Client - Server Distributed Project

Prepared for

Dr. Amr El Kady

By

Kirolos Mikhail 900191250

Mariam Ali 900191779

Nour Montasser 900191101

Chapter 1: Design

This project presents a distributed system with a client-server model on a small scale. This system represents some traits associated with distributed systems, such as transparency, load balancing, and fault tolerance. Five machines were used. Two machines represent clients, and the other three machines represent servers. All the machines are on the same LAN. Clients and servers communicate through UDP sockets.

Furthermore, interprocess communication (IPC) between servers is through UDP sockets. To showcase the said traits of the distributed system, we use an election algorithm (discussed later) to choose a server to fail. When that server fails, we see how the system applies the load balancing algorithm (discussed later) between the two running servers. The project is entirely developed using Rust programming language.

1.1. Client

A Client is an entity responsible for bombarding the server with requests. The content of the request itself is meaningless, as long as messages are sent and received between the client and the server. Each client machine is threaded into 500 clients, where each client keeps sending requests constantly to the server. These requests are handled by the agent on each client machine before being forwarded to the server. The following is a pseudocode for the client.

```
Main
{
        Spawn thread for agent;
        let mut counter = 4000; //counter for ports to add to the ip for each new client
        let address = "10.7.57.77:";
        For client in 0..500
                Spawn thread and add it to threads vector
        {
                create values to store success, fails, and total time;
                create files for avg, failed and success per client per machine;
                new_address= address+counter;
                Loop
                {
                        Bind to new_address;
                        Data = “hello world”;
                        Start timer;
                        socket.send_to(data, "10.7.57.77:8080");       //send request to agent
                        //timeout so client does not stall waiting for response
                        socket.send_to(data.as_bytes(), "10.7.57.77:8080").expect("couldn't send data");
```

```
                    match socket.recv_from( buf) {
                Ok((_amt, _src)) =>       //if we received reply, request succeded
                    {
                            Collect statistics for successful data and store in files;

                    }
                    Err(_e) =>              //else the request failed
                    {
                            Collect statistics for failed data;
                    }


            }
            Counter ++;
    }));
}
```

1.2. Agent

The agent is considered the most important part of the middleware of this system. It has 3 pre-threaded processes, responsible for communicating with each one of the servers. An agent implements the round-robin algorithm for loading balancing (discussed later). Round robin determines which server will the request at hand be forwarded to using UDP on the corresponding pre-threaded process. The following is the pseudo-code for agent, the agent is a function in the client code.

```
Agent ()
{
        let thread1_addr = "10.7.57.77:8001";    //bind a socket for each pre-threaded process
        let thread2_addr = "10.7.57.77:8002";
        let thread3_addr = "10.7.57.77:8003";

        let agent_address = "10.7.57.77:8080";        //to receive requests from clients
        let agent_address2 = "10.7.57.77:8081";       //to send agent IP to server for notification

        let server1_addr = "10.7.57.176:8082";        //addresses to send requests to the servers
        let server2_addr = "10.7.57.73:8082";
        let server3_addr = "10.7.57.80:8082";

        let server1_addr2 = "10.7.57.176:8081";       //addresses to send agent ips
        let server2_addr2 = "10.7.57.73:8081";
```

```
let server3_addr2 = "10.7.57.80:8081";

//spawn the prethreaded thread
Bind on thread_addr;
Set socket read timeout to half a second;
Loop
{
        Receive requests from parent agent and send them to the bound socket;
        Wait for response:
         match socket.recv_from( buf)
        Ok((_amt, _src)) =>
        {
                send it to client;
                Write to file the number of request;
        }
        Err(_e){}



}
//the above code is replicated three times for each server using the above addresses
//respectively

Spawn a thread that informs servers of its IP and receives notifications;
{
        Create flags to determine if each server is up;
        Bind to agent address;          //recieve requests from clients
        Bind to agent address2;                 //recieve requests from clients
        socket2.send_to(b"tst", server1_addr2).unwrap();
        socket2.send_to(b"tst", server2_addr2).unwrap();
        socket2.send_to(b"tst", server3_addr2).unwrap();

        loop {
                socket2.recv_from( buf).unwrap();    //recieves notifications from server
                sender4.send(message).unwrap();              //send msg to parent thread
        }
}
//return to parent thread
Loop
{
        socket.recv_from( buf).unwrap();  //recieves requests from all clients in this machine
```

```
                Change flags according to msg received from sender4 of channel=
                {sender4,reciever4};

                //given the flags for the servers, check them
                Apply round robin;
                //if three servers are up, apply round robin normally
                //if A is down, apply round robin to B and C
                //…
        }
}
```

1.3. Server

A server communicates with other servers to implement the bully election algorithm (discussed later) to choose one of the servers to fail. Each server has a randomly assigned priority number. Each server broadcasts its priority number on using UDP to the other servers, that is where the IPC takes place. Then, the three servers determine based on who has the highest priority number.
Each server communicates with the agent to receive requests, receive the agents' IP addresses, or to tell the agent when they timeout and when they start running again. Each server has 4 threads and three channels for IPC between threads. We use two functions to generate and know when to broadcast priorities on the servers.

generate_priority_number()
This function uses rand::Rng to generate a random integer number of 32 bits from 1 to 101, and it returns that value to be used as a priority for one of the servers

check_time()
This function uses std::time::Duration to know the current global time. Whenever the seconds are exactly 00, in other words, the current time modulus 60 is equal to 0, it returns true. otherwise, it returns false. Whenever the return value of this function is true, the servers generate and send priorities.

The following is the pseudo-code of the server in terms of threads, and all threads are in a constant loop.

1.3.1. thread 1

```
{
        //Receive the agent's IP addresses from socket 3 on buf;
        let (amt, src) = socket3.recv_from(&mut buf).expect("Didn't recieve data");
        //Send the IP received to thread 4 through the sender in the channel= {ipagents_sender,
        //ipagents_reciever};
        ipagents_sender.send(src);

}
```

1.3.2. thread 2

```
{
        If check_time is true
        {
                Call generate_priority_number();

                //Send the priority number generated to thread 3 through the sender in the
                //channel={sender3, reciever3};
                sender3.send(priority_number);

                //Broadcast the priority number to the other two servers on socket2;
                socket2.send_to(priority_number, "10.7.57.73:8080").expect("couldn't send data");

                socket2.send_to(priority_number, "10.7.57.80:8080").expect("couldn't send data");

                Make the thread sleep for 1 second so it only generates the priority and broadcasts it
                once in the 00 second;
        }
}
```

1.3.3. thread 3

```
{

        //Receive priority from other server on socket1 through buf2;
        let (amt, src) = socket.recv_from(&mut buf2).expect("Didn't recieve data");

        //Receive priority from the second server on socket1 through buf4;
        let (amt2, src2) = socket.recv_from(&mut buf4).expect("Didn't recieve data");

        Typecast them from strings to integers and save them as data_int & data_int2;
```

```
        //Recieve the priority_number generated from thread 2 through the receiever in the
        //channel={sender3, reciever3};
        priority_number = receiver3.recv().unwrap();

        If priority_number > data_int
        {


                If priority_number > data_int2
                {
                        //server should go down
                        //Send “sleep” to thread 4 using channel={sender,reciever};
                        sender.send("sleep".to_string());
                }
        }

}
```

1.3.4. thread 4

```
{
        loop
        match agent IPs received through the channel= {ipagents_sender, ipagents_reciever}
        {
                ok(response) //if anything is received through the channel
                {
                        ip_agents.push(response);
                }
                Err(e)
                {
                        //used to print error for test if ips were received or not
                        println!(error receiving agents ip, e);
                }

        }
        //receive request from agents
        let (amt, src) = socket5.recv_from(&mut buf3).expect("Didn't recieve data");
        //if anything is sent on channel= {sender,reciever}, then this server is down
        if receiver.try_recv().is_ok()
```

```
        {
                for agent in &agent_ip
                        {
                        //send to agents that this server is down
                        socket5.send_to("a down", agent).expect("couldn't send data");
                        }
                        //timeout
                        sleep(Duration::from_secs(15));
                        //after 15 secs server will be up again
                for agent in &agent_ip
                {
                        socket5.send_to("a up", agent).expect("couldn't send data");
                }

                Continue;       //so it does not send back anything to client
        }

        //send reply to agent's request if the server is up
        socket5.send_to("message recieved", src).expect("couldn't send data");

}
```

Then, we join all threads.

1.4. Design shifts

This project has minimal design shifts. The election algorithm chosen for it was actually for electing a leader, not a server to time out. However, for the sake of testing the distributed environment, we used the election algorithm to time out a server. This way we can see how the system transparently responds to dynamic changes in its composition. Furthermore, the priorities of the server for the bully algorithm were generated randomly. Due to the small scale of our system, and the round robin algorithm for load balancing, if it was chosen to assign priority based on the number of processes handled by each server, it would coincide frequently that priorities of each server are the same, which is not our goal and could lead to unexpected behavior.

Chapter 2. Algorithms used

2.1. Load balancing (Round Robin)

Round robin assigns processes to servers using the round robin order. In other words, proc1 will be assigned to server1, proc2 for server2, proc3 for server3, and then we recirculate again where proc4 is assigned to server1.

2.2. Election (Bully)

The bully algorithm originally chooses the server with the highest priority to be the leader by using IPC between servers. However, as previously mentioned, it was modified to elect a server to time out where the priority is randomly generated using generate_priority, and the priority is communicated between servers whenever check_time () is true (discussed in section 1.3).

Chapter 3. Performance

3.1. Assessment

We assessed the performance of our system by collecting data (presented below) from the client machines, and saving them in files. For each client thread in the 500 thread per machine, we calculated the number of successes, failures, and average response time, and we save each of them in a separate file. Hence, each client has three files, and we have 1000 clients per the two machines, so we have around 3000 files to process. We used a parser to add up all the data obtained from the clients so we can obtain the statistics in the following table.

3.2. Statistics

	Machine 1	Machine 2	Overall
Average response time (/ms)	8	5	7
Total number of failed requests(/requests)	6,678,520	6,618,450	13,296,970
Average number of failed requests per client	13,357	13,237	13,297
Total number of successful requests	426,737,950	662,038,900	1,088,776,850
Average number of successful requests per client	853,476	1,324,078	1,088,777
Total number of requests to Server 1	149,576,110	232,135,150	381,711,260
Total number of requests to Server 1 (%)	35	35	35
Total number of requests to Server 2	149,251,380	231,817,370	381,068,756
Total number of requests to Server 2 (%)	35	35	35
Total number of requests to Server 3	129,192,860	199,216,700	328,409,560
Total number of requests to Server 3 (%)	30	30	30
Average number of requests per server	142,673,450	221,056,407	181,864,929
Total number of requests	433,416,470	668,657,350	1,102,073,820
Success rate	98.5	99	98.8

	Server 1	Server 2	Server 3	Total
Total number of requests received	381,069,000	381,711,000	328,409,000	1,091,189,000

Chapter 4. Contributions

The code for this project was developed collectively by all three of us. We dedicate at least one day per week where we sit and make strides in the code. We helped each other brainstorm ideas, and we helped one another when someone got stuck. We first tried two servers and one client on our laptops, and when the code was functional, we maximized it to 2 clients and 3 servers on the actual machines in the lab. Each one of us practiced their fair share of debugging

Chapter 5. Conclusion

This project has been very insightful and enriching. It has allowed us to discover a lot more about distributed systems than just theory. However, it had its limitations and, surprisingly, delimitations as well. As for the limitations, we discovered that there is a bug in rust with shutting down sockets. In other words, there is no such thing as destroying a connection between a client and server when the server times outs. The socket's connection is closed; However, it is still there, and it is still binded to the ip address and the port number. In our case, to work around this bug, when the server supposedly went down, it still received the clients' request but they did not respond. On the other hand, concerning the delimitations, we tried using three clients, instead of just two, and three servers in our system, and it worked perfectly. This implies that our code and system is capable of expansion, which is something that can be considered in the future.