Calculator as a Distributed System

Sebastian Fernandez Lopez, 0235314, Data Intelligence and Cybersecurity Engineering

Abstract— **A calculator as a distributed system that involves three main components: clients, MOMs and servers. Each server performs a single type of operation, the clients request the operations and the MOMs are the link between clients and servers dedicated to redirecting messages. Everything is connected in a mesh topology to make the system more fault tolerant and robust. Plus both the middleware and client have a queue to handle any outgoing messages asynchronously.**

***Keywords*— Distributed system - Sockets - Server - Clients - Calculator - MOM(Message-Oriented Middleware) - Mesh Topology - Network**

I. Introduction

A. Distributed systems

Distributed systems help us perform operations using different resources which might not be in a single place because of physical or efficiency complications. Instead of depending on a single system to perform these operations, a distributed system relies on the help of a network interconnecting multiple systems that serve a specific purpose. And by having them connected via a network they can use one another, perform conjoined or chain operations and become an overall more efficient system. It also comes with more advantages like being fault tolerant, highly available, concurrent, replicable and easily scalable since if a machine goes down another one can replace it and you can add more to the system as needed relatively easily. The only major disadvantages are security, since it is considerably more complex; synchronization process if two or more systems are accessing the same data, and most importantly they are highly dependent on the network infrastructure. If it goes down the whole system becomes unusable. This is where a mesh topology comes into play because by having multiple instances of the same components, if one goes down it can be instantly replaced by another one and allows for heavier traffic since the load can be balanced between all of them.

B. Objective of the project

With this project I seek to demonstrate the capabilities of an easy to understand implementation of a distributed system. The system I seek to create is a calculator as a distributed system connected in a local network; the idea behind this concept is to have a calculator divided in three parts: clients, MOM middlewares and servers. The purpose of the clients is to display a graphic interface in which any user connected to the local network can write an arithmetical operation and send it to the servers by clicking "=". The operation is then received (in a standardized manner which will be explained further in the report) by a Message-Oriented-Middleware, also referred as MOM. Each and everyone of these has 4 dedicated servers. The operation is analyzed by the MOM and depending on which type of operation it is, it will be redirected to one of its 4 servers. Each of these servers is exclusively capable of performing a single type of operation (addition, subtraction, multiplication and division). Whenever the corresponding server receives the request, it does the operation and then sends back the result (in a standardized manner) to the MOM which will then redirect the result to every client connected so every single one of them displays the same result. Every MOM is also connected with each other so every time one of these receives the message it is propagated between all of them.

C. Purpose of the report

With this report I seek to elaborate a little bit in the context and theory behind a project like this. In addition to the explanation in further detail of how this project was brought to life; from the conceptual design to the implementation in Java programming language and its complications which had to be solved.

II. The Project

A. Conceptual Design

Client— The client was the easiest to plan of all, since it only required to capture the user's input (with some restrictions) and send it to the MOM. I started planning the GUI of the calculator which includes buttons for 0-9, the four basic operators, an equal and C button to erase. As of now, the design only allows for one type of operation to be done but with as many operands as the user wants, i.e., it is possible to do "1+21+34+4" but not "5*6+7". The design of the client also included designing a standardized way of sending out the data as bytes. For this I opted for sending objects that would include an operation identifier and another object that contains a size and a list of operands. So the receiving end would do as many operations as the size of this list and the operator would be determined by the operation identifier. For this particular implementation where there can be different server sockets, the client will now be capable of randomly selecting an available port to connect with using a configuration file. And it will also have the ability to connect to the next available MOM if the one it is connected to goes down. If it cannot connect to any of the available MOMs meaning they're all down it will simply stop trying until the user does another operation and checks again for an available MOM.

MOM— The message-oriented middleware was the most crucial part of the design since a server or client might be down and still kind of work, but without the MOM nothing does. This middleware node had to receive structurized messages from either the clients or servers and redirect them to the right recipient. As of now, the distribution system simply redirects the clients messages to the corresponding server depending on the operator used and the server messages to every client. Whenever a socket was accepted the MOM had to either identify it as a server or client. I achieved this by sending an initial message when connecting from a client or server, either CLIENT or SERVER# (# represents one operator *+-/) and save every connection in their corresponding map. In the case of the servers, the map only allows four servers, one for each operator. So by using these maps, the MOM can choose which server or which clients will receive the message. Additionally, every middleware is interconnected with all the other nodes so that if any of them were to go down, another one can take its place. This also introduced an extra difficulty of how to propagate the messages through the network if two clients are connected to different MOMs but both of them have to receive the message, here is where the mesh topology comes into play.

Mesh Topology— In this section I will explain how the mesh topology was planned in order for this project to work. There are three components that can be instanced multiple times, the client, MOM and servers. The way the network will work is as follows: the main component is the MOM which each instance needs a port which will be randomly generated, every time a MOM is started its port will be stored in a configuration file that every MOM and client will have access to. So if another port other than this MOM's is in the file, this node will connect to every available port so they are all connected between them. At the same time each instance of the MOM will bring up four servers, one for each type of operator, so if for instance there are 4 MOM's instances there will also be 16 servers, 4 for each middleware. And then whenever a client's instance is initialized it will connect to a randomly chosen port from the configuration file. And if it goes down it will choose another port from the file until it can connect or there aren't any available.

Server— For the servers I designed a single project with four classes, one for each type of operation, and the project initializes four servers in different threads to simulate them being separated. They could be easily separated in different projects and each initializing just one type of server but for practical purposes I opted for the former. The server initializes but connects to the MOM with an operator identifier so the mom can store it. Whenever a client sends a request that matches its operation type, it deserializes the incoming object which will include the number of operands and a list with all of them. It then calculates the result and sends it back to the MOM. For every operation, since every server is only capable of performing one type of operation, three of the four servers will respond with a message like *"incapable of performing this operation"* and the one that can will respond with the result.

B. Implementation

Client— To implement the client I created a maven FXML project to have access to a GUI. The GUI consists of a single fxml file with buttons connected to onAction methods and a TextField associated with an fx:id to access and modify its properties from the controller. The application starts by initializing the graphic interface and to have the connection to the server always active, it initializes in a different thread. Which then again starts a listener in another thread. So the client runs on four synchronous threads; one handles the GUI and input/output, another one the output to the MOM, another one that is always listening to the incoming streams and finally one to handle the queuing and dequeuing of messages from and to the MOM node. I also created classes to standardize the incoming and outcoming messages. A MsgStrcut that stores a short to get the type of operation and a Message object. The Message object stores a short to get the number of operands passed and a List of shorts to store all the operands. There's also an OperationResult class that has two attributes, a String log and a Double result; the log is a string describing the performed operation and the result whatever came out of the operation. And finally a Streams class that has an ObjectInputStream and an ObjectOutputStream. This one serves the purpose of storing in the Client and server handlers their respective out and in streams instead of the socket itself. This makes it far easier to access the desired out or in from the MOM.

Now, to make it capable of connecting to a random port and handle reconnecting if any went down I had to add some logic to the application. When the application starts it needs to connect to a port, it first checks for the configuration file *"ports.config"* which has all the ports that are available (there are MOMs in those ports). After getting the ports list, it randomly connects to one of them. This is the easy but most important part and then the rest of the logic is to handle what to do when the MOM is down or lost. Whenever a MOM goes down it sends a last signal to the client in a specific form that the client doesn't recognize which tells it that the MOM disconnected. This event and the exception caused when the client can't send the message to the MOM trigger the same series of events. First, the ports list is updated from reading the configuration file to see if there are more ports available, then the index *i* which was the index of which port the client was connected with is incremented by one (keeping it in range with modulus) and tries to connect to that port. This can be repeated *n* amount of times with *n* being the number of ports, and if the client can't connect to any of them it stops trying and sends a log that there are no middlewares available. If the client tries to do another operation it tries again to connect to a port to see if there is a new one. This is all done with a try catch where the catch is triggered when the client can't connect to the server and here is where the "try again" logic happens.

Regarding the sending of messages to the middleware, instead of directly sending the messages after the user clicks the equals button, the message is added to a queue running in an independent thread. There are 4 different queues, one for each type or operation and each one of them is then periodically checked to see if it's not empty to deque the message it holds. When a message is dequeued a variable called *acuses* decreases by one and is only increased by one again when the client receives a *received confirmation* form the MOM. Before dequeuing any message it is first verified that the *acuses* variable is not less than the *min_acuses* global constant which is currently set to 1. This prevents sending many messages to a dead MOM, if the client does not get a received confirmation, it will not send any more messages. When an *acuse* is received, the *acuses* variable is increased and then the method that checks the 4 queues to see if there are any pending messages to be sent is called.

MOM— The MOM, being the most crucial component of the system, had to be implemented in a no-GUI Java application to avoid any possible interference or delay from managing the graphic interface thread. For input and output I decided to use ObjectStreams to send serialized objects to have a more manageable and easier way to read the input. The MOM simply has a thread handling the outputs and another one actively listening at all times. This middleware has three types of handlers, ClientHandler, ServerHandler and OtherMOMHandler which handle either type of connection in a different thread. When any of the handlers receives a message transforms with the help of a cast to either a MsgStruct or an OperationResult for the ClientHandler and ServerHandler respectively, MOMHandler also receives OperationResult objects (which can have a message in the log of how a particular server wasn't able to perform the operation). In the case of the client handlers, it sends the message to the corresponding server based on the type of operation and in the case of the server handlers, it sends the result to every client.

In order to handle other MOMs connecting between them I took a similar approach as how the client connects to a MOM. But in contrast to the clients, MOMs don't just connect to one other MOM, they connect to every MOM available. So this is what I did, when the MOM is initialized it first starts four servers for each operator and gives them its port so they can try and connect to it. After this, the MOM, after modifying the ports configuration file adding the port it was created on, reads it again and iterates over the available ports (except for the one it is on) and sends a connection request to all of them with a first message of "MOM" which tells the other nodes to store this socket in the other_moms_hashmap. Which will be useful to propagate messages.

In order to achieve the message propagation between all of the active nodes, whenever a server from one of the MOM instances sends it a result which needs to be received by the clients, it also sends it to all the connected MOMs which are stored in that node's hashmap, connected_moms. And similarly when a MOM receives an OperationResult from another node, it sends it to the clients connected to that port by iterating over the hashmap storing all those sockets.

Similar to the queue handler that helps the client, there is a QueueHandler thread in each MOM as well. In this case the queue stores a special structure holding the object to send (either OperationResult or MsgStruct) and an ObjectOutputstream in case a received confirmation message needs to be sent to a specific client. When there is a message queued, it first checks the object's type to either send it to the clients and middlewares (OperationResult) or to the operation servers (MsgStruct). And when a received confirmation needs to be sent it needs to create a special OperationResult which reads in the log attribute *"Acuse de recibido"*.

Server**—** The structure of all 4 servers is the same. It has a Main class that takes a port as an argument and creates the connection socket with the MOM in this port (the port is given by the MOM that started these 4 servers) and a class dedicated to the logic needed for each one. The addition server does additions inside a loop of all the operands and similarly with the other three operators. After performing the arithmetic operation the result is sent out to the MOM in an OperationResult object. All the servers receive each operation request but evidently only one of them is capable of solving the operation. Si if they can't simply send an OperationResult object with an empty result and a *"Server incapable of solving this operation"* so for every operation sent to the servers three of them will respond negatively and one with the correct result. It is this result which will be propagated among the mesh.

Models**—** It's worth mentioning that every project has the same definition of the 4 classes explained in the Client section so that every component of the system can either serialize or deserialize them. In addition, to make this possible, every project has the same package name so that it recognizes every input and output object as the same class.

III. Conclusion

Wrapping up, the hardest part of making this project was getting to the part of how the input and output behaves when working with sockets and how the messages need to be propagated among all the connected clients and MOM middlewares. Not to mention the logic needed to handle the client's reconnecting to another MOM if the one it is connected to goes down. Once I got over this obstacle, everything was relatively easy. Through the elaboration of this project other issues that showed up were handling the threads and different instances of the graphic interface. But leaving all the complications and tears behind, I learned a lot coding this system. This was my first interaction with both Java and distributed systems and although the learning curve was very steep, it was very rewarding to successfully finish it. Distributed systems solve a whole bunch of problems and that is worth the added complexity of developing them.

IV. References

[1] C. Kidd, "Distributed Systems Explained," Splunk, https://www.splunk.com/en_us/blog/learn/distributed-systems.html#:~:text=A%20distributed%20system%20is%20simply,a%20single%20device%20ran%20it. (accessed Sep. 12, 2023).

[2] "What is message oriented middleware (MOM)?," GeeksforGeeks, https://www.geeksforgeeks.org/what-is-message-oriented-middleware-mom/ (accessed Sep. 12, 2023).

[3] J. Meza "Sockets en Java" Programarya, https://www.programarya.com/Cursos-Avanzados/Java/Sockets (accesed Sep. 12, 2023)