Calculator with an Autonomous-Distributed Service-Oriented Architecture (ADSOA)

Alfonso Murillo Suárez
Information Technologies &
Intelligent Systems Engineering
0203278@up.edu.mx

I. Description of the project

This project incorporates several functionalities developed through the Distributed Computing course. The final deliverable is a calculator with an autonomous distributed service-oriented architecture (ADSOA).
For achieving the objective of this project, we developed four main agents:

- **Graphical client:** this is a graphical interface that allows the user to input operations and to see the results after a server returns them. It is represented as a calculator in the diagram shown in Fig. 1.
- **Server:** represented as circles in the diagram. They are the ones that receive the operations and return the results. Each server is capable of executing certain operations, but at the beginning, the servers are not able to do any of them since they do not have the microprograms that resolve the operations.
- **Manager:** it is a single console program, shown as a computer in the diagram, that knows the location in the hard drive of all the microprograms for each operation (services shown as little squares below the computer). When the user selects one operation, the manager sends the program through the data field to one specific server for it to be able to perform it.
- **Node:** one or more nodes (square elements) make up the data field (dashed circle), which allows the communication between all the cells (clients, servers, and manager).

The system also incorporates an *auto-recovery* property for having a minimum number of servers capable of executing each operation.

Throughout the report I will explain, first, the different deliverables made, and finally the modifications for accomplishing this final project.

II. Previous deliverables

A. First Deliverable: Distributed Calculator

This first deliverable was mainly to develop the data field, which is the core of the communication between the cells.
When a node is created it reserves 50 available ports between 6000 and 6999 for using them to manage the communication between other nodes and the cells. For each other agent, the node creates a thread to manage its communication.

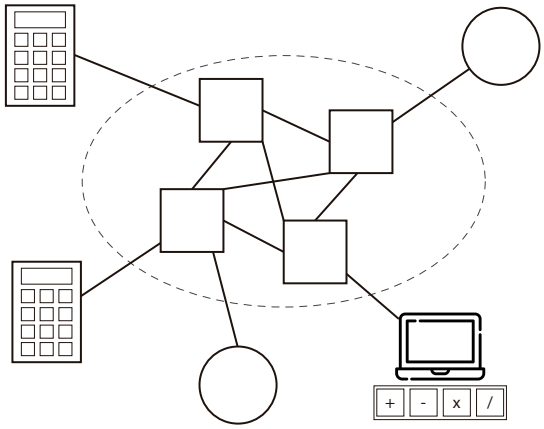

Fig. 1. Diagram of the final ADSOA calculator

When a cell is created, it searches randomly for a node and connects to it, so it is its main point of contact with the data field. When it sends a message to the data field, the message is broadcasted, which means that all the clients receive the message. There are two simple rules to achieve the broadcast:

1) If a node receives a message from a cell, it sends the message to all the other nodes and cells connected to it.
2) If a node receives a message from another node, it sends the message only to the cells connected to it.

The cells are able to accept or reject messages because all of them include a Content Code, which indicates the type of information contained in it.
This deliverable only had the client cell and the server cell, but the server was able to perform all the operations with its own code. Diagram in Fig. 2 shows the architecture of this deliverable.

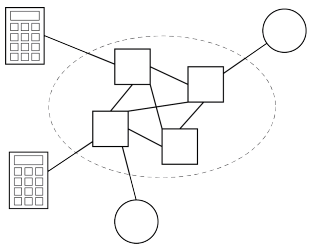

Fig. 2. Diagram of the distributed calculator

B. Second Deliverable: Service-Oriented Architecture and Auto-Recovery

Once the basic infrastructure of the first deliverable was working correctly, the next step was to implement the *service-oriented architecture* and the *auto-recovery protocol*. First I will explain the auto-recovery protocol.

The system needs, at least, three servers capable of performing each kind of operation. The client, when it sends an operation, waits until it receives three receipts from different servers, which indicates that there are enough for the system to work properly. If the client does not receive the minimum amount of receipts, it makes two other attempts, and then it selects a server and tells him to duplicate itself. All the clients that are requesting the same kind of operation select the same server by alphabetically sorting the footprints and taking the first one, avoiding cancer.

When an operation is unavailable all the other operations of the same type are held in a queue, while the client waits until the system has recovered. The other operations, if there are enough servers to perform them, keep working without any problem since each of them has a dedicated thread. Figure 3 shows the architecture of the client with different threads.

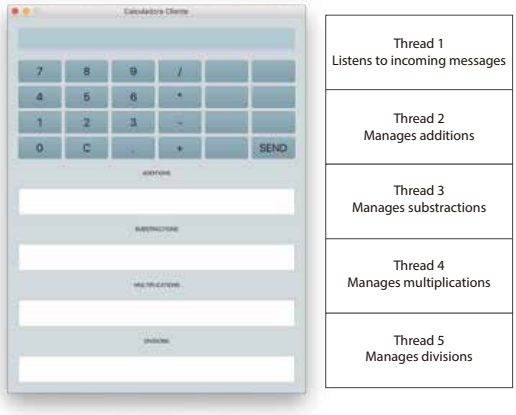

Fig. 3. Different threads for the client

The service-oriented architecture consists of separating the operations from the server. The operation executable files are stored in the hard drive and, when they are needed, the server loads them dynamically. This has several advantages. Some of them are:

- The server program is lighter, so it uses fewer RAM memory.
- The operations can be easily modified without having to take down the whole server. item If we need new operations, they can be developed separately and the changes in the main server code are minimal.
- The servers can be focused on specific operations instead of all of them, so we can distribute the workload depending on the most used operations.

These service-oriented servers can be represented as shown in Fig. 4, where all the operations are stored in the hard drive and the server is capable of loading them only when they are needed.

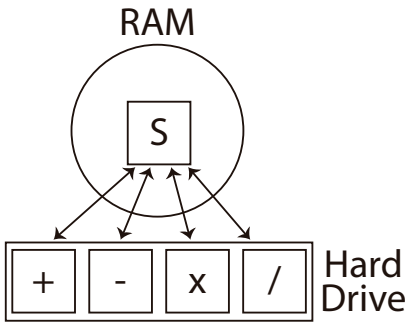

Fig. 4. Service-oriented server

Through command-line arguments, we can create servers that perform only specific operations, as mentioned in the list of advantages.

III. Final Deliverable

The final deliverable is based on the previous developments, but now an extra type of cell is added: the manager. This cell connects to the data field and consists of a command-line program that allows the user to select an operation and send it to the servers.

Now, when a new server is created, it does not know what operations it can perform because no executable files are created in its folder, so the manager is able to select a random server and send a specific operation's jar to it. I will now explain the modifications made to the previous versions of the system to accomplish these new specifications.

A. Client must be able to manage 0 receipts

When the client did not receive the minimum amount of receipts, it selected a server to duplicate itself, but now there might not be any server capable of performing the operation, so no receipts are sent back.

When the client detects this, it continues trying to obtain the operation's result by resending it multiple times, but it does not ask any server to duplicate, since there are no servers available for that operation.

B. Servers are created empty

As mentioned before, when a new server is created it is not able to perform any operation, since it does not count with the needed jar files. The server accepts the content codes for each type of operation but an extra indicator is set for it to know if it is capable of performing the operation or not before sending back a receipt. Figure 5 shows the process for this validation. The servers can also accept content codes for indicating that they are capable of receiving a specific operation (when they do not have the jar file already) and another content code indicating that the message contains the jar file for the operation. In section III-C, I detail all the possible content codes that go through the system, and in section III-E I explain how the jar files are sent through the data field.

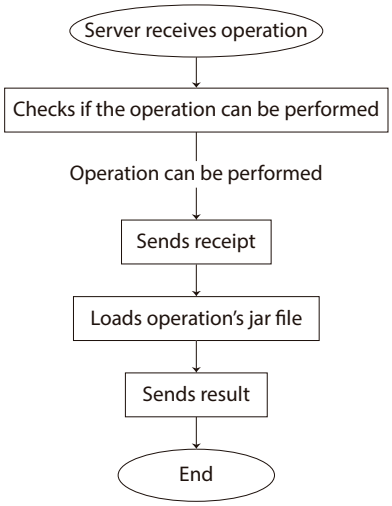

Fig. 5. Server's process for managing incoming operations

C. Content Codes

Every message in the system has a content code that indicates what information it is carrying. Table I show the original content codes used for the previous deliverables. These content codes were for messages sending operations, results, receipts, and duplication requests.
Each message contains an event, which is a unique identifier for each message, that helps the cells identifying their messages.

TABLE I
ORIGINAL CONTENT CODES

Operations		
Content Code	**Message Content**	**Receiver**
1	Addition operation	Servers
2	Substraction operation	Servers
3	Multiplication operation	Servers
4	Division operation	Servers
Results		
Content Code	**Message Content**	**Receiver**
5	Addition result	Clients
6	Substraction result	Clients
7	Multiplication result	Clients
8	Division result	Clients
Receipts		
Content Code	**Message Content**	**Receiver**
101	Addition receipt	Clients
102	Substraction receipt	Clients
103	Multiplication receipt	Clients
104	Division receipt	Clients
Duplication request		
Content Code	**Message Content**	**Receiver**
500	Duplication request	Servers

Table II shows the new content codes used for this new deliverable. First, it shows the content codes for searching for recipients for operations. When a server is not able to perform an operation because it does not have the necessary jar file, it is going to accept the *searching for recipient* code for that operation.
A server that is able to accept an operation will send back a message containing the *accepting operation* content code for that operation and its footprint. In section III-D I explain how the manager handles these codes.
When the manager sends the jar file, it is sent with a *jar file* content code for the operation.

TABLE II
NEW CONTENT CODES

Searching for recipients		
Content Code	**Message Content**	**Receiver**
501	Searching sum recipient	Servers
502	Searching substraction recipient	Servers
503	Searching multiplication recipient	Servers
504	Searching division recipient	Servers
Jar files		
Content Code	**Message Content**	**Receiver**
511	Sum's jar	Servers
512	Substraction's jar	Servers
513	Multiplication's jar	Servers
514	Division's jar	Servers
Accepting operations		
Content Code	**Message Content**	**Receiver**
511	Accepting sum	Manager
512	Accepting substraction	Manager
513	Accepting multiplication	Manager
514	Accepting division	Manager

D. Manager

The manager is a command-line program that knows the location in the hard drive of the different jar files for the operations. Diagram in Fig. 6 shows the architecture of this cell.

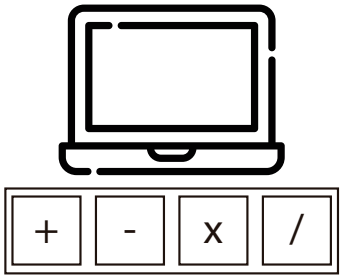

Fig. 6. Architecture of the manager cell

The program offers a simple menu where the user is able to write the operation he wants to send and then the manager performs the next steps:

1) The manager detects the operation to be sent.
2) The manager sends a message with the corresponding *searching for recipients* code for the operation.
3) The manager receives the response from the servers that can accept the operation because they do not know how to implement it (message with an *accepting operation* content code). This message contains the footprints of the servers.

4) The manager sorts alphabetically the received footprints and takes the first one.
5) The manager loads the file and sends it in a message with the structure shown in Fig. 7, with the *jar files* content code for the operation it is sending (*CC*), the footprint of the selected server, and the encripted jar file.

Fig. 7. Structure of the message sent with the jar file

6) The manager restarts and shows the principal menu after successfully sending the operation.

The manager, as mentioned before, is a new type of cell that connects to the data field and is capable of sending and receiving messages with the broadcasting protocol. The way it loads the jar file is described in the following section (III-E).

E. Sending Jar files through the data field

The communication protocol, since the beginning, was designed for sending string messages, so being able to send a jar file through the system was a challenge.
To accomplish this, the manager has a function that loads the jar file from the known location to a byte array, then, this byte array is ciphered to a base64 string for being able to send it through the data field.
The servers then receive this message, each of them checks if the footprint is their own (so that they are the recipients of the file) and they are able to decipher the base64 string to a byte array, and then saving this byte array to a file in their own directory. Once this is made correctly, the server changes its indicator for that operation for being able to accept it from now on.

IV. Results

The final project works as it was intended to. The clients are able to work the operations independently according to the available servers. When a certain operation does not have a server to work it then they are stored in a queue attempting to be solved when a server is available.
Once there are available servers for the operations, if they are not enough, the clients tell a server to duplicate itself until the minimum number of receipts is reached.
The manager is able to send perfectly the selected operation to a determined server that is still not able to make that operation. Every time the server is restarted, even if they have the jar files in its folder, it is not able to perform the operation until the manager sends it. There is no problem if there is already an operation, it overwrites the existing one.

V. Project's code

The code for the final deliverable can be found in this Github repository (click me).