the neighboring servers. In this case, the message is concerned with the distributed elections algorithm which is discussed in the following subsection.

Fault Tolerance

To assess the behavior of our system in case of a server failure, we need to periodically choose one of the three servers to go down for a period of time. We implemented a distributed elections algorithm to do so. In each server, a separate thread is responsible for a repeating timer that fires every 60 seconds, triggering the start of a new election process. When the election first starts, the server sends an election message to its 2 neighboring addresses. When a server receives an election message from one of its neighbors, the message is stored in an array ELECTION, such that the election messages from the left neighbor and the right neighbor are stored in ELECTION[0] and ELECTION[1] respectively. The election message is of the format "e,<addr>,<val>"; where "e" indicates that it is an election message, <addr> is the address of the currently nominated server, and <val> is the value for this specific server. This value is used as the election criteria; this is the value upon which we could decide which of the servers would win each election. Since our project is implemented in the scope of a LAN, and only has 3 servers, it is not realistic and/or possible to collect meaningful values for the election criteria. Typically, the election criteria would consist of network connection strength, distance from other servers, etc. Therefore, we generate a random value in each server to be used as the election criteria. This value is regenerated differently in every election round.

For every election message received, the server updates the ELECTION array. The server then checks the values of the nominated servers in the array and if its own address is not yet nominated, it starts to consider itself too to be elected. The server then compares the value of the servers in the election messages with each other and with its own value. The server with the highest value would be the one it would nominate next. The server will then send an election message with the chosen server address and value in the aforementioned format. If the nominated server is one of the ones already in the ELECTION array, then the new election message will only be sent to one server, which is the neighbor that is not the same server that had sent the election message to begin with. In other words, when an election message is received from neighbor A, the server compares the values, decides on the current nominee, and sends it to neighbor B. If the server itself were not already one of the nominated servers and wants to nominate itself (because it has the highest value), it would send the new election message with its address and value to neighbors A and B, regardless of who had sent election messages before.

The election process ends when one of the servers receives election messages from both neighbors that are nominating the same server. This means that all servers have agreed on who should be elected next. When this happens, the server sends an election result message to its neighbors. This message has the format “r, <addr>, <val>”, where “r” indicates that this is an election result message, <addr> is the address of the elected server and <val> is the value of the server when it was elected.

When the election is done, each server checks if they are the one that was elected, and if that is the case, they go down for 15 seconds. When a server goes down, it sends a message to all the agents in its local array and to its 2 neighboring servers that it is going down. This message is just one byte “d”. This is one of the points where the agents update their arrays of the currently active servers. Within these 15 seconds of down time, the server does not respond to any messages. When the down time is done, the server generates a new random value for the next election round and informs the agents and its 2 neighbors that it is back up. This message is “u”. The agents, again, update their arrays of currently active servers.

Since the periodic timers in the servers are all local, they are not synchronized. Therefore, to avoid having overlapping elections, before a server can start a new election, it must check its local list of active servers to make sure that the previously elected server is now back up.

Message formats

Reason for Message	**Sender**	**Receiver**	**Format**	**Message meaning**
Election in progress	Server	Server	“e,<addr>,<val>”	The currently elected server has address <addr> and the value to compare is <val>
Election is done	Server	Server	“r,<addr>,<val>”	The server that won this election has address <addr> and value <val>
Server down	Server (the one that is going down)	Agent & Server	“d”	Informing the agents and the other servers that this server is going down. Address is known by checking the source of this message.

Server up	Server (the one that is back up)	Agent & Server	“u”	Informing the agents and other servers that this server is back up. Address is known by checking the source of this message.
Data (request)	Client	Agent	[1,2,3]	Requests sent from clients to servers are sent to the middleware (agent) first.
Data (request)	Agent	Server	[1,2,3,<xxx>,<xx>]	Data received from clients is sent to servers, and client port <xxxxx> is concatenated to the message for the agent to send the server reply back to the corresponding client.
Data (reply)	Server	Agent	[3,2,1,<xxx>,<xx>]	Data received from clients is mirrored and sent to servers, keeping the client port number<xxxxx> at the end of the message.
Data (reply)	Agent	Client	[3,2,1]	The reply is propagated from the server to the client, after removing the added client port from the data.

Distributed System Design Challenges

Designing a well-functioning distributed system is not an easy task, as there are many challenges associated with the design process. Below, we discuss some of the design challenges that we have addressed in our project.

Heterogeneity

Our implementation does not assume any properties of the hardware or software components of any of the nodes (clients, servers and agents). Our project can work on machines of different operating systems, for example.

Security

There are three main components of the security of the system: confidentiality, integrity and availability. For confidentiality, when the agent receives the data from a client, it stores the port number of that client and ensures that the reply is only sent to the client that sent the request. Any other data is not accessible to the clients. For integrity, the data intended for one client cannot be modified by another client. Finally, for availability, our system ensures that in case of server failure, data requested by clients would still be available and can be fetched by the other servers.

Scalability

Scalability is the ability of a system to increase or decrease in size. Regarding the number of clients, our system is scalable. This is because the servers dynamically collect the agent addresses when a request is received; so if a new client was introduced into the system, the agent would add its agent's address to its local list of agents and continue operation normally. However, when it comes to the number of servers, it is not so easy to add new servers to the system. This is a result of our use of the ring topology in our system. In our topology, each server has the addresses of its 2 neighbors hard-coded. The server addresses are also hard-coded in the agents. So if a new server was introduced, it would have to join the ring, which would alter some servers' neighbors, as well as informing the agents that a new server has been added.

Failure Handling

Our system has the ability to detect failures. When a server is elected to go down, it informs its neighboring servers and all the agents in the system. It also implements failure masking, as when a server fails, the agents no longer send requests to that server, so that failure is made less severe. However, we do not have the ability to recover from failure, as if an agent sends a request to a server before being informed that it is down, this message is lost.

Concurrency

Each node of our system (clients, agents and servers) is implemented in a multi-threaded manner, where each thread carries out different tasks at the same time. The

implementation details of each node was discussed in the Design & Implementation section above.

Transparency

Our system provides multiple types of transparency for its clients. One of which is access transparency, as the client sends its request to the local agent, which fetches information from the remote servers. This way the client accesses the remote servers the same way it would access local resources. Another type of transparency that is provided in the system is location transparency, as the client is unaware which server is replying to its request and where this server is located.

Evaluation

The main aim behind balancing the loads on the servers and handling server failures is to optimize the end-user (client) experience.

Methodology

Our objective is to have a fully-constructed standard to evaluate the effectiveness of our distributed system implementation. For each component, 3 servers, 1000 clients, and 2 agents, we have some schedule repeating to save some important values after x seconds and add them in files. We ran our code for 5 hours and 40 minutes on 5 different machines that are in the same LAN. The total number of requests in this simulation is 850,344,882.

Metrics

We used different metrics to ensure the performance and durability of our distributed system implementation. Our metrics used are (1)evaluating the effectiveness of **load balancer** by calculating the number of requests that each agent sends for each server. (2) calculating **the success rate** by just dividing the number of requests/the number of replies in the client code, (3) calculating the **average response time** as $\frac{number\ of\ replies}{sum\ of\ response\ time}$. To be able to get the desired results, we have collected the needed data as follows:

1. Agent

   Each agent is responsible for appending in the file the **number of requests** received from each server every **10 seconds**.

2. Server

   Each server is adding up the **number of requests** received from the agents for each **10 seconds.**

3. Client

   We do need to calculate the response time for each request. In this sense, for each accepted request- a request has been sent and received successfully- we calculate the response time. Also, to be able to calculate the success rate, we have appended in the file the number of requests sent to the agents and the number of requests received from the agents.

Results

The results below are based on data collected every 10 seconds for around 5 hours and 40 minutes. So, there are 2058 data points for each metric. The data collected at each point is an accumulation of all previous data.

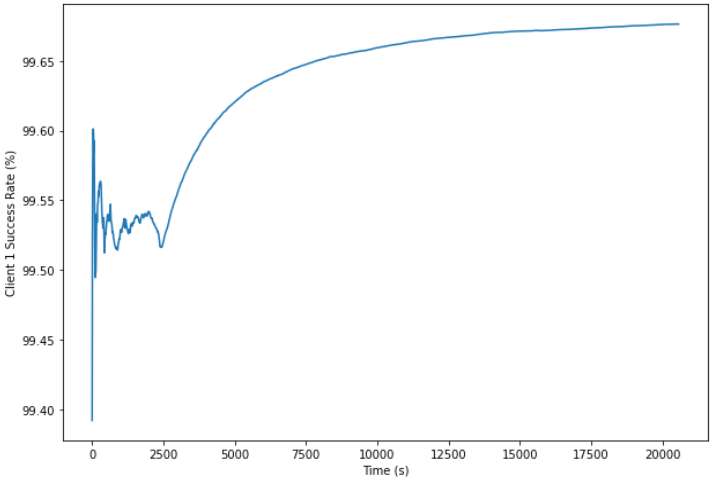
Client-Side

The response time is calculated as the time interval between the moment the sender first sends the request, until a response is received. The average response time is the sum of the response times divided by the number of responses received. The success rate is the number of responses received divided by the number of requests sent.

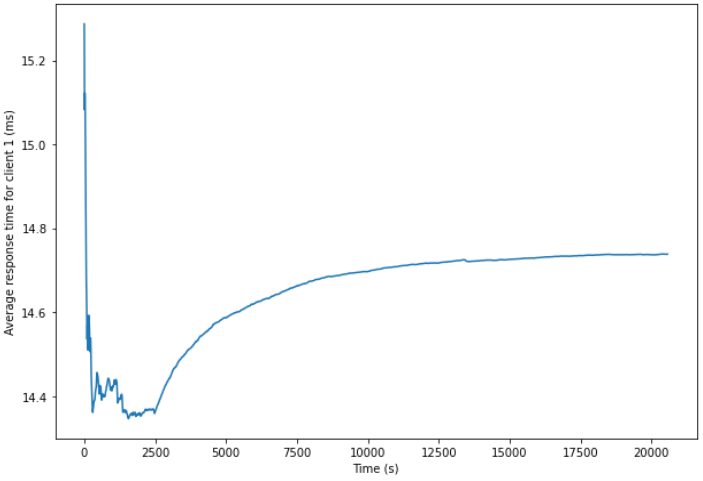
	Average Response Time (ms)	Success Rate (%)
Client 1 (500 threads)	14.74	99.68
Client 2 (500 threads)	14.18	99.70
Average Client in the System (all 1000 threads)	14.45	99.69

The plots below show how these metrics varied over time.

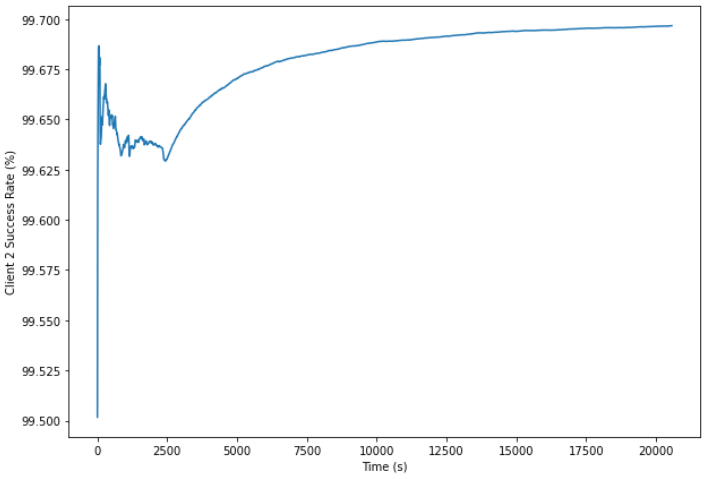
Success Rate of Client 1



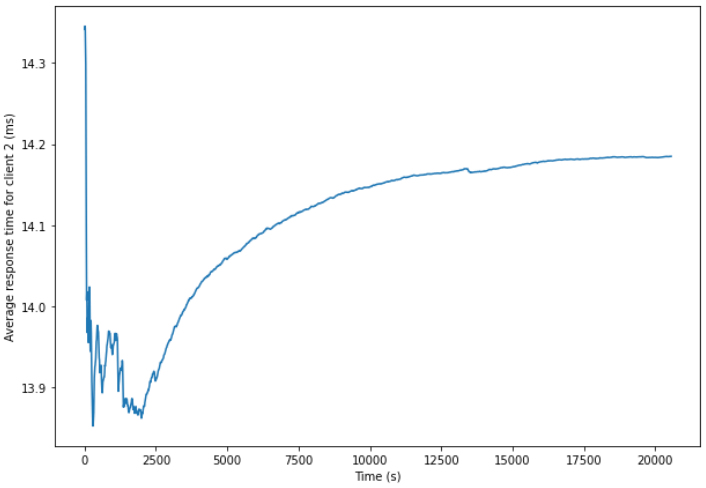
Average Response Time for Client 1



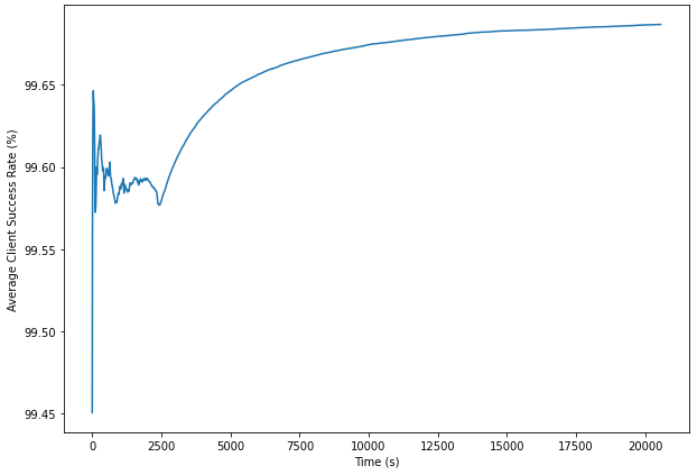
Success Rate of Client 2



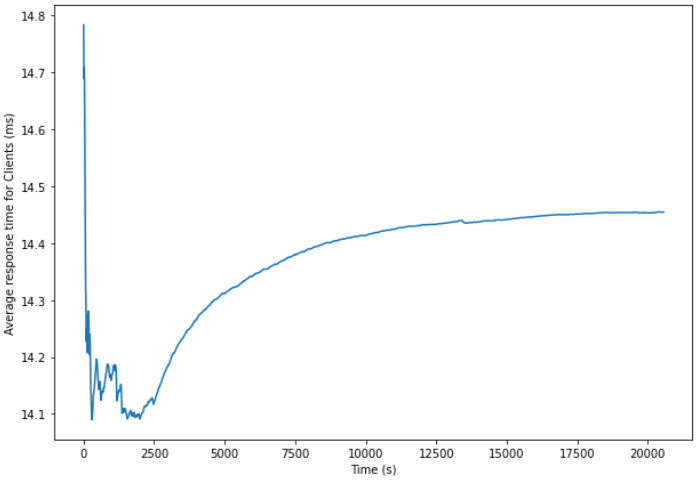
Average Response Time for Client 2



Success Rate of Average Client



Average Response Time for Average Client

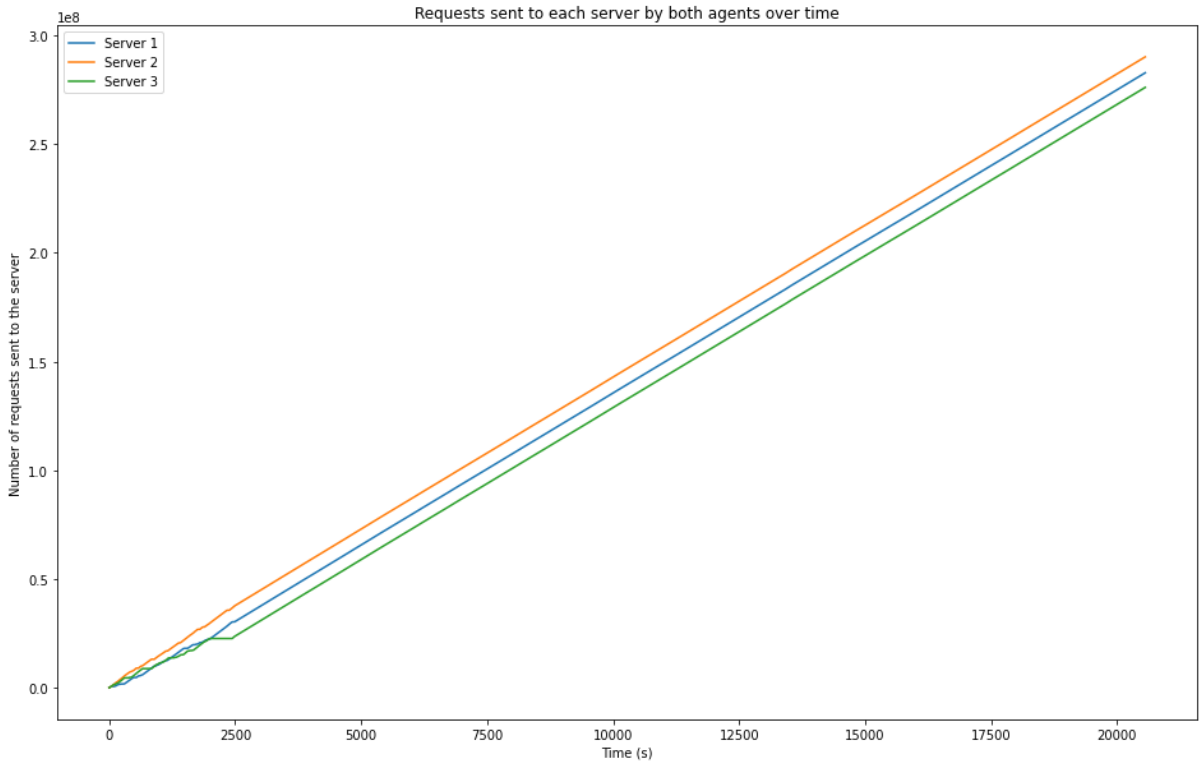


Agent-Side

To access the performance of the load balancer, the table below shows the ratios of the total requests that were sent to each server. This is based on the accumulated total requests at the end of our experiment.

	Ratio of Requests Sent from the Agent to Each Server		
	Server 1	Server 2	Server 3
Agent 1 (for client 1)	0.332	0.341	0.326
Agent 2 (for client 2)	0.334	0.342	0.324
Average of Agents	0.333	0.342	0.325

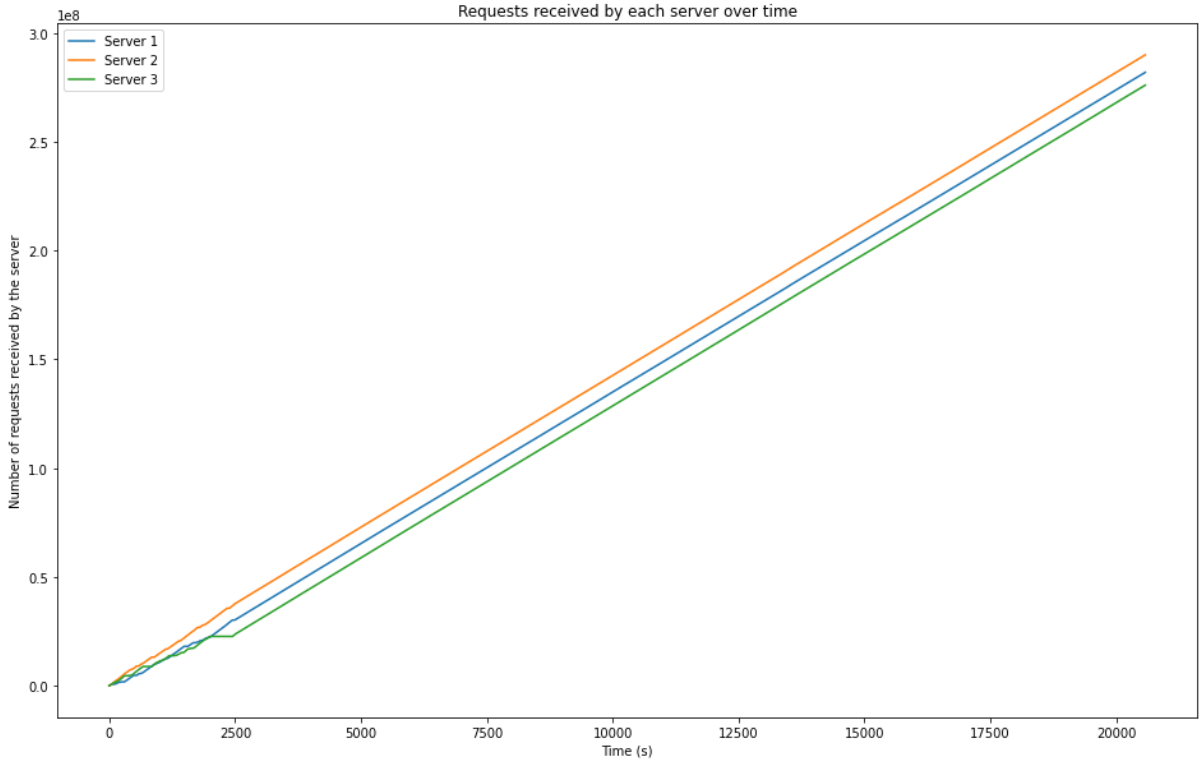
The plot below shows the number of requests sent to each server by both agents over time.



Server-Side

Even though the load balancer has distributed the load fairly over the servers, packets may be lost in the network for many reasons. Therefore, below we show the actual load on each of the servers.

Ratio of Requests Received by Each Server		
Server 1	Server 2	Server 3
0.332	0.341	0.326



Team Members Contributions

The work was divided among the team members at first, in order to have a faster pace and have the most productivity. Yassin worked on developing the client and agent, including the load balancing, Nada worked on developing the servers, including the distributed election algorithm, and Mohamed helped in both areas. After the initial design and implementation were done, the whole team would sit together to debug and adjust the project in order to meet the requirements and deliver the required functionality.