



UNIVERSIDAD DE CASTILLA-LA MANCHA
ESCUELA SUPERIOR DE INFORMÁTICA

GRADO EN INGENIERÍA INFORMÁTICA
COMPUTACIÓN

SISTEMAS MULTIAGENTES
BOT CONVERSACIONAL

María Blanco González-Mohíno

Enero, 2022

SISTEMAS MULTIAGENTES

© María Blanco González-Mohino, 2022

Este documento se distribuye con licencia CC BY-NC-SA 4.0. El texto completo de la licencia puede obtenerse en <https://creativecommons.org/licenses/by-nc-sa/4.0/>.

La copia y distribución de esta obra está permitida en todo el mundo, sin regalías y por cualquier medio, siempre que esta nota sea preservada. Se concede permiso para copiar y distribuir traducciones de este libro desde el español original a otro idioma, siempre que la traducción sea aprobada por el autor del libro y tanto el aviso de copyright como esta nota de permiso, sean preservados en todas las copias.

Este texto ha sido preparado con la plantilla \LaTeX de TFG para la UCLM publicada por [Jesús Salido](#) en GitHub¹ y Overleaf² como parte del curso « *\LaTeX esencial para preparación de TFG, Tesis y otros documentos académicos*» impartido en la Escuela Superior de Informática de la Universidad de Castilla-La Mancha.

¹https://github.com/JesusSalido/TFG_ESI_UCLM, DOI: 10.5281/zenodo.4574562

²<https://www.overleaf.com/latex/templates/plantilla-de-tfg-escuela-superior-de-informatica-uclm/phjgscmfqtsw>

SISTEMAS MULTIAGENTES

María Blanco González-Mohino
Ciudad Real, Enero 2022

Resumen

En este trabajo se aborda la problemática de construir un *bot* conversacional. Esta opción ha sido desarrollada mediante un sistema multiagentes, se ha utilizado la plataforma de sistemas multiagentes [Spade](#), escrita en Python y basada en la mensajería instantánea (XMPP). En este documento repasaremos diferentes facciones del *bot*, iremos desde la arquitectura empleada en su desarrollo, hasta el manual de usuario, pasando por los protocolos de interacción de los comportamientos entre agentes y las decisiones que se han tomado a la hora de su desarrollo.

Se han implementado diferentes tecnologías como bases de datos de SQLITE3 y contenedores Docker.

IMPORTANTE: la página web [404.city](#) borró mis credenciales para este *chatbot* y no me dejó crear unas nuevas, por lo que se han utilizado las suministradas para la introducción.

Índice general

Resumen	IV
Índice de figuras	VII
Índice de tablas	IX
Índice de listados	IX
1. Arquitectura del sistema	1
1.1. Emisor/Usuario	1
1.2. Receptor/Chatbot	3
1.3. Scripts adicionales	4
1.4. Docker y SQLite	4
1.5. Distribución del proyecto	5
Índice de algoritmos	1
2. Decisiones	7
2.1. Sobre el Chatbot	7
2.2. Sobre SQLite y Docker	8
3. Protocolos de interacción	9
3.1. TimeBehav	9
3.2. WhoIs	10
3.3. CreateFile	11
3.4. History	12
3.5. Download	13
3.6. Facial	14
3.7. MemeCreator	15
3.8. Exit	15
4. Despliegue y Manual de Usuario	17
4.1. Prerrequisitos en local	17
4.2. Despliegue en Docker	17
4.3. Manual de Usuario	19
Bibliografía	27

Índice de figuras

1.1.	DB imágenes	4
1.2.	DB urls y personas	4
1.3.	Distribución del proyecto	5
3.1.	Protocolo Tiempo	9
3.2.	Protocolo Personas	10
3.3.	Protocolo Fichero	11
3.4.	Protocolo Historia	12
3.5.	Protocolo Descargar	13
3.6.	Protocolo Detección Facial	14
3.7.	Protocolo Meme	15
3.8.	Protocolo Exit	15
4.1.	Pedir comandos	19
4.2.	Pedir comandos	19
4.3.	Ejecutando TimeBehav	19
4.4.	Ejecutando WhoIs	20
4.5.	Ejecutando WhoIs web	20
4.6.	Ejecutando WhoIs nadie	20
4.7.	Ejecutando CreateFile con contenido	21
4.8.	Ejecutando CreateFile sin contenido	21
4.9.	Ejecutando Download	22
4.10.	Download ejecutado	22
4.11.	Ejecutando History	22
4.12.	Ejecutando Meme	23
4.13.	Figura Meme	23
4.14.	Ejecutando Meme con ruta	24
4.15.	Figura Meme 2	24
4.16.	Ejecutando Exit	25
4.17.	Ejecutando Exit CTRL+C	25

Índice de listados

1.1.	Envío de mensaje	1
1.2.	Inicialización Response	2
4.1.	Instalación FFMPEG	17
4.2.	Comprobación FFMPEG	17
4.3.	Instalación X11	17
4.4.	Host X11	17
4.5.	Crear imagen Docker	18
4.6.	Crear imagen Docker	18
4.7.	Ejecución Docker	18

Arquitectura del sistema

En este apartado se aborda la arquitectura empleada en el *chatbot*, se plantearán los diferentes agentes así como los comportamientos que los componen y las conexiones entre estos.

Dado que el sistema se ha desarrollado en Spade, nos encontramos un método *main*, llamado **Chatbot.py** y dos ficheros **SenderAgent.py** y **ReceiverAgent.py** que se corresponden con el **usuario**, persona o máquina que interactúa con el *chat* y con el *bot* conversacional, respectivamente.

1.1. EMISOR/USUARIO

El agente **usuario** es el encargado de detectar los diferentes comandos que podemos introducir para interactuar con el *chatbot*, también se ocupa mostrar las diferentes respuestas del *chat*, este agente también es el encargado de apagar ambos agentes.

1.1.1. Comportamientos

1. **Init**. Comportamiento de inicialización del *chatbot*, este únicamente manda un mensaje al comportamiento **request** para que comiencen las peticiones de comandos.
2. **Request**. Este comportamiento es el encargado de ejecutar los diferentes comandos con los que podemos interactuar con el *chatbot*. Puede aceptar los siguientes comandos:

- | | |
|----------------------------|--------------------|
| (1) show me the time | (5) history |
| (2) who is 'famous person' | (6) face detection |
| (3) file | (7) meme creator |
| (4) download 'url' | (8) exit |

En este comportamiento también se ha incluido un comando **help** el cual no interactúa con el *chatbot* y nos indica los diferentes comandos que podemos utilizar para la interacción.

Para cada uno de los diferentes comandos el agente emisor enviará un mensaje con una serie de características, como el protocolo a seguir o el acto performativo del mensaje; para los comandos que se necesita algún tipo de información como la persona a buscar se introducirá esta información en el cuerpo del mensaje.

Listado 1.1: Envío de mensaje

```
1 msg = Message(to=data['spade_intro_2']['username'],
2               sender=data['spade_intro']['username'])
3 msg.set_metadata("performative", "request")
4 msg.set_metadata("protocol", "file")
5 msg.body = body
6
7 await self.send(msg)
```

En este comportamiento también se ha incluido un procesado natural de lenguaje, el cual cuando introducimos una palabra mal escrita, nos indica una de las posibles opciones para su corrección pudiendo indicar si el comando a introducir es correcto o no.

3. **Response.** Este comportamiento es el encargado de recibir las contestaciones del *chatbot*, en el se distingue si la contestación es de tipo 'inform' o 'failure' y se imprime la respuesta del *bot* sea la que sea.

Este comportamiento es inicializado por cada uno de las funcionalidades, es decir, tenemos el **mismo comportamiento** para la funcionalidad 'file' que para la 'history' con protocolos diferentes, por lo que tenemos un comportamiento activo por cada comando, como ya hemos mencionado, esto es, 8 comportamientos response. Ejemplo gráfico:

Listado 1.2: Inicialización Response

```
1 # Plantillas para el protocolo
2 template_file = __create_template_user__("file")
3 template_history = __create_template_user__("history")
4
5 # Inicialización del comportamiento
6 self.add_behaviour(self.Responses(), template_file)
7 self.add_behaviour(self.Responses(), template_history)
```

1.2. RECEPTOR/CHATBOT

El agente receptor es el encargado de ejecutar las diferentes funcionalidades implementadas y mandar la respuesta correcta si se ha podido llevar a cabo la funcionalidad sin ningún tipo de problema o mandar una señal de fallo con la especificación sobre el error ocurrido.

En este apartado se explicarán cada una de las funcionalidades y como ha sido su desarrollo. Cada una de las funcionalidades se asocia con un comportamiento diferente.

1.2.1. Comportamientos

1. **TimeBehav.** Este comportamiento muestra la hora, día, mes y año de ejecución del comportamiento y envía esta hora al agente emisor para que la muestre al usuario. Comando necesario: **show me the time.**
2. **WhoIs.** El comando **who is** seguido de el nombre de una persona nos busca información sobre esta, si esa persona ya se encuentra introducida en nuestra base de datos nos mandará el resultado ya almacenado de la búsqueda indicando que ya se encuentra en la base de datos. Si la persona no se encuentra almacenada en la base de datos, la buscará en *Internet* y nos indicará si ha sido posible encontrarla (mandándonos información respecto a la persona) o si no se ha podido encontrar. Esta funcionalidad también actualiza el número de búsquedas de cada persona en nuestra base de datos.
3. **CreateFile.** Una vez ejecutado el comando, el **bot** nos mandará un mensaje para que introduzcamos el nombre del fichero a crear y la ruta donde lo querremos almacenar, una vez introducida la información le enviaremos un mensaje para su creación. Comando necesario: **file**
4. **Download.** Este comportamiento recoge una url de Youtube y descarga el contenido de esa url. Una vez este contenido es recogido se envía un mensaje tipo *request* al usuario en el que le pide que introduzca el nombre del archivo con su correspondiente extensión, cuando obtiene el nombre y la extensión del archivo, el *chatbot* almacena el archivo descargado en la carpeta *música*. Este comportamiento utiliza el comando **download, Youtube** o una **url**.
5. **History.** Este comando realiza una búsqueda en la base de datos y nos envía la persona que más ha sido buscada, el número de veces que se ha buscado y lo que encontró respecto a esa persona. El comando necesario para su ejecución es **history**.
6. **Facial.** Este comportamiento inicia un reconocedor facial, cuando el usuario cierra el reconocedor (barra espaciadora), el comportamiento termina. Para ejecutar este comportamiento se utiliza el comando **facial detection**.
7. **MemeCreator.** El comando **meme creator** crea un meme con tu cara. El agente **bot** nos da la opción de añadir la ruta de la imagen que queremos poner en el meme, si no se añade ninguna ruta, el *chatbot* elegirá una imagen aleatoria de la base de datos. Si la ruta es añadida, el *chatbot* creará el meme e introducirá la imagen en la base de datos. Este meme es almacenado en la carpeta *random*.
8. **ShutDown.** Comportamiento utilizado para recibir la orden de apagar el agente receptor desde el agente emisor. Este comportamiento se apaga y manda un mensaje de éxito al emisor, el cual también se apaga después de su recepción. Comando: **exit**.

Cuando no se introduce ningún comando, es decir, insertamos un texto sin letras, se ejecutará el comportamiento **TimeBehav**.

1.3. SCRIPTS ADICIONALES

Para el desarrollo de los comportamientos más complejos se han añadido diferentes *scripts* de manera que las funcionalidades se recogen a modo de librería en diferentes archivos.

A continuación se incluye una lista con los *scripts* que se han añadido y los comportamientos que lo usan.

1. *broker.py* : conexión a la base de datos. Usado por los comportamientos 'WhoIs', 'History' y 'MemeCreator'.
2. *face_detection.py* : 'Facial'. Utiliza la interfaz de pantalla del host para reconocer caras.
3. *meme_creator.py* : 'MemeCreator'. Creador de memes a partir de una foto introducida o una guardada en la base de datos.
4. *music.py* : 'Download'. Descargador musical según la url.
5. *scraper.py* : 'WhoIs'. Búsqueda de personas utilizando *web scrapping*

1.4. DOCKER Y SQLITE

Se ha añadido una base de datos [SQLite](#) para el almacenamiento de imágenes, búsquedas de persona y almacenamiento de *urls* con las que se puede hacer *web scrapping*.

Para crear esta base de datos se encuentran en el archivo *broker.py* dos métodos, *createDB()*, para la creación de la base de datos, y *createTables()* para la creación de las tablas de la base de datos.

images			CREATE TABLE images (id integer PRIMARY KEY AUTOINCREMENT, image_name text, image blob)
id	integer	"id" integer	
image_name	text	"image_name" text	
image	blob	"image" blob	

Figura 1.1: Tabla de imágenes

urls			CREATE TABLE urls (url text)
url	text	"url" text	

whoIs			CREATE TABLE whoIs (search text, result text, number_search integer)
search	text	"search" text	
result	text	"result" text	
number_search	integer	"number_search" integer	

Figura 1.2: Tablas de urls y personas

El proyecto puede desplegarse en un contenedor propio, por lo que no interfiere para nada con el sistema anfitrión, para esto se ha utilizado la plataforma de código abierto [Docker](#), en el proyecto se ha incluido un *Dockerfile* con los comandos necesarios para la creación de la imagen y contenedor necesarios para su virtualización.

Dado que nuestro *chat* utiliza el servicio de interfaz de imagen, en este caso, de Linux, este admite el modo de trabajo cliente/servidor

Así que para que nuestro sistema Docker pueda utilizar la cámara de nuestro equipo debemos permitir a todos los usuarios acceder a la interfaz de pantalla, para esto instalaremos el paquete *x11-xserver-utils* en el que un cliente de X es un programa que interactúa con un servidor de X.¹

¹Para más información acceda a: <https://programmerclick.com/article/1860160122/>

1.5. DISTRIBUCIÓN DEL PROYECTO

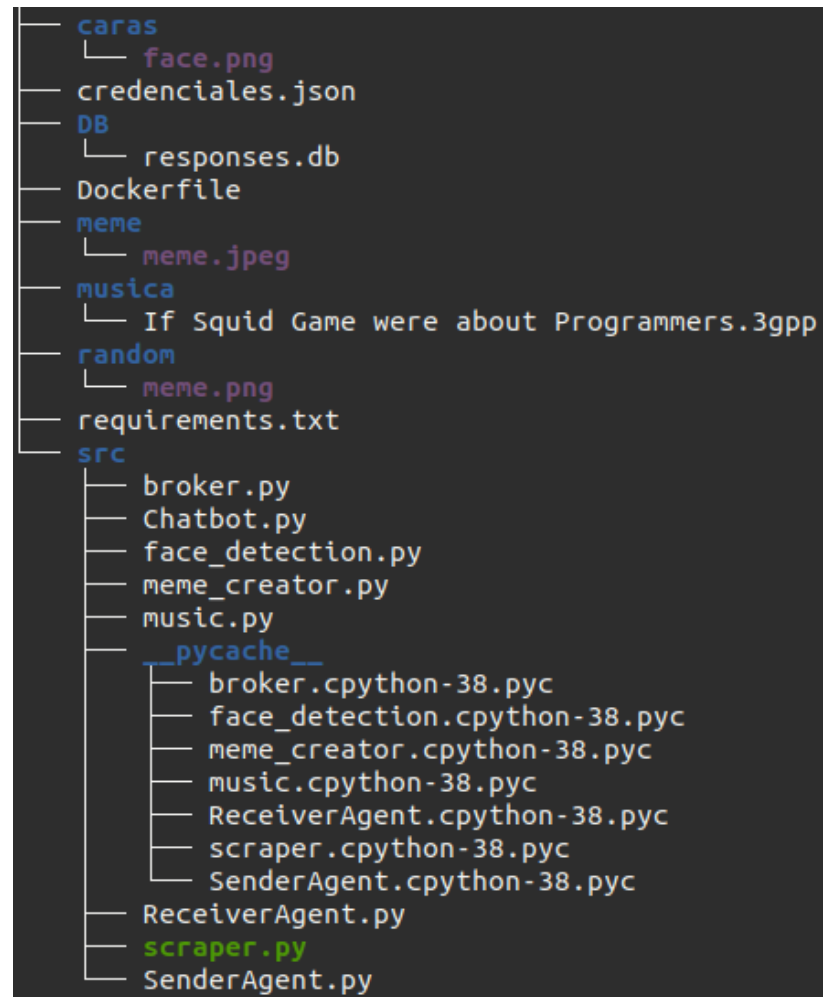


Figura 1.3: Distribución del proyecto

CAPÍTULO 2

Decisiones

2.1. SOBRE EL CHATBOT

Como hemos ido introduciendo a lo largo del documento, se han creado diferentes módulos para el correcto desarrollo de la práctica, por un lado tenemos el *main* (**Chatbot.py**), este archivo nos permite iniciar los dos agentes, usuario y *bot*.

Por otro lado se han añadido los diferentes *scripts* que componen la mayoría de funcionalidades del *bot*. Estos archivos se han decidido incluir aparte, para una más sencilla implementación. De esta forma tenemos un código más limpio sobre el que resulta mucho más fácil realizar cualquier cambio.

2.1.1. Usuario

Dentro del agente usuario o emisor se han tomado diferentes decisiones para su implementación. Como podemos observar el comportamiento *Init* envía un único mensaje al comportamiento *Request*, esto nos evita que el comportamiento *Request* este en ejecución continua sin ningún control. En este caso se ha implementado un tipo de mensaje (*menu*) de modo que cada vez que el *chatbot* envía un mensaje de tipo *inform* (esto es, informa sobre cualquier tema y acaba ese comportamiento), este envía un mensaje de tipo *menu*, lo que hace que el comportamiento *Request* solicite un nuevo comando.

Dentro del comportamiento *Request* se han incluido el comando *help*, esto se ha incluido dentro de este comportamiento debido a que si nuestro *chatbot* se encuentra desactivado nosotros podríamos seguir viendo los comandos con los que podamos interactuar.

Por otro lado se decidió crear un comportamiento *Responses*, el cuál filtra los mensajes que envía el *chatbot*. Este comportamiento informa sobre si el comando ha sido realizado con éxito o no (*inform* o *failure*) y manda información si así lo desea el *bot*.

2.1.2. Chatbot

Como tenemos diferentes funcionalidades, se ha decidido implementar un comportamiento por funcionalidad. En este caso 8 comportamientos diferentes componen las funcionalidades del agente receptor. Se han decidido mandar mensajes con la mayoría de información que el agente necesitaría para realizar cada comportamiento. Por ejemplo, en el caso de crear un fichero, el agente receptor enviará un mensaje al agente usuario para que este le envíe el nombre del archivo junto con su ruta y el contenido de este archivo. Esto se ha realizado de esta determinada manera debido a que el agente *chatbot* no debería comunicarse directamente con el agente usuario.

En todos estos comportamientos se han añadido diferentes mensajes en caso de que algún comportamiento falle, de esta manera si se encuentra un error se mandaría un mensaje de tipo *failure* con la información sobre el error.

2.2. SOBRE SQLITE Y DOCKER

Como ya se ha comentado con anterioridad se ha decidido utilizar la base de datos SQLite debido a que no requiere de un servidor separado y permite acceder a la base de datos usando una variación del lenguaje de consulta SQL.

Se ha decidido utilizar los contenedores Docker a las Máquinas Virtuales, gracias a su portabilidad y a su ligereza, Docker es mucho más rápido que una máquina virtual, y nos permite ahorrarnos espacio en el disco, que nunca viene mal. Por otro lado no se necesita una instalación del sistema operativo, ya que funciona mediante imágenes.

Protocolos de interacción

Se ha desarrollado un diagrama de protocolos por cada uno de los comportamientos que componen el agente receptor.

3.1. TIMEBEHAV

El protocolo más simple de todos, en el solo encontramos una única interacción por parte de las dos partes.

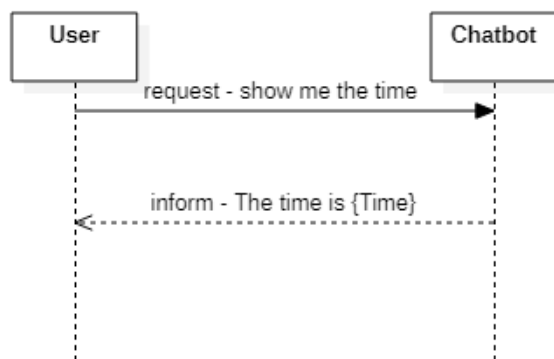


Figura 3.1: Protocolo Tiempo

3.2. WHOIS

Protocolo sobre buscar personas, se ha añadido la interacción con la base de datos ya que encontraba interesante introducir la relación con el *broker*

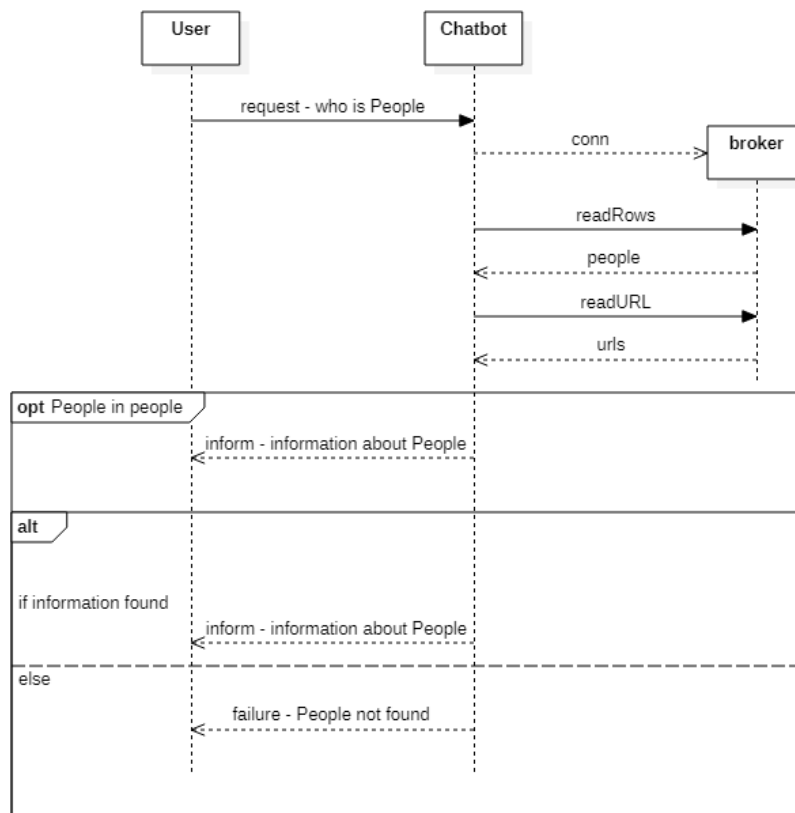


Figura 3.2: Protocolo Personas

3.3. CREATEFILE

Creación de fichero, el error podría surgir en cada una de las interacciones entre los dos.

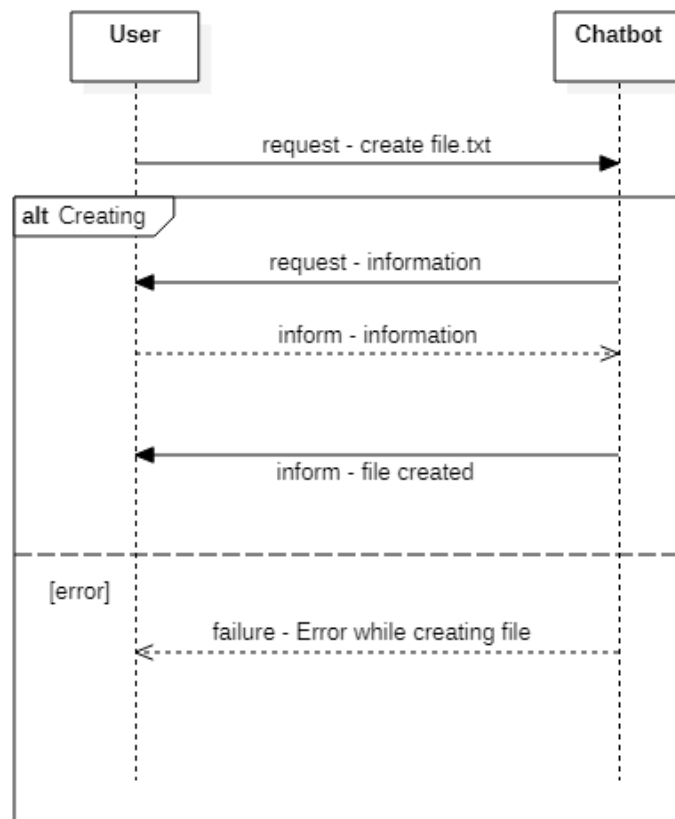


Figura 3.3: Protocolo Fichero

3.4. HISTORY

También protocolo muy simple con una única interacción por cada uno de los agentes e interacción con la base de datos.

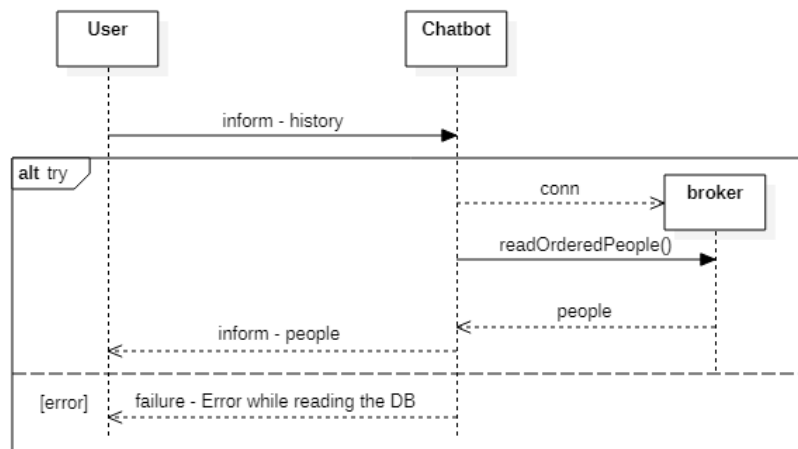


Figura 3.4: Protocolo Historia

3.5. DOWNLOAD

Uno de los comportamientos más complejos, se realizan múltiples interacciones entre los agentes así como interacciones con la base de datos.

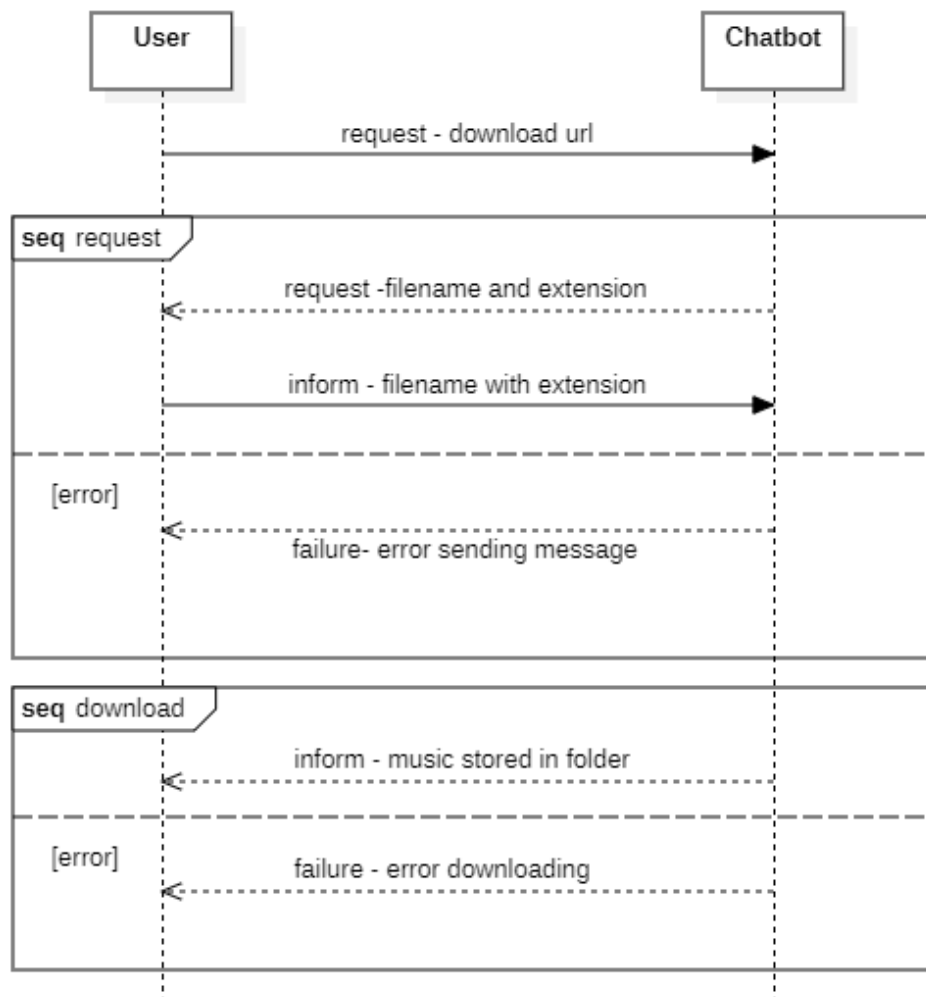


Figura 3.5: Protocolo Descargar

3.6. FACIAL

Protocolo que inicia el reconocimiento facial, comportamiento simple con pocas interacciones.

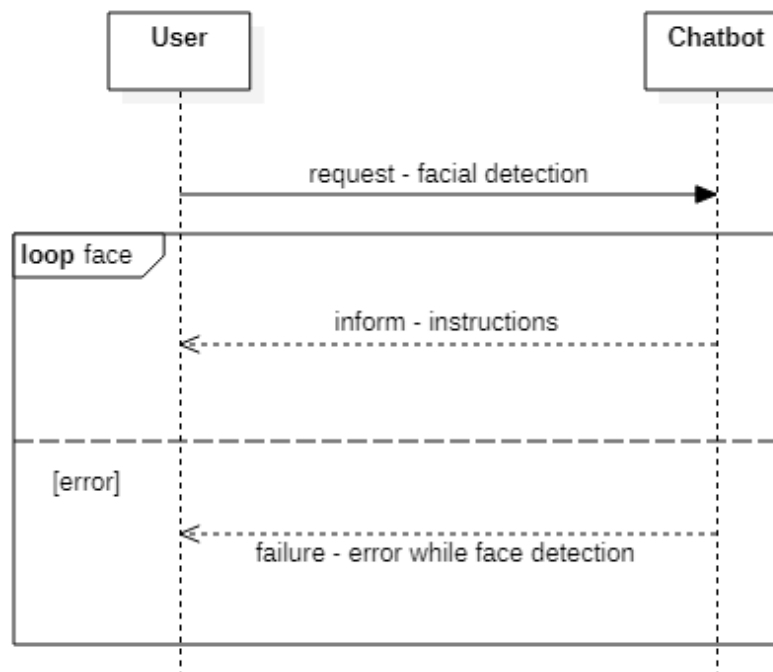


Figura 3.6: Protocolo Detección Facial

3. Protocolos de interacción

3.7. MEMEcreator

Protocolo para la creación de memes, uno de los mas complejos.

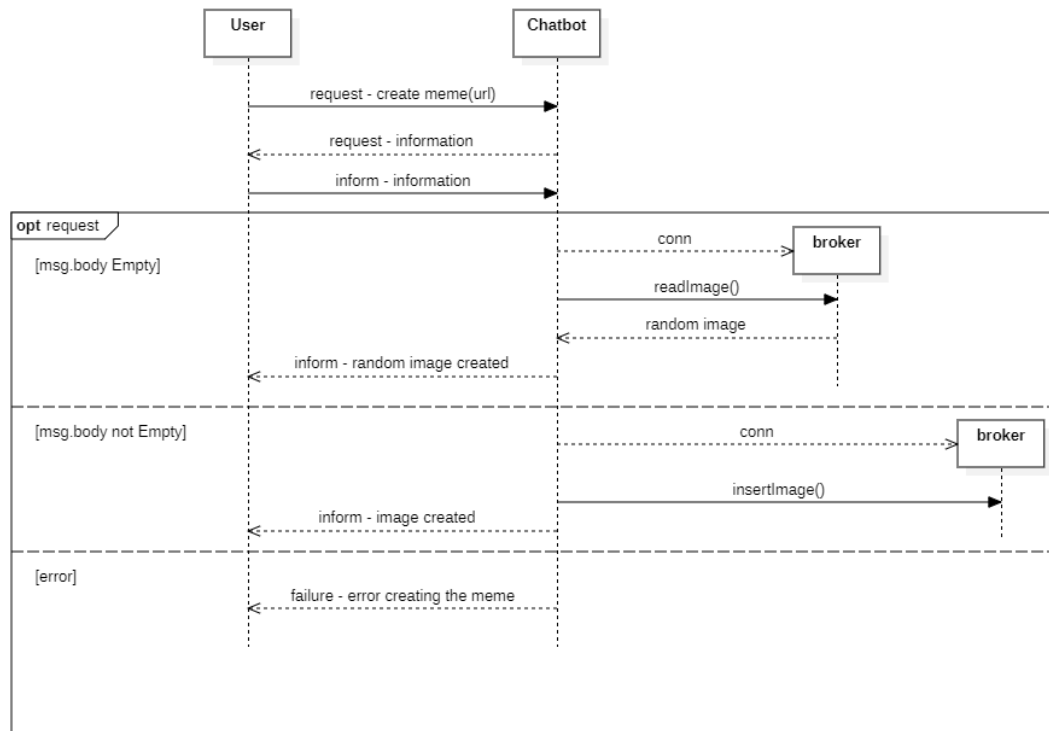


Figura 3.7: Protocolo Meme

3.8. EXIT

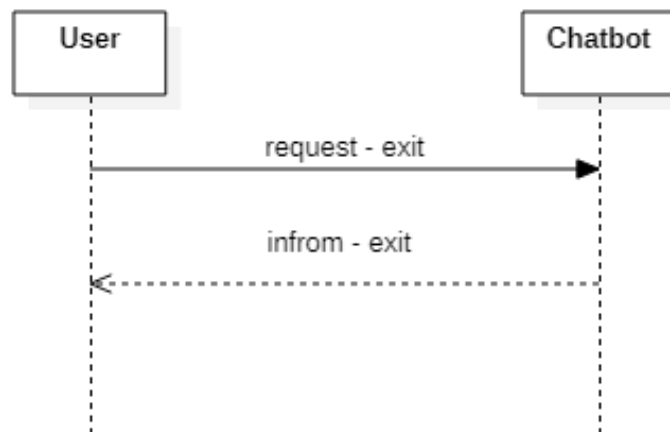


Figura 3.8: Protocolo Exit

Despliegue y Manual de Usuario

El proyecto se puede desplegar de varias maneras, en un contenedor Docker, cuya imagen esta introducida en los archivos del proyecto con el nombre de **Dockerfile** y de manera local.

4.1. PRERREQUISITOS EN LOCAL

Se han utilizado diferentes comandos e interfaces para el correcto desarrollo del *chatbot*, para configurar bien nuestra máquina se deben instalar las librerías de *python* que se encuentran incluidas en el archivo **requirements.txt**. Una vez instaladas las dependencias se necesita instalar el siguiente comando, este es llamado *ffmpeg*, es un conversor de archivos el cual utilizamos en el comportamiento *download*, nos permite cambiar la extensión de los archivos que nos descargamos de Youtube. A continuación se muestra la instalación de *ffmpeg* en **Linux**, si se quisiese instalar en **Windows** pulsa [aquí](#)

Listado 4.1: Instalación FFMPEG

```
1 | sudo apt install ffmpeg
```

Para comprobar que se ha instalado correctamente:

Listado 4.2: Comprobación FFMPEG

```
1 | ffmpeg --version
```

Es conveniente que para el correcto funcionamiento de la base de datos se instale algún programa o librería que permita la visualización interna de las tablas, por ejemplo, la extensión **SQLite** de Visual Studio Code realizada por el usuario *alexcvzz*. Esta extensión nos permite visualizar de manera dinámica los cambios realizados en la base de datos cuando se ejecutan los diferentes comportamientos.

4.2. DESPLIEGUE EN DOCKER

Como se comenta en el apartado de Docker y SQLite [1.4], necesitaremos habilitar la interfaz de pantalla y sonido¹, para esto utilizaremos los siguientes comandos:

Listado 4.3: Instalación X11

```
1 | sudo apt-get install x11-xserver-utils
```

Una vez instalado, permitimos a los usuarios acceder a las interfaces.

Listado 4.4: Host X11

```
1 | xhost +
```

¹Nótese que estos comandos son exclusivamente para entornos **Linux**

Una vez abiertos completado este paso se debe configurar el contenedor Docker con el proyecto, para esto crearemos una imagen Docker con el archivo *Dockerfile*, esto puede ser realizado con el siguiente comando. Una vez situado en la carpeta donde se encuentre el Dockerfile:

Listado 4.5: Crear imagen Docker

```
1 | docker build -t image_name .
```

A continuación se pueden mostrar las imágenes Docker que tenemos con el siguiente comando:

Listado 4.6: Crear imagen Docker

```
1 | docker images
```

Una vez introducido podemos asegurar la correcta creación de nuestro entorno. Lo siguiente que debemos hacer es crear el contenedor sobre el cual vamos a trabajar, para esto introduciremos el siguiente comando, donde damos acceso a nuestro servicio de vídeo y audio local.

Listado 4.7: Ejecución Docker

```
1 | sudo docker run -it -v /tmp/.X11-unix:/tmp/.X11-unix -e ↵  
    ↵ DISPLAY=unix$DISPLAY --device=/dev/video0 name bash
```

Una vez realizados todos estos pasos obtenemos una copia del entorno de ejecución local, con la ventaja de que nada se ha instalado en nuestra máquina propia, a partir de aquí, el funcionamiento de nuestro programa resulta ser el mismo que en local, con pequeñas peculiaridades que se irán exponiendo a su debido tiempo.

4.3. MANUAL DE USUARIO

Una vez en nuestro sistema ya sea Docker o local nos situaremos en la carpeta en la que se encuentran los archivos `.py` y ejecutamos nuestro archivo `main`, en este caso, ***Chatbot.py***. Una vez ejecutado nos encontraremos algo parecido a esto:

```
Creating Agents...
[Receiver Agent] spade_intro_2@404.city started
Loading responses...
[User Agent] spade_intro@404.city started
How can I help you?:
```

Figura 4.1: Pedir comandos

Como podemos ver nos indica los agentes que han sido creados a la vez que nos pregunta por un comando para interaccionar con el receptor. Vamos a explorar los diferentes comandos que nos encontramos

Antes de nada, mostremos como podemos interaccionar, para esto introduciremos la palabra *help*, cuya respuesta es la siguiente:

```
This is a list with all commands available:
- show me the time
- who is 'famous person'
- file
- download or youtube or youtube's urls
- history
- face detection
- meme creator
- exit
- help
```

Figura 4.2: Pedir comandos

Vayamos por orden, vamos a explorar los diferentes comandos y su modo de empleo, así como dos ejemplos de comando erróneos y un error mientras se ejecuta un comportamiento.

4.3.1. TimeBehav

Para este comportamiento podemos introducir *time* o *show me the time*, hecho esto, se enviará un mensaje al receptor, el cual enviará la hora actual al emisor, este nos mostrará la hora y volverá a llamar al comportamiento recogedor de comandos².

```
How can I help you?: time
-You say: time
.....
-Bot say: The current time is: 14/01/2022 20:04:13
How can I help you?:
```

Figura 4.3: Ejecutando TimeBehav

²Cuando finaliza la ejecución completa de un comando se vuelve a pedir otro en todos los comportamientos, por lo que solo se ha añadido en la primera imagen

4.3.2. Whols

Este es el único comportamiento que incluye información dentro de él, pueden pasar varias cosas cuando lo introducimos, lo primero que nos encontramos es que la persona que buscamos ya se encuentre en nuestra base de datos debido a búsquedas anteriores. Este comportamiento también actualizará el número de búsquedas de esta persona en la base de datos.

```
How can I help you?: who is Peter parker
-You say:  who is Peter parker
.....
Peter Parker
-Bot say: This person is already in our database: Spider-Man, traducido en ocasiones como Hombre Araña,[11][12] es un personaje creado por los estadounidenses Stan Lee y Steve Ditko,[13][14] e introducido en el cómic Amazing Fantasy n.º 15, publicado por Marvel Comics en agosto de 1962.[15] Se trata de un superhéroe que emplea sus habilidades sobrehumanas, reminiscentes de una araña, para combatir a otros supervillanos que persiguen fines siniestros.
```

Figura 4.4: Ejecutando Whols

Como podemos observar el *chatbot* nos informa de que la persona ya se encuentra dentro y nos indica la información que tiene almacenada (en Español). Por otro lado, podría pasar que fuese la primera vez que buscamos a esa persona, en cuyo caso el *chat* nos indica la información que encuentra una vez realizado el *web scrapping* y su posterior almacenaje en nuestra base de datos de esta información.

```
How can I help you?: who is penelope cruz
-You say:  who is penelope cruz
.....
-Bot say: Penelope Cruz: Penélope Cruz Sánchez (Alcobendas, Madrid; 28 de abril de 1974)[1], más conocida como Penélope Cruz, es una actriz española.
Person store in our database
```

Figura 4.5: Ejecutando WhoIs web

Por último, podría ocurrir que esta persona no fuese famosa y no encontrásemos información al respecto, esto también sería indicado.

```
How can I help you?: who is Paquita la pita
-You say:  who is Paquita la pita
.....
-Bot say: Fail, this person is not famous
```

Figura 4.6: Ejecutando WhoIs nadie

4.3.3. CreateFile

Para crear un fichero debemos introducir el comando *create file*, una vez introducido el *bot* nos mandará un mensaje indicando lo que debemos introducir, una vez introducido y mandado de nuevo, se creará el fichero en la ruta especificada.

Diferentes maneras de introducir información:

```
How can I help you?: create file
-You say: create file
.....
- Bot say: If you want to include a route type the hole path,
           if not, the file will be created in the current directory.

Example: file.txt | maria/University/Multiagent/file.txt.

- Bot say: you also have to include some text in the file (enter if not).
- User asking: Name+path: ../ejemplo.txt
- User asking: Text: Hola! Gracias por usar este chatbot!
-Bot say: I am creating the file...
-Bot say: file created in ../ejemplo.txt, with content: Hola!
```

Figura 4.7: Ejecutando CreateFile con contenido

```
How can I help you?: create file
-You say: create file
.....
- Bot say: If you want to include a route type the hole path,
           if not, the file will be created in the current directory.

Example: file.txt | maria/University/Multiagent/file.txt.

- Bot say: you also have to include some text in the file (enter if not).
- User asking: Name+path: ../ejemplo.txt
- User asking: Text: Hola! Gracias por usar este chatbot!
-Bot say: I am creating the file...
-Bot say: file created in ../ejemplo.txt, with content: Hola!
```

Figura 4.8: Ejecutando CreateFile sin contenido

4.3.4. Download

Podemos ejecutar este comportamiento con los comandos *download*, *youtube* o insertando una url de Youtube válida. Este archivo de descarga se introducirá en la carpeta *musica*. Una vez introducida la url se pedirá un nombre con una extensión para guardar el archivo.

```
How can I help you?: https://www.youtube.com/watch?v=YBYI7E2PqWE
-You say: https://www.youtube.com/watch?v=YBYI7E2PqWE
.....
-Bot say: give me a name with the extension for the file: video.mp3
```

Figura 4.9: Ejecutando Download

```
-Bot say: searching url...
-Bot say: Video store in music folder
```

Figura 4.10: Download ejecutado

4.3.5. History

Simplemente introducimos el comando *history* y el *chatbot* nos mandará información sobre la persona más buscada.

```
How can I help you?: history
-You say: history
.....
-Bot say: The most search person is: Peter Parker,
-Bot say: Result of this search: Spider-Man, traducido en ocasiones como Hombre
Araña,[11][12] es un personaje creado por los estadounidenses Stan Lee y Steve
Ditko,[13][14] e introducido en el cómic Amazing Fantasy n.º 15, publicado por M
arvel Comics en agosto de 1962.[15] Se trata de un superhéroe que emplea sus hab
ilidades sobrehumanas, reminiscentes de una araña, para combatir a otros supervi
llanos que persiguen fines siniestros.
-Bot say: Times this person has been search: 9
```

Figura 4.11: Ejecutando History

4.3.6. MemeCreator

Este comportamiento se ejecuta con el comando *meme creator*, una vez ejecutado el *chatbot* nos pide información sobre la imagen con la que crear el meme, si no incluimos ninguna imagen, el meme se creará con una imagen aleatoria y se almacenará en la carpeta *random* para que pueda ser usado.

```
How can I help you?: meme creator
-You say: meme creator
.....
-Bot say: Insert image for using that one or enter to get a random image from db
:
-Bot say: Meme created randomly and stored in random folder
```

Figura 4.12: Ejecutando Meme

En este caso el meme ejecutado aleatorio ha sido el siguiente:



Figura 4.13: Figura Meme

Si introducimos la ruta de una imagen podriamos obtener el siguiente resultado:

```
How can I help you?: meme creator
-You say:  meme creator
.....
-Bot say: Insert image for using that one or enter to get a random image from db
: /home/maria/Descargas/over.png
-Bot say: Meme created, stored in random folder and the image has been stored in
the database
```

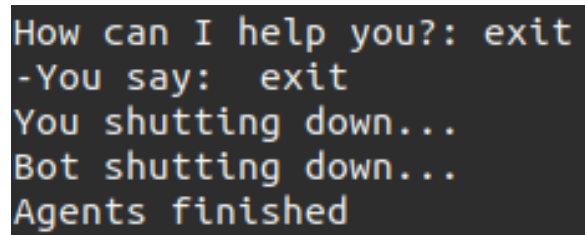
Figura 4.14: Ejecutando Meme con ruta



Figura 4.15: Figura Meme 2

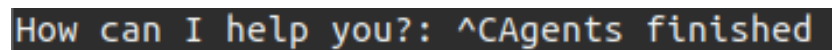
4.3.7. ShutDown

Para apagar los diferentes agentes es necesario pulsar *CTRL+C* o introducir el comando *exit*



```
How can I help you?: exit
-You say: exit
You shutting down...
Bot shutting down...
Agents finished
```

Figura 4.16: Ejecutando Exit



```
How can I help you?: ^CAgents finished
```

Figura 4.17: Ejecutando Exit CTRL+C

4.3.8. Vídeo explicativo

Se ha incluido un vídeo con una breve explicación sobre el funcionamiento de la práctica, para verlo pulse [aquí](#).

Bibliografía

- [1] DANIEL CIALDELLA. Foro sobre docker. URL: <https://dockertips.com/>, 2020. Último acceso: ene. 2022.
- [2] Programmer Click. Programador clic sobre docker. URL: <https://programmerclick.com/article/1860160122/>, 2020. Último acceso: ene. 2022.
- [3] Docker. Contenedores docker. URL: <https://hub.docker.com/>, 2020. Último acceso: ene. 2022.
- [4] Docker. Docker official page. URL: <https://www.docker.com/>, 2020. Último acceso: ene. 2022.
- [5] Docker. Docker para ubuntu. URL: <https://docs.docker.com/engine/install/ubuntu/>, 2020. Último acceso: ene. 2022.
- [6] D. Richard Hipp. Sqlite official page. URL: <https://www.sqlite.org/support.html>, 2020. Último acceso: ene. 2022.
- [7] Javi Palanca. Spade documentation official page. URL: <https://spade-mas.readthedocs.io/en/latest/readme.html>, 2020. Último acceso: ene. 2022.