



ESCUELA TÉCNICA SUPERIOR DE SISTEMAS INFORMÁTICOS
UNIVERSIDAD POLÍTÉCNICA DE MADRID

BFMB: Framework Base para Bots Modulares

TRABAJO FIN DE GRADO
GRADO EN INGENIERÍA DEL SOFTWARE

AUTOR: Ángel González Abad
TUTOR: Dr. Francisco Javier Gil Rubio

AGRADECIMIENTOS

Aquí estarán los agradecimientos cuando se me ocurra que poner.

RESUMEN

Los chatbots no son aplicaciones que haya surgido recientemente, ya estuvieron presentes durante años en la investigación y en las redes con el desarrollo de Internet y la web. Pero es ahora cuando existe un "boom" en ellos, sobre todo gracias a los servicios que conforman la Web 2.0 y los recientes asistentes virtuales, tales como Siri, Alexa o Google Now.

El problema que surge a la hora de desarrollar un bot conversacional o una máquina de estados es cuando el desarrollador tiene como requisito la necesidad de interactuar con varios medios de forma simultánea. Por ejemplo, un sistema que requiera una comunicación simultánea entre redes sociales o un sistema que mande órdenes a un conjunto de dispositivos IoT (Internet of Things), ya que cada servicio utilizará protocolos e interfaces diferentes. Esto aumenta la complejidad en el desarrollo y puede producir duplicidades si se quieren desarrollar varios bots con usos diferentes.

Ante esta problemática, mi proyecto se basará en un sistema base para desarrollar bots (u otro tipo de automatismos software) multiprotocolo. Dicho sistema se compone de un servidor de comunicaciones central al que podemos anexar diferentes conectores que interactúan con los servicios de terceros. Cada conector pertenece a un servicio concreto, donde podremos activar los que nos sean útiles. La parte lógica del bot se comunica con el servidor a través de JSON-RPC sobre HTTP, HTTPS, TLS sobre TCP o TCP (dependiendo de las necesidades del proyecto).

La finalidad de este proyecto es hacer que el desarrollador se centre en la lógica y en la inteligencia que pueda tener en mayor o menor nivel en lugar de tener que centrarse en las interfaces de los servicios de terceros.

SUMMARY

Chatbots aren't recent applications, they were present during multiple years as research projects and they were also present in the Internet during its growth. But now it's the moment where there's a "boom" in them, mainly caused by the Web 2.0 services and virtual assistants like Siri, Alexa or Google Now.

The problem begins when you want to develop a chatbot or a state machine where you need to interact with multiple sources simultaneously. For example, a system that requires a simultaneous communication between social networks or a system where you order some commands to a group of IoT (Internet of Things) devices, where each service will use different protocols and interfaces. These situations can increase the complexity in development process and can cause duplicates if you make multiple applications.

Due to this problem, my project will be based in a base system for multiprotocol chatbot development (or another software automatisms). This system will be integrated by a central communication server where you can attach different connectors and the developer will activate some of them for his requisites. The logic part of the bot will communicate with the server using the JSON-RPC protocol over HTTP, HTTPS, TLS or TCP (according to the requirements of the project).

The purpose of this project is making the developers concentrate in his bot logic and functionality instead of thinking in the connections between his bot and external services.

Índice

1.	INTRODUCCIÓN Y OBJETIVOS	1
2.	ESTADO DEL ARTE	3
2.1.	Historia de los chatbots	3
2.1.1.	Origen	3
2.1.2.	ELIZA	3
2.1.3.	ALICE	3
2.1.4.	Siri (Apple)	4
2.1.5.	Google Assistant	5
2.1.6.	Alexa (Amazon)	6
2.2.	Ejemplos de chatbots para usos posibles o reales.	6
2.2.1.	Seguridad	6
2.2.2.	Sanidad	7
2.2.3.	Banca	8
3.	ANÁLISIS	9
3.1.	Idea base del proyecto	9
3.2.	Tecnologías usadas	9
3.2.1.	JSON-RPC	9
3.3.	Servicios utilizados	10
3.3.1.	Tado°	10
3.3.2.	Telegram	13
4.	DISEÑO Y ARQUITECTURA DEL SISTEMA	15
4.1.	Metodología de trabajo	15
4.2.	Casos de uso	15
4.3.	Proceso (Diagrama de actividad)	16
4.4.	Secuencia	18
4.5.	Infraestructura	19
4.5.1.	Software bot	19
4.5.2.	Servidor de comunicaciones	20
4.6.	Estructura de base de datos	20
4.7.	Disposición del código	22
4.7.1.	bfmb-comcenter	22
4.7.2.	bfmb-base-connector	23
4.7.3.	bfmb-telegram-connector y bfmb-tado-connector	23
5.	DESARROLLO	25
5.1.	Lenguaje usado en desarrollo	25
5.1.1.	TypeScript	25
5.2.	Dependencias	27
5.2.1.	MongoDB	27
5.2.2.	NodeJS	28
5.2.3.	Jayson	29
5.2.4.	Api de Telegram	29
5.2.5.	Api de Tado°	32

6.	PRUEBAS	35
6.0.1.	Mocha	35
7.	MANUAL DE USUARIO	39
7.1.	Instalación	39
7.1.1.	Prerrequisitos	39
7.1.2.	Procedimiento de instalación	39
7.1.3.	Inicio del servidor	43
7.2.	Añadir usuarios al servidor de comunicaciones	44
7.3.	Cómo interactuar con el servidor de comunicaciones	45
7.3.1.	Realizar login en el servidor	46
7.3.2.	Obtener información del usuario de la api	46
7.3.3.	Comando de recepción de información	47
7.3.4.	Comando de envío de información	47
8.	CONCLUSIONES	49
9.	ASPECTOS ÉTICOS, SOCIALES, PROFESIONALES Y MEDIOAMBIEN- TALES DEL PROYECTO	51
10.	LÍNEAS FUTURAS	53

Índice de figuras

1.	Joseph Weizenbaum (2005). Foto de Ulrich Hansen	3
2.	Captura de la aplicación Siri en su versión más reciente (iOS 12)	4
3.	El dispositivo Google Home. Este incluye Google Assistant.	5
4.	Tecnologías que utiliza Negobot	6
5.	Niveles de evaluación de Negobot	7
6.	El chatbot Neo, desarrollado para la aplicación de gestión bancaria de Caixabank.	8
7.	Aparatos de gestión de calefacción de la empresa Tado°	12
8.	Diagrama de casos de uso	17
9.	Diagrama de actividad del envío y recepción de órdenes de las API	18
10.	Diagrama de secuencia del envío / recepción de mensajes	19
11.	Esquema de infraestructura (Diagrama de componentes)	21
12.	Estructura de base de datos	21
13.	Diagrama de clases	24
14.	Logo de MongoDB	27
15.	Logo de NodeJS	28
16.	Logo de Telegram	29
17.	Logo de Tado°	32
18.	Logo de Mocha	35
19.	Comunicación entre sistemas	46

1. INTRODUCCIÓN Y OBJETIVOS

Los chatbots no son aplicaciones que hayan surgido recientemente, sino que han tenido una larga historia por detrás. Desde el primer software chatbot (ELIZA en 1966), se han realizado desarrollos de chatbots hasta la actualidad. Pero es ahora cuando existe cierto surgir comercial de estos, gracias sobre todo a los asistentes virtuales de las grandes empresas tecnológicas como Siri (*Apple*), Alexa (*Amazon*), Watson (*IBM*) o Google Assistant (*Google*).

Pero existe una problemática para quienes quieran realizar un chatbot: la necesidad de una conexión con el exterior para poder comunicarse. No suele ser compleja esta parte si solamente va a interactuar con un único servicio, pero cuando se requiere la conexión a múltiples servicios e interactuar con ellos de forma simultánea, la complejidad del desarrollo aumenta, llegando a dedicar más recursos a la conexión de servicios que a la lógica del software.

Para ello nace la idea propuesta para este proyecto final de grado. Me centraré en el desarrollo de un sistema base por el cual nuestro nuevo bot se conectará a los servicios que requiera. No solo valdría para chatbots y su conexión a redes de chat o redes sociales, sino también para máquinas de estados conectadas a servicios IoT. Este sistema se centra en las comunicaciones para que el desarrollador solamente tenga que centrarse en desarrollar la lógica, el cual ya supone bastante trabajo.

Este proyecto cubre los siguientes objetivos:

- El desarrollo de un servidor de comunicaciones que hará de mediador entre el bot y los servicios externos en Internet, usando para ello unos módulos denominados conectores.
- La creación de uno o dos de esos módulos conectores para interactuar con los servicios de terceros.
- La creación de un bot sencillo para poder realizar las demostraciones de funcionamiento.

2. ESTADO DEL ARTE

2.1. Historia de los chatbots

2.1.1. Origen

El origen del nombre "chatbot" viene de un software llamado CHATTERBOT, el cual era un jugador virtual del videojuego de mazmorras TinyMUD. La principal tarea de este bot era responder a las preguntas de los usuarios que tenían relación con la navegación por la mazmorra u objetos del juego. El mismo simulaba habilidad conversacional mediante reglas, mediante las cuales logró "engaños" a los usuarios y que estos creyeran que era un jugador humano más. [1, pág. 2]

2.1.2. ELIZA

ELIZA fue un programa de procesamiento del lenguaje natural creado entre los años 1964 y 1966 por Joseph Weizenbaum (1923-2008) en el Laboratorio de Inteligencia Artificial del MIT (Instituto Tecnológico de Massachusetts). El objetivo de este software era demostrar la superficialidad de la comunicación entre humanos y máquinas.

El software que realizó fue una de las primeras vías de interacción entre persona y máquina mediante el uso del lenguaje natural. El mismo era una parodia de un terapéuta que ejercía psicoterapia centrada en el cliente, una teoría psicológica creada por el psicólogo norteamericano Carl Rogers. El software reutilizaba con frecuencia las frases enviadas por el cliente y las convertía en preguntas para el mismo.

ELIZA era un *software* limitado, ya que solo fue programado para responder a ciertas palabras o frases clave. Así que lo normal era llegar a conversaciones sin sentido.

2.1.3. ALICE

ALICE (Artificial Linguistic Internet Computer Entity) es un bot con inspiraciones en ELIZA creado en 1995 por Richard Wallace. Puede procesar lenguaje natural mediante el uso de patrones. Dichos patrones se formaban con un lenguaje de marcado basado en XML llamado AIML. Actualmente, el lenguaje AIML sigue estando en desarrollo y la última versión disponible es la especificación 2.1, publicada en Marzo de 2018.

Es un lenguaje de marcado que ha tenido cierto éxito. Mitsuku, un chatbot que utiliza AIML como base y desarrollado por Steve Worswick, ha logrado ganar el Premio Loebner en cuatro ocasiones (en los años 2013, 2016, 2017 y 2018)

A continuación os muestro un ejemplo de un patrón AIML:



Fig. 1: Joseph Weizenbaum (2005). Foto de Ulrich Hansen



Fig. 2: Captura de la aplicación Siri en su versión más reciente (iOS 12).

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <aiml version="2.0">
3   <category>
4     <pattern>HI *</pattern>
5     <template>Hello world!</template>
6   </category>
7
8   <category>
9     <pattern>What is a chatbot</pattern>
10    <template>
11      A chatbot is a computer program designed to respond
12      to text or voice inputs in natural language.
13    </template>
14  </category>
15 </aiml>
```

2.1.4. Siri (Apple)

Siri es un asistente virtual desarrollado originalmente a principios de 2010 por una start-up del mismo nombre, la cual ofrecía dicha aplicación. Posteriormente fue adquirida por Apple para incorporarse como funcionalidad del sistema operativo iOS en Octubre de 2011 y se incorporó en el sistema MacOS Sierra, versión liberada en el año 2016.



Fig. 3: El dispositivo Google Home. Este incluye Google Assistant.

Es capaz de reconocer el lenguaje natural mediante el uso de un conjunto de servidores remotos, es decir, no es el dispositivo en sí quien procesa la voz. Aparte, en base a ese procesamiento de voz, también procesa la orden para ser realizada por el dispositivo. Pueden ver una captura de la aplicación en su versión más reciente para iOS en la figura 2.

Además de reconocimiento del lenguaje natural mediante la voz, es capaz de expresar sus resultados con voz, lo que la otorga un valor añadido.

Por último, los desarrolladores de aplicaciones para iOS pueden añadir órdenes concretas para que puedan ser realizadas mediante la comunicación con Siri, usando la biblioteca SiriKit. SiriKit está disponible desde la versión 10 de iOS, la versión de macOS 10.12 y la versión 3.2 de watchOS (el sistema operativo que contienen los Apple Watch).

2.1.5. Google Assistant

Google Assistant es el asistente virtual desarrollado por Google, el cual es el núcleo de los productos Google Home (uno de ellos lo puede ver en la figura 3) pero también se encuentra presente en los dispositivos Android que incorporan las aplicaciones de Google (ya que hay dispositivos Android que no incluyen dichas aplicaciones).

Al igual que Siri, dispone de un sistema de reconocimiento de lenguaje natural por voz y texto con el genera resultados en base a los términos dados usando sus sistemas en la nube.

Al igual que Siri, también deja abierta la puerta a los desarrolladores para añadir



Fig. 4: Tecnologías que utiliza Negobot

funcionalidad adicional mediante la creación de acciones. Para ello, se crea un fichero llamado actions.xml y a partir del mismo se crean todas las acciones que se planteen. Para el caso de Google Home, hay que crear una aplicación servicio en la nube.

2.1.6. Alexa (Amazon)

Alexa es el software que incorpora los dispositivos Echo de la misma empresa y puede integrarse en otros dispositivos como smartphones, ordenadores y otros dispositivos de terceros. Tiene las mismas funcionalidades que Google Assistant y Siri: permite la ejecución de órdenes o iniciar conversaciones en lenguaje natural y recibir respuestas al mismo nivel.

Para desarrollar nuevas órdenes en esta plataforma, se hace uso de un sistema denominado (Alexa Skill Kit) pero, para poder usar este sistema, se requiere que dicha "skill" esté en un servicio en red (preferiblemente en la nube de Amazon). Sería similar a la forma que se realiza en los servicios para Google Home.

2.2. Ejemplos de chatbots para usos posibles o reales.

2.2.1. Seguridad

Como ejemplo de uso en el campo de la seguridad tenemos el chatbot Negobot. Desarrollado por la Universidad de Deusto en el año 2012, Negobot es un chatbot que simula ser un menor en los chats públicos de Internet para la detección de conductas pedófilas.

Para lograr este objetivo, este chatbot utiliza el procesamiento de lenguaje natural, la teoría de juegos y funciones de evaluación para determinar si el interlocutor tiene un comportamiento que se pueda asociar con un posible pedófilo en base a la conversación presente. El sistema tiene siete niveles en los que se determina dicho comportamiento, empezando desde el nivel -3 (no hay comportamiento pedófilo) a +3 (sí lo tiene) comenzando por el nivel 0. (Ver figuras 4 y 5)

Otros de los objetivos que tiene este chatbot es poder superar el test de Turing (tener capacidad que el otro interlocutor no sepa si es una máquina o una persona), el cual es

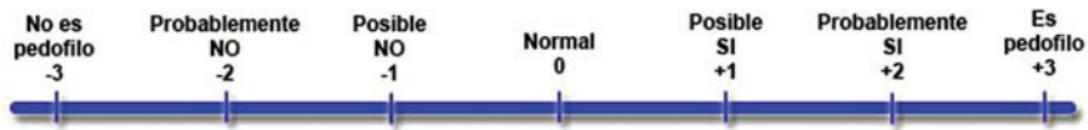


Fig. 5: Niveles de evaluación de Negobot

importante para la finalidad del mismo.

Por último, hay que decir que esto no podría aplicarse para perseguir a posibles agresores, porque no puedes juzgar a nadie por decirle cosas impropias a un bot, aunque el bot se comporte como un niño. Aparte, se estaría investigando a esa persona en base a que pueda ser capaz a cometer un delito contra alguien real. Bajo ese pretexto se estaría vulnerando la presunción de inocencia y solo produciría mayor perjuicio que beneficio.

2.2.2. Sanidad

Existen la presencia de múltiples chatbots orientados al ámbito sanitario: Your.MD, Ada... pero me voy a enfocar en el chatbot que provee Babylon Health (no confundir con el traductor Babylon). Usa procesamiento de lenguaje natural y preguntas junto con la Inteligencia Artificial para comprobar la gravedad de los síntomas del usuario. El éxito que tiene el servicio que provee es tan alto que el Servicio Nacional de Salud de Reino Unido (UK's National Health Service - NHS) tiene un acuerdo con esta empresa para proveer este servicio a los usuarios del servicio público.

El NHS está usando este servicio para poder gestionar aquellos problemas de salud que no requieran urgencia o especialización (que bastarían con tomar un paracetamol y descansar, por ejemplo) y ayudar a reducir la saturación de su servicio público sanitario y evitar el colapso en ciertos momentos del año (temporada de gripe, por ejemplo). También está pensado este servicio para desarrollar asistentes a la hora de consultar necesidades en materia de salud sin tener que pedir cita con el médico.

Aun así, esta innovación no está exenta de crítica y duda. Hay temas sin resolver con estos servicios en materia legal. Cuando un médico comete una negligencia, él es el responsable directo pero, ¿qué sucede con un bot? En teoría, el responsable legal podría ser la empresa que ha usado los servicios del bot para sus clientes, pero también podría trasladar esa responsabilidad a la empresa que ha desarrollado el bot. ¿A quién le exiges la responsabilidad?

Otro punto de crítica puede venir de la falta de enfoque en base a la localización del paciente. Por ejemplo, el sistema no sería válido si no plantea hacer preguntas y evaluaciones por un posible caso de Malaria porque en el país donde se ha desarrollado el software no va a haber esa posibilidad (por cuestión de clima) pero que sí es vital hacer esas preguntas en los países del sur del Sahara (zona donde se producen mayor número de casos).

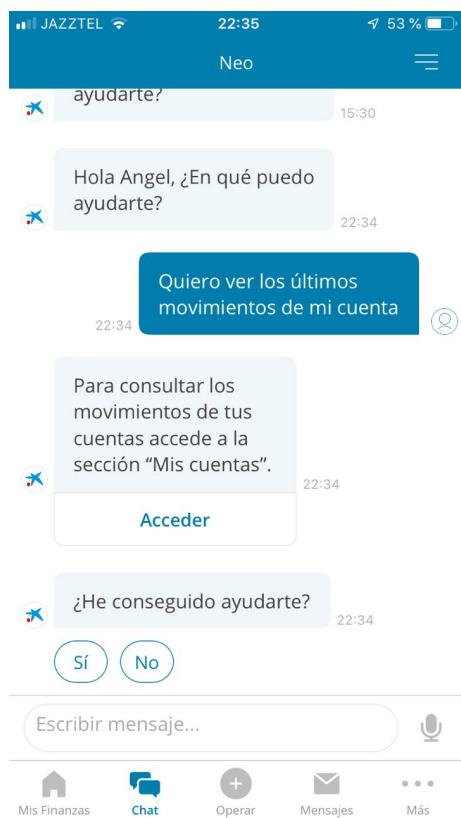


Fig. 6: El chatbot Neo, desarrollado para la aplicación de gestión bancaria de Caixabank.

2.2.3. Banca

Se están usando los chatbots en el ámbito de la banca para ser el sustituto de la sección de ayuda en las aplicaciones de gestión para el usuario, concretamente, de las secciones de *Preguntas frecuentes* (o FAQ). Por ejemplo, está el caso de Neo, un chatbot que está incorporado en los servicios de *Línea Abierta* (ahora *CaixaBankNow*), del grupo CaixaBank.

El bot puede entender el lenguaje natural y responder a dichas preguntas. También incorpora un mecanismo de retroalimentación para mejorar el bot ante respuestas no válidas.

Este bot puede comunicarse con el usuario a través de la web de *Línea Abierta*, la aplicación móvil de CaixaBank, Google Home y en Amazon Echo.

3. ANÁLISIS

3.1. Idea base del proyecto

La idea principal es el desarrollo de un sistema de comunicaciones con diversas APIs externas para facilitar el desarrollo de bots o automatismos que interactúen con ellos. En sí no forma una aplicación, sino que será una base para que otras personas puedan realizar sus aplicaciones. Por ello, es un framework.

Un framework es un software compuesto de componentes personalizables e intercambiables para el desarrollo de una aplicación. Podría considerarse como una aplicación genérica incompleta en donde se añadan las últimas piezas. [11, pág. 1]

3.2. Tecnologías usadas

3.2.1. JSON-RPC

JSON-RPC es un protocolo de llamada a procedimiento remoto (Remote Procedure Call) cliente-servidor cuyos mensajes se componen de datos codificados en formato JSON (Javascript Object Notation). Es un protocolo agnóstico, puede realizar comunicaciones cliente-servidor a través de HTTP, HTTPS o TCP, entre otros. La especificación más reciente es la versión 2.0, publicada en 2010.

Como el protocolo usa JSON como mensaje, dispone de las mismas características que un fichero JSON: dispone de los cuatro tipos primitivos (String, Number, Boolean y Null) y de dos tipos de estructura (Object y Array). El protocolo en sí se basa en dos modelos de datos: petición (Request) y respuesta (Response), los cuales explico a continuación:

- **Objeto petición (Request):** La llamada a procedimiento remoto en JSON-RPC se basa en enviar este objeto a un servidor, el cual dispone de los siguientes atributos:
 - **jsonrpc:** Atributo que especifica la versión del protocolo JSON-RPC. Debe indicar el número de versión del protocolo. Es decir, si es la versión 2.0, debe ponerse "2.0".
 - **method:** Un String donde se indica el método a invocar. Cualquier nombre de método es válido excepto los que empiecen con rpc.", ya que son de uso interno.
 - **params:** Un atributo de tipo estructura que indica los parámetros a enviar al método. Es un atributo que puede ser opcional.
 - **id:** Un identificador establecido por el cliente que puede ser un tipo distinto de Boolean (y Null, en base a las actualizaciones de la especificación). Si este valor no está incluido, el protocolo considera la petición como una notificación.

- **Objeto respuesta (Response):** La llamada a procedimiento remoto en JSON-RPC debe devolver una respuesta a cada petición excepto si es una notificación. El objeto respuesta tiene los siguientes atributos:
 - **jsonrpc:** Atributo que especifica la versión del protocolo JSON-RPC. Debe indicar el número de versión del protocolo. Es decir, si es la versión 2.0, debe ponerse "2.0".
 - **result:** Este atributo es obligatorio si el resultado es exitoso y, en caso contrario (error), no debe estar presente. Siempre va a ser un atributo de estructura.
 - **error:** Este atributo debe existir en los casos de error y no aparecer en casos de éxito. Siempre va a ser un atributo de estructura.
 - **id:** El identificador en las respuestas es obligatorio y debe ser el mismo id que el recibido en el objeto de la petición. En caso de no llegar un id correcto, el valor de este atributo debe ser Null.
- **Objeto error:** Si una llamada a procedimiento remoto encuentra un error, el objeto respuesta debe incorporar este objeto de error en el atributo denominado como tal. El objeto error tiene los siguientes atributos:
 - **code:** Código de error de JSON-RPC
 - **message:** Un String que indica la descripción resumida del error.
 - **data:** Atributo de tipo primitivo o de estructura que entrega información adicional del error. Es un atributo opcional.

3.3. Servicios utilizados

3.3.1. Tado°

Tado° GmbH es una empresa tecnológica alemana con sede en Múnich y es un fabricante de sistemas de automatización de calefacción y refrigeración para el hogar conectados a Internet.

Fundado en el año 2011 por Johannes Schwarz, Christian Deilmann y Valentin Sawadski, el nombre de dicha compañía está inspirado en una unión de las expresiones japonesas "tadaima" (ただいま), que significa "Estoy en casa" y "okaeri" (おかえり), que significa "Bienvenido" a la hora de dirigirse a alguien con quien convives. Puede parecer que este detalle no aporta valor en este documento pero es lo contrario, ya que uno de los valores añadidos de sus sistemas de automatización es, precisamente, la gestión de la climatización basándose en la geolocalización.

Su gestión basada en geolocalización se realiza de la siguiente forma: El usuario o usuarios de la vivienda tienen asociado un dispositivo móvil, el cual debe tener la app de la compañía instalada y configurada. Esta app recoge la posición actual y se compara con

un *geofence*¹ definido por el usuario maestro de la vivienda. Cuando todos los usuarios de la vivienda abandonan la zona definida por el *geofence*, Tado ordena la desconexión de la calefacción o la reducción de la temperatura de las estancias.

Otra de sus funcionalidades es la detección de ventanas abiertas. Los sensores instalados (termostatos y válvulas) detectan la temperatura y en la humedad, y los cambios repentinos en los mismos se asocian con la apertura de ventanas. Ante ello, Tado desconecta la calefacción por un tiempo definido en la configuración (entre 5 y 60 minutos) para evitar el consumo innecesario de energía. Esta funcionalidad es desactivable si no se precisa de ella (la estancia no tiene ventanas, por ejemplo).

La última funcionalidad adicional que provee es el ajuste de potencia en la calefacción basándose en el tiempo atmosférico actual. La calefacción se ajusta en base a la estimación de incidencia solar que puede haber.

Dispositivos Tado^o, en sus sistemas de calefacción, tiene distintos dispositivos de gestión, los cuales explico a continuación:

- **Puente de conexión (Bridge):** Es el dispositivo de conexión entre Internet (la api de Tado) y la red 6LoWPAN² en la que se comunican los dispositivos entre sí. Funciona con una conexión Ethernet y una conexión USB para alimentación eléctrica.
- **Termostato:** Dispositivo que tiene la capacidad de encender o apagar la caldera o ajustar su potencia (según compatibilidad de calderas). Para ello, dispone de un relé para la conmutación. Incluye sensores para la medida de temperatura y humedad. Funcionan con tres pilas AAA y pueden ser instalados también como controladores de temperatura inalámbricos.
- **Válvula termostática:** Mecanismo eléctrico que incluye sensores para la medición de temperatura y humedad. Con ellos, activa un motor que regula la apertura de la válvula del radiador donde se encuentre instalado y también pueden regularse de forma manual. Funcionan con dos pilas AA.

Pueden ver las fotografías de cada dispositivo en la figura 7.

También Tado^o tiene un dispositivo para gestionar de forma remota aparatos de aire acondicionado. Aun teniendo una forma diferente de trabajar a nivel técnico, dispone de las mismas funcionalidades que su equivalente en calefacción.

¹ El término *geofence* es un perímetro virtual asociado a un área geográfica. Su utilidad se basa en la ejecución de eventos en software ante un suceso de entrada o salida del área definida.

² 6LoWPAN (IPv6 over Low-Power Wireless Personal Area Networks) es un estándar que adapta la transmisión de paquetes IPv6 a redes que siguen el estándar IEEE 802.15.4, los cuales son dispositivos con recursos muy limitados y un flujo de datos muy bajo. Otra característica de esta red es la capacidad de crear redes en malla.



(c) Válvula termostática Tado°

Fig. 7: Aparatos de gestión de calefacción de la empresa Tado°

No entro en detalle en ese modelo de termostato porque queda fuera del proyecto (no dispongo de ese aparato y, por ello, no puedo probar su funcionamiento para adaptarlo al software que presento).

3.3.2. Telegram

Telegram es un servicio de mensajería instantánea y VoIP³ creado en el año 2013 por Nikolai y Pavel Durov, quienes anteriormente crearon la red social VK, conocida sobre todo en la Federación de Rusia. Se estima que el servicio tiene 200 millones de usuarios activos al mes.

Las funcionalidades que dispone el servicio son las siguientes:

- **Acceso multiplataforma:** Telegram, a diferencia de otras redes, permite que un usuario esté activo en varios dispositivos a la vez, sin obligar a cerrar la sesión del resto de conexiones ni borrar el contenido de los chats almacenados. Lo que sí requiere es registrar la cuenta de Telegram con un número de teléfono móvil.
- **Capacidad para adjuntos de gran tamaño:** Gracias al protocolo MTProto, creado por Nikolai Durov, es posible transferir adjuntos multimedia de hasta 1,5GB de tamaño.
- **Grupos y canales:** Para conversaciones entre varias partes, existe la creación de grupos. En ellos pueden haber hasta 200 000 usuarios y se pueden realizar respuestas a un mensaje concreto del chat, mencionar a un usuario y generar *hashtags*⁴. Dichos grupos pueden ser públicos o privados y disponer de varios administradores.

Por otra parte, los canales están pensados para difusión de mensajes por parte de uno o varios administradores. No existe límite de usuarios en los canales, por lo que cualquiera puede acceder. Al igual que los grupos, los canales pueden ser públicos o privados y tienen las mismas capacidades (pueden enviar adjuntos y mensajes, aunque el resto de usuarios no puedan hacer uso de ello).

- **Bots:** En Telegram es posible el uso de bots y se incentiva el desarrollo de nuevos bots gracias a la documentación que se provee a los desarrolladores. Los bots pueden ser configurados de distintas formas (escuchando comandos, escuchando menciones o cualquier mensaje). También existen los *inline bots*, son bots que no requieren estar presentes en un chat para funcionar y proveen datos en base una consulta.
- **Stickers:** Una extensión de los emojis para poder expresar reacciones de forma breve y visual. Los stickers ya eran una característica que ya tenía la aplicación de mensajería LINE, creada en 2011.

³ VoIP: Voz por protocolo de Internet. Permite la transmisión de la voz a través de Internet usando el protocolo IP. Se usa actualmente por los proveedores de telefonía en lugar de la telefonía analógica.

⁴ Cadena de caracteres que se usa para clasificar la información presente. Sería un tipo de metadato.

- **Chats secretos:** Telegram provee la opción de usar chats secretos entre dos usuarios. Para ello, se utiliza cifrado extremo a extremo⁵ Por defecto, Telegram no activa este tipo de chats, ya que ciertas funcionalidades de Telegram no funcionan con este tipo de chats, como el acceso multiplataforma o los bots.
- **VoIP:** Al igual que otros servicios de mensajería actuales, dispone de servicio de llamadas por voz.
- **Inicio de sesión en sitios terceros:** Telegram también tiene implementado un sistema para poder iniciar sesión con las credenciales del servicio en páginas web tercera, siempre que estas incluyan los componentes necesarios para ello.

⁵ El cifrado extremo a extremo (End-to-end encryption) es un sistema de comunicación donde dos usuarios pueden comunicarse sin que sus mensajes sean captados por los nodos intermedios en una red.

4. DISEÑO Y ARQUITECTURA DEL SISTEMA

4.1. Metodología de trabajo

La metodología de trabajo que he usado para este proyecto sigue el modelo de desarrollo ágil, ya que se ha realizado un desarrollo iterativo e incremental. Únicamente se ha usado Kanban para el desarrollo, ya que al ser un único desarrollador, no le veía mucho sentido realizar metodología SCRUM, un método pensado para el desarrollo en equipo.

Entrando en detalle, el método Kanban es un método con origen en Japón, a partir de las metodologías de fabricación en Toyota. El tablero se compone de un mínimo de tres columnas ("Por hacer", "En proceso" y "Hecho") y consta de cuatro principios fundamentales:

1. **Calidad garantizada:** Todo lo que se desarrolle debe salir bien a la primera. Se debe priorizar la calidad final de la tarea en lugar de la rapidez.
2. **Reducción del desperdicio:** Hacer lo justo y necesario pero hacerlo bien.
3. **Mejora continua:** Permite estar abierto a pequeñas mejoras de forma continua.
4. **Flexibilidad:** Se realiza la entrada de nuevas tareas en base al backlog y ajustándose a las necesidades del momento.

4.2. Casos de uso

Los casos de uso para este proyecto están definidos a través de dos actores: el software que va a realizar las órdenes (el bot) y el usuario.

El bot realizará los siguientes casos de uso:

- El bot solicitará un autenticación con el servidor de comunicaciones (uno de los componentes del framework), el cual se envía un usuario y una contraseña al mismo. El servidor realiza la autenticación, comprobando si el usuario existe y si el hash de la contraseña guardada coincide con el hash generado de la contraseña enviada. Si la autenticación tiene éxito, se genera un token con los datos necesarios del usuario para poder realizar las acciones pertinentes.
- El bot puede solicitar mensajes. Esta acción es una abstracción de las órdenes GET de una API HTTP REST. Para ello, antes de realizar acción alguna, se solicita el token que se recibió al autenticar para así validar la acción. Tras la verificación exitosa, se solicita los datos requeridos a través del servidor de comunicaciones. El servidor de comunicaciones dispone de unos componentes que denominaremos como "conectores", los cuales transforman la solicitud en los datos necesarios para el endpoint elegido. La respuesta o el error de la solicitud a la api la devuelve el servidor de comunicaciones.

- El bot puede enviar mensajes. Esta acción es una abstracción de las órdenes POST, PUT y DELETE de una API HTTP REST. Al igual que con el caso de uso ya definido anteriormente, se verifica el token, se transforma el contenido y se envía a la api. La respuesta o error se devuelve a través del servidor de comunicaciones.

Por otra parte, el usuario realizará los siguientes casos de uso:

- A través de las aplicaciones que disponga cada servicio, el usuario interactúa con dichos servicios. Por ejemplo, puedo interactuar con el bot a través de un chat (Telegram) o cambiar la configuración de la calefacción (Tado°).

Pueden ver el diagrama de casos de uso en la figura 8.

4.3. Proceso (Diagrama de actividad)

En esta sección se va a definir los procesos que disponemos en el framework: el envío de mensajes a través del servidor de comunicaciones y la recepción de los mismos. El diagrama de actividad es el mismo para ambos casos. Se ejecuta el mismo procedimiento.

Viendo la figura 9, tenemos este procedimiento:

1. **Se recibe el mensaje JSON-RPC** que habrá enviado nuestro chatbot, el cual estará conectado a nuestro servidor de comunicaciones.
2. Se comprueba si dicho **mensaje contiene los parámetros universales** del servidor de comunicaciones, los cuales son:
 - **Token:** Cuando el usuario bot ha iniciado sesión en el servidor de comunicaciones, recibe un token. Dicho token tiene una información anexa (payload) donde se indica las redes que tiene configuradas.
 - **Network:** Este parámetro indica el nombre del servicio en red donde se debe realizar la orden. Por ejemplo, si quisiéramos ejecutar un endpoint de Telegram, este atributo tendrá "Telegram" como valor.
 - **Options:** En este parámetro se almacenan los datos necesarios para el conector correspondiente (endpoint a conectar, parámetros requeridos de ese endpoint...).
3. Se comprueba que el **token siga siendo válido** en tiempo y forma.
4. Se comprueba que el **usuario bot tenga la red a usar configurada**. Es decir, que disponga del token o las credenciales necesarias para poder establecer el contacto.
5. Si todo sale bien, **se buscará el conector activo correspondiente** (si lo está). Un conector activo se determina por la configuración del servidor, donde se activan los conectores que uno quiera ejecutar en ese servidor.
6. Tras esto, **se ejecuta la solicitud al endpoint del servicio elegido**. Después, si no ha habido error, recibiremos la respuesta del endpoint que hemos ejecutado.

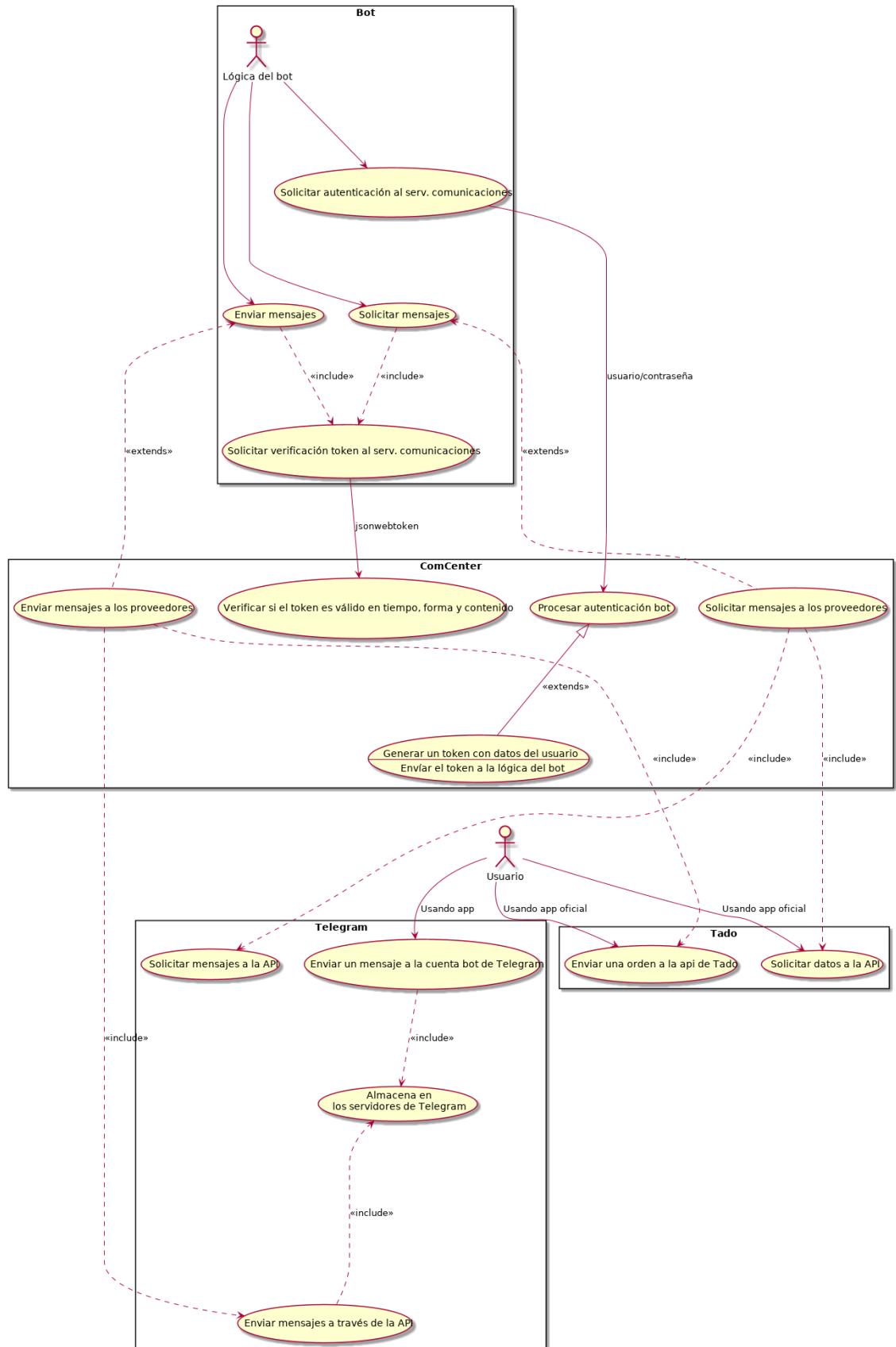


Fig. 8: Diagrama de casos de uso

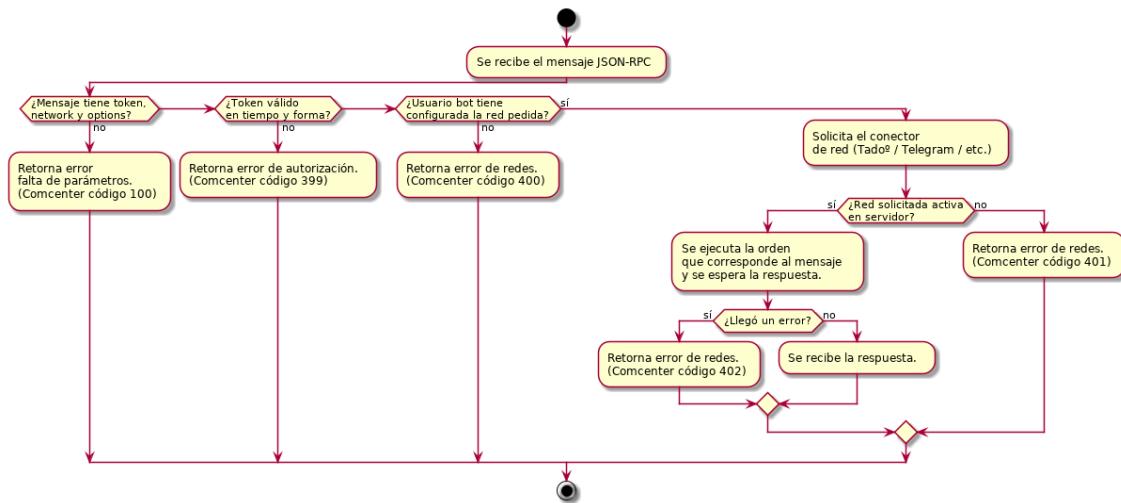


Fig. 9: Diagrama de actividad del envío y recepción de órdenes de las API

7. **Ante cualquier caso que no se cumpla o en caso de error, el sistema responderá con un error.** El código de error está definido para el servidor de comunicaciones. Puede ver la lista de códigos de error en los anexos.

4.4. Secuencia

El diagrama que se muestra en la figura 10 representa la secuencia que se ejecuta cuando el servidor de comunicaciones (ComCenter) recibe un mensaje por parte del software bot conectado a este.

Todo comienza con el envío del mensaje JSON-RPC al servidor de comunicaciones, donde es recibido por el servidor JSON-RPC incorporado. Después, dicho mensaje ejecuta el método RPC solicitado que se encuentra localizado en el gestor de mensajes.

La ejecución del método RPC comienza con la comprobación de los parámetros requeridos para dicho método. Si estos no se cumplen, se devolvería una respuesta de error. En caso de encontrarse presentes todos los parámetros, se procede a la validación del token.

Los métodos que requieren autenticación incluyen una llamada al método de validación de token situado en el gestor de autenticación. Ahí es donde se va a comprobar que el token sigue siendo válido tanto en tiempo (se encuentre dentro del tiempo de vida que le corresponde) como en la forma (la firma del token es válida para este servidor).

Si por cualquier motivo, el token no es válido, se devolvería un mensaje de error como respuesta. En caso de éxito, se volvería a llamar al gestor de autenticación pero, para esta vez, analizar el contenido del payload del token. En ese payload se va a encontrar el listado de redes a las cuales el usuario puede ejecutar órdenes.

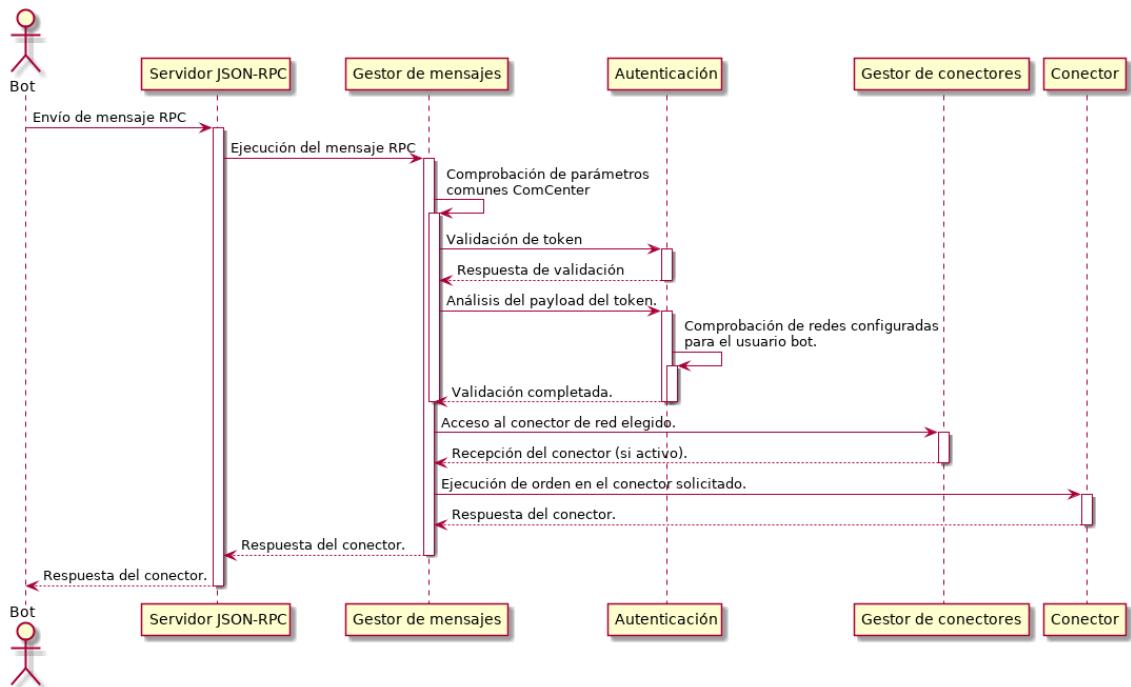


Fig. 10: Diagrama de secuencia del envío / recepción de mensajes

En caso de no encontrarse la red que se solicita en el mensaje, se notificará un error al bot y en caso de encontrar la información de red necesaria, se solicitará el conector necesario para la red solicitada.

Una vez que tenemos el conector de red, ejecutaremos la orden solicitada a través de dicho conector, el cual se encargará de contactar con el servicio api elegido. Una vez que haya recibido la respuesta o error de la api (o del conector si hay condiciones que no se cumplan), se devolverá dicha respuesta al bot.

4.5. Infraestructura

El framework BFMB requiere la ejecución de dos aplicaciones (el bot y el servidor de comunicaciones) y un servidor de Base de Datos (en este caso es MongoDB). Pueden ejecutarse en máquinas independientes o todo en una única máquina.

Ahora, vamos a definir en detalle cada aplicación:

4.5.1. Software bot

El software bot es la aplicación que realizará la persona que quiera utilizar este framework. Dicha aplicación puede desarrollarse en cualquier lenguaje de programación; el único requisito que debe tener para funcionar con este framework es la capacidad de poder usar el protocolo JSON-RPC, ya sea usando una biblioteca externa o una implementación propia.

4.5.2. Servidor de comunicaciones

El servidor de comunicaciones es una aplicación desarrollada en el lenguaje TypeScript, el cual se compila a JavaScript y se ejecuta usando NodeJS. Dicha aplicación contiene las siguientes funcionalidades:

- **Servidor JSON-RPC:** Usando una biblioteca externa llamada jayson, se ha realizado la implementación del servidor JSON-RPC. Lo único que hay que acoplar son los métodos que pueden ser accesibles a través del protocolo. Las aplicaciones bot pueden conectarse a través de protocolo TCP puro, TLS, HTTP o HTTPS. Para los protocolos seguros se requiere disponer de un certificado que pueda ser reconocido por una CA⁶, sea propia o externa.
- **Conectores de servicio:** Los conectores son un elemento imprescindible para este software, son quienes permiten la conexión del servidor con los servicios externos a los que queramos acceder. Por el momento y para este proyecto, se han desarrollado dos conectores de servicio: uno para el servicio de mensajería instantánea Telegram y otro para un servicio que gestiona sistemas de calefacción IoT de la marca Tado°. Son quienes realizan la conexión a la api usando los datos que se envían por parte del bot.
- **Almacenamiento de credenciales:** Para garantizar que solo las aplicaciones autorizadas puedan usar el servidor de comunicaciones, se crean usuarios en una base de datos MongoDB. El detalle de los datos que se almacenan los mostraré en la siguiente sección pero, en resumen, el servidor almacena credenciales para acceder al servidor y los token y/o credenciales de las apis a las que debe acceder.

Pueden ver el diagrama de infraestructura en la figura 11.

4.6. Estructura de base de datos

En la figura 12 pueden ver la sencilla estructura de base de datos que tenemos. Disponemos de dos entidades: usuario y red (UserSchema y NetworkSchema), las cuales vamos a detallar a continuación:

- **UserSchema:** Almacena los datos de acceso para el software se pueda conectar con nuestro servidor de comunicaciones. En ese modelo almacenaremos un nombre de usuario (*username*), una contraseña (*password*) y también almacenaremos la fecha de creación (*createdAt*).
- **NetworkSchema:** Almacena los datos necesarios para que el conector pueda conectarse con la api para la cual se ha realizado el desarrollo. Para ello almacenamos el nombre de la red (*name*), el token (si la api funciona con un token permanente, como en la api de Telegram), el nombre de usuario (*username*) y la contraseña (en los casos donde la autenticación se realiza por usuario y contraseña).

⁶ Autoridad de certificación

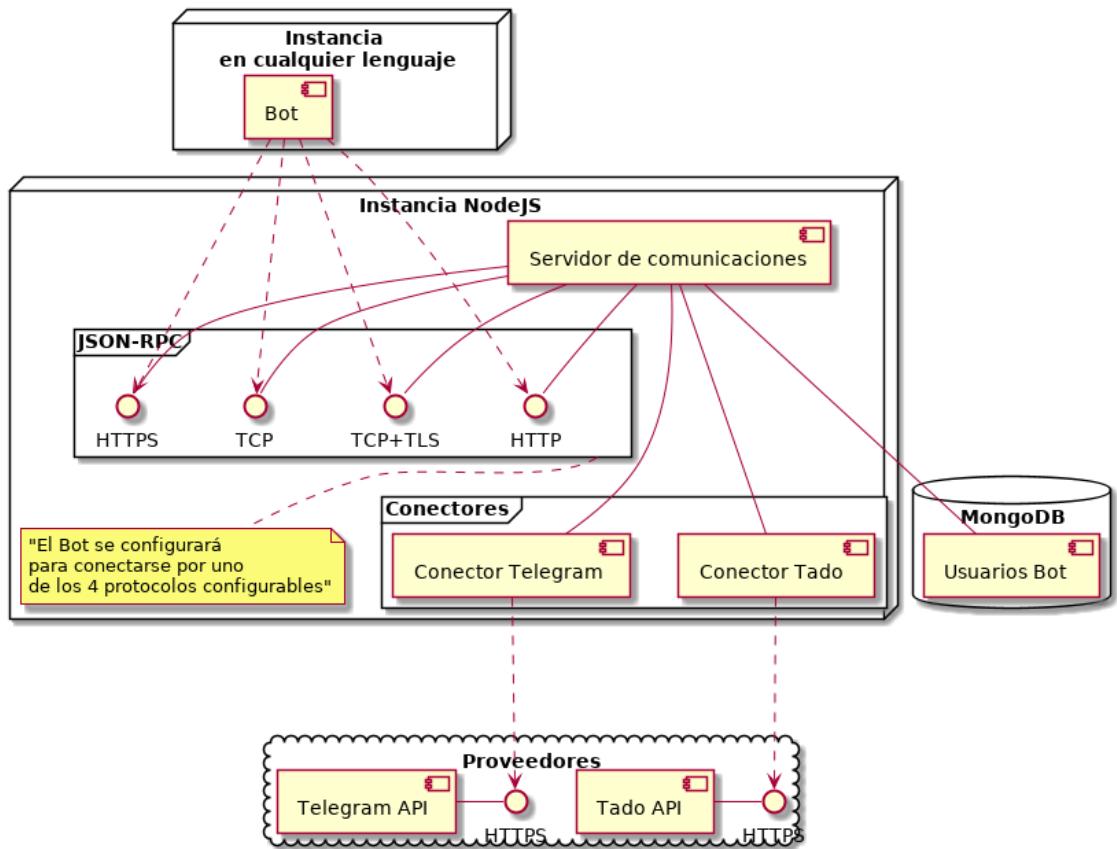


Fig. 11: Esquema de infraestructura (Diagrama de componentes)

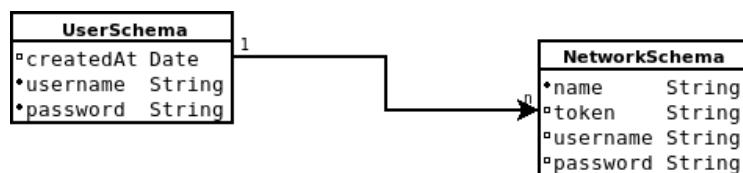


Fig. 12: Estructura de base de datos

La relación que hay entre ambos es una relación *1 a n*, donde para cada usuario (User) puede haber *n* redes (Network).

4.7. Disposición del código

Por último, vamos a explicar la estructura de clases que existe en el framework que se ha desarrollado. Pueden ver el diagrama UML correspondiente en la figura 13.

Actualmente, este framework se compone de cuatro módulos:

- **bfmb-comcenter:** Es el paquete que conforma el servidor de comunicaciones.
- **bfmb-base-connector:** Es el paquete que contiene una abstracción del conector, para así poder realizar llamadas a los conectores usando el polimorfismo⁷.
- **bfmb-telegram-connector:** Es el paquete que contiene el conector correspondiente para usar con la api de Telegram.
- **bfmb-tado-connector:** El el paquete que contiene el conector correspondiente para usar con la api de Tado°.

Los conectores adicionales seguirán el siguiente patrón como nombre:

bfmb-<SERVICIO>-connector

Donde <SERVICIO>será el nombre de la api que corresponda.

Ahora nos centramos en cada módulo:

4.7.1. bfmb-comcenter

El módulo bfmb-comcenter, el servidor de comunicaciones, se compone de cinco clases, las cuales explicamos a continuación:

- **BFMBServer:** Es la clase maestra de este módulo y quien inicia el servidor JSON-RPC. Contiene una instancia de cada clase existente en el módulo y un atributo para acceder a la biblioteca jayson, la cual es la encargada de ejecutar el servidor JSON-RPC. También tiene implementado un patrón (o antipatrón si se abusa de su uso) singleton para poder obtener la instancia activa usando un método estático.
- **MongoEvents:** Es la clase que contiene los métodos callback que se usan ante los eventos de la base de datos MongoDB. Con ello podemos continuar nuestro proceso si se ha logrado una conexión exitosa con la base de datos, mostrar un error en los casos no exitosos o cerrar la base de datos antes de finalizar la aplicación.

⁷ El polimorfismo es una relajación del sistema de tipos, de tal manera que una referencia a una clase acepta direcciones de objetos de dicha clase y de sus clases derivadas. [12, Def. académica de Fernández, L.]

- **AuthHandler:** Es la clase encargada de comprobar que las aplicaciones que se conecten con el servidor de comunicaciones tengan credenciales para ello. Comprueba la existencia de los usuarios en la base de datos y si la contraseña es válida. Tras ello, genera un token de acceso para que pueda validarse sin recurrir a sus credenciales originales por un tiempo definido.
- **ConnectorManager:** Es la clase que contiene todos los conectores activos del servidor, donde se puede llamar al conector que corresponda a la api que solicitemos. Los conectores activos se inicializan en base a los datos de configuración, por lo que no varían en tiempo real.
- **MessageHandler:** Es la clase encargada de procesar los mensajes que llegan al servidor de comunicaciones y ejecutar las acciones que soliciten. Normalmente esta clase busca el conector que corresponda con el servicio donde debe ejecutarse la orden y solicita la conexión que corresponde con su usuario, donde se la envía los datos para dicha api.

4.7.2. **bfmb-base-connector**

El módulo bfmb-base-connector es un módulo que contiene dos clases abstractas, cuyo objetivo es poder realizar polimorfismo entre las clases del servidor de comunicaciones y las clases de cada uno de los conectores de servicio. Se compone de dos clases:

- **Connector:** Es el gestor de la conexiones con la api que gestiona. Almacena n objetos conexión. Esta clase es abstracta y la usan los módulos conectores como clase padre en herencia.
- **Connection:** Es la clase que define la conexión con la api. También es una clase abstracta, la cual usan los módulos conectores como clase padre. También sirve para poder realizar polimorfismo desde el servidor de comunicaciones.

4.7.3. **bfmb-telegram-connector y bfmb-tado-connector**

Los restantes módulos son los conectores con los que podremos conectarnos con las api de los distintos servicios en la red. Ambos módulos contienen como mínimo dos clases: conector (Connector) y conexión (Connection). Estos son clases hija de las clases Connector y Connection del módulo bfmb-base-connector.

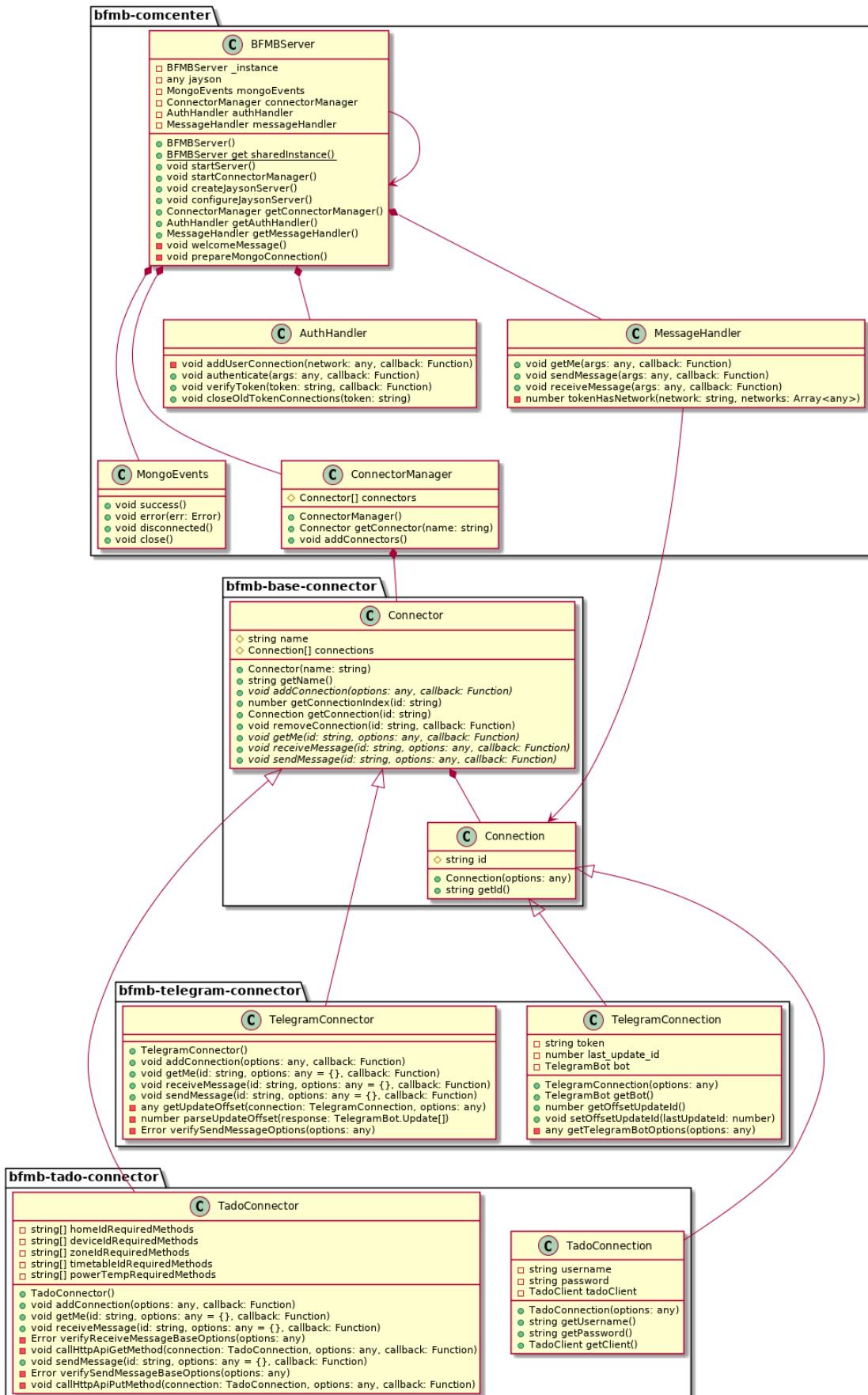


Fig. 13: Diagrama de clases

5. DESARROLLO

5.1. Lenguaje usado en desarrollo

El lenguaje usado para el desarrollo del software de este proyecto es TypeScript, el cual vamos a hablar de él a continuación:

5.1.1. TypeScript

TypeScript es un lenguaje de programación que es un superconjunto de JavaScript al que se le añade el tipado estático y la definición de objetos basados en clases. Es un proyecto de código abierto con licencia Apache 2.0 creado y mantenido por Microsoft.

Este lenguaje ha nacido bajo el contexto de los avances en el rendimiento de los motores de procesamiento de JavaScript y la presencia de NodeJS en el ámbito de los servidores de red. Se empieza a desarrollar aplicaciones más ambiciosas con este lenguaje, pero no deja de haber una gran dificultad cuando se usa JavaScript en aplicaciones a gran escala. El dinamismo de JavaScript juega en contra para estos casos.

El compilador de TypeScript lo único que realiza es comprobación estática de código y transformación a código JavaScript. Por ello, para desarrollar en TypeScript no se requiere un entorno de ejecución especial, ya que se puede ejecutar el código JavaScript.

A continuación vamos a indicar un listado con los puntos a destacar de este lenguaje:

- **Tipado estático de variables:** Otorga la posibilidad de definir el tipo que acepta una variable. Con ello evitamos introducir otro tipo de dato en el mismo, cosa que sí es posible en JavaScript. En caso de querer tener una variable con tipado dinámico, basta con definir la variable con el tipo "any".
- **Definición de clases:** Aunque JavaScript lo permita en la sintaxis ECMAScript 2015, esta funcionalidad está pensada para versiones anteriores de la especificación, como ECMAScript 5 o ECMAScript 3. Además, las clases definidas en TypeScript pueden usar la funcionalidad de tipado estático. Es decir, podremos definir una variable con un tipo que ya hayamos definido en otra parte del código.

También permite herencia y polimorfismo.

Ejemplo en código:

```

1 class BankAccount {
2     constructor(public balance: number) {
3         }
4     deposit(credit: number) {
5         this.balance += credit;
6         return this.balance;
7     }
8 }
```

```

10 class CheckingAccount extends BankAccount {
11     constructor(balance: number) {
12         super(balance);
13     }
14     writeCheck(debit: number) {
15         this.balance -= debit;
16     }
17 }
```

- **Definición de enumerados:** Los enumerados en TypeScript se pueden realizar de una forma más simplificada sin recurrir a la creación de un objeto al que se debe añadir atributos con valores. En los enumerados de TypeScript no es necesario asignar un valor a un elemento del enumerado, se generan valores a la hora de compilar.

Ejemplo en código:

```

1 const enum Operator {
2     ADD,
3     DIV,
4     MUL,
5     SUB
6 }
7
8 function compute(op: Operator, a: number, b: number) {
9     console.log("the operator is" + Operator[op]);
10    // ...
11 }
```

Ejemplo en código para usar el enumerado:

```

1 switch (op) {
2     case Operator.ADD:
3         // execute add
4         break;
5     case Operator.DIV:
6         // execute div
7         break;
8     // ...
9 }
```

- **Clases y métodos genéricos:** TypeScript permite la creación de clases y métodos genéricos. Una clase o método genérico es una clase o método parametrizado en la cual permite usar el tipado fuerte sin conocer aún el tipo que se va a usar.

Ejemplo en código:

```

1 interface NamedItem {
2     name: string;
3 }
4
5 class List<T extends NamedItem> {
```

```

6     next: List<T> = null;
7
8     constructor(public item: T) {
9         }
10
11    insertAfter(item: T) {
12        var temp = this.next;
13        this.next = new List(item);
14        this.next.next = temp;
15    }
16
17    log() {
18        console.log(this.item.name);
19    }
20}

```

- **Espacio de nombres:** TypeScript reduce la complejidad de crear espacios de nombres en el código. En Javascript se define el espacio de nombre como un objeto (al igual que sucede con los enumerados). Acompañó un ejemplo para ello:

```

1 namespace M {
2     var s = "hello";
3     export function f() {
4         return s;
5     }
6 }
7
8 M.f();
9 M.s; // Error, s is not exported

```

Todos los ejemplos de código provienen del documento de especificaciones de TypeScript.

5.2. Dependencias

5.2.1. MongoDB



Fig. 14: Logo de MongoDB

la licencia AGPL.

MongoDB es un motor de base de datos orientado a documento que se encuentra clasificado como una base de datos NoSQL (por su característica de base de datos no relacional). Dicho software tiene dos licencias, donde se usa SSPLv1 en las versiones liberadas después del 16 de Octubre de 2018 mientras que en las versiones anteriores a esta fecha se usa

En el caso de este software de base de datos, los tipos de modelos de datos se diferencian entre documentos y colecciones en lugar de las filas y las tablas de una base de datos relacional. Cada registro en la base de datos se define como un documento, el cual puede estar definido con una variante de JSON pero también con otro formato denominado *Binary JSON (BSON)* que permite mayor variedad de tipos de datos. Los atributos de un documento se pueden comparar con las columnas de una tabla en una base de datos relacional y uno de ellos debe ser una clave primaria con un identificador único. Una colección es un conjunto de documentos de igual estructura de atributos.

MongoDB dispone de una consola para poder manipular la información y consultarla sin necesidad de una aplicación tercera que sigue una sintaxis similar a Javascript en la forma de acceder a los datos. Además, MongoDB dispone de conectores para múltiples lenguajes de programación como C o Javascript. En el caso de este proyecto, no se usa directamente el conector de MongoDB para Javascript, sino que se usa un ODM⁸ llamado *Mongoose* para gestionar la información como los modelos de datos que se han definido en la aplicación.

5.2.2. NodeJS



Fig. 15: Logo de NodeJS

NodeJS es un entorno de ejecución multiplataforma que permite ejecutar código escrito en Javascript fuera del entorno del navegador.

Está diseñado para el desarrollo de aplicaciones de red escalables mediante el uso de la escucha de eventos para procesar las solicitudes de red en lugar de otros modelos de procesamiento. También tiene como objetivo unificar el lenguaje de desarrollo de una aplicación web, ya que toda la lógica la

harías en el mismo lenguaje de programación (JavaScript en el navegador y en el servidor). Este modelo de ejecución en base a eventos tiene un único hilo de ejecución, pero permite la ejecución de múltiples hilos mediante la creación de procesos hijo.

Dispone también de un sistema de paquetes similar al que tiene el lenguaje Python, llamado npm (Node Package Manager) en el que te ofrece diversos módulos para simplificar el desarrollo de aplicaciones y evitar hacer implementación propia de cada funcionalidad (siempre que el módulo de npm tenga buena calidad).

⁸ ODM (Object Document Mapping) es el equivalente al los ORM (Object Relation Mapping) de las bases de datos. Un software que realiza un mapa entre los modelos de datos de la aplicación y sus documentos equivalentes.

5.2.3. Jayson

Jayson es un módulo de NodeJS que se compone de un servidor y un cliente que cumplen con la especificación JSON-RPC 2.0, aunque también es compatible con la versión 1.0. Dispone de estas funcionalidades:

- Puede iniciar múltiples interfaces de red en el mismo proceso. Es decir, se puede iniciar un servidor que escuche el protocolo TCP y, a la vez, escuchar el protocolo HTTPS.
- Soporta los protocolos TCP, HTTP y sus equivalentes seguros (TLS sobre TCP y HTTPS).
- Permite el enrutamiento previo de métodos RPC. Por ejemplo, si tenemos una función que sume dos números y queremos hacer un método que añada dos a un valor numérico podemos utilizar el enrutamiento para llamar al método de suma en su lugar. Actualmente, mi proyecto no usa esta funcionalidad.
- Dispone de un cliente que se puede ejecutar directamente desde la terminal / consola de comandos.

5.2.4. Api de Telegram



Fig. 16: Logo de Telegram

Telegram dispone de gran cantidad de documentación para la realización de bots para su servicio. Además, hay disponible varias bibliotecas para facilitar la tarea de desarrollo. Aun así, hay que realizar unos pasos previos para poder conectar nuestro futuro bot al sistema de forma legítima.

Para crear un bot en Telegram necesitas primero formar parte de los usuarios de Telegram, ya que es necesario contactar con una cuenta especial para crear bots llamada *BotFather*, el cual es el creador y gestor de bots principal.

BotFather dispone de los siguientes comandos:

- **/newbot:** Este comando inicia el asistente para crear un nuevo bot. Te pedirá para ello un nombre visible y un nombre de usuario. El nombre de usuario debe terminar con la palabra "bot".

Una vez creado, te enviará un token. Con ese token se podrá acceder a la api de bots de Telegram y empezar a trabajar.

- **/token:** En caso que el token haya sido expuesto a terceros o lo hayas perdido, puedes solicitar uno nuevo con este comando.

- **/mybots:** Devuelve la lista con los bots que has creado para su configuración.
- **/setname:** Cambia el nombre del bot.
- **/setdescription:** Cambia la descripción del bot. Dicho texto puede tener una longitud de hasta 512 caracteres cuyo fin es describir la función del bot. Los usuarios verán dicho mensaje cuando inicien conversación con el bot.
- **/setabouttext:** Cambia el texto ^{Acerca de}” del bot. Es un texto que no supera los 120 caracteres que se podrá ver en el perfil del usuario correspondiente al bot. Cuando compartes un bot, este texto vendrá adjunto al enlace.
- **/setuserpic:** Cambia la imagen de perfil del bot.
- **/setcommands:** Configura la lista de comandos soportados por el bot. Esto sirve para mostrar un listado de sugerencias cuando un usuario quiera ejecutar un comando. Cada comando tiene un nombre que comienza con el carácter / seguido de hasta 32 caracteres alfanuméricos o carácter _, parámetros y un texto descriptivo.
- **/deletebot:** Borra el bot.

Existen más comandos para gestionar los bots, pero se encuentran fuera de lo realizado para este proyecto, porque también el sistema de bots de Telegram permite crear juegos.

También BotFather puede enviar avisos si el sistema detecta alguna anomalía que pueda suponer una sospecha de fallo en nuestro bot, ya sea porque haya recibido pocos mensajes de lo esperado o por realizar pocas a respuestas frente a los mensajes recibidos. Desde el propio bot BotFather se puede notificar si está solucionado el problema, si se necesita asistencia o si poner en mudo las alertas durante ocho horas o una semana.

Tras indicar el procedimiento de registro de un nuevo bot, debemos pasar a realizar las conexiones pertinentes con los servidores de Telegram. Aunque se puede hacer un software donde construyamos las peticiones HTTP hacia dichos servidores, disponemos de bibliotecas en diversos lenguajes de programación para hacer esa comunicación. Por ello, como la conexión en sí no es objetivo del proyecto y no deberíamos reinventar la rueda, vamos a hacer uso de la biblioteca node-telegram-bot-api del usuario de Github *yagop*.

Estos son los pasos para empezar a usar esta biblioteca:

1. Instalar el paquete a través de npm:

```
1 npm install --save node-telegram-bot-api
```

2. Importar el paquete instalado en el código fuente:

```
1 import * as TelegramBot from "node-telegram-bot-api";
```

-
3. Guardar el token de alguna forma. En este ejemplo, lo guardamos en una constante:

```
1 const token = 'TELEGRAM_TOKEN';
```

4. Crear el objeto TelegramBot:

```
1 const bot = new TelegramBot(token, {polling: true});
```

5. Esta biblioteca permite crear una escucha de eventos del bot, pero en este proyecto no se ha usado ese método para procesar los mensajes entrantes. En cambio, se ha realizado la ejecución manual del método receiveMessages, el cual devuelve todos los mensajes recibidos en un día. Si se manda el contador de mensaje, se recibirán los mensajes que han llegado con un valor superior al contador. A continuación el ejemplo:

```
1 bot.getUpdates({lastUpdateId: 3072}.then(function(response: ←
    TelegramBot.Update[]) {
2   // Caso de éxito. Se recibe un array con todos los mensajes a ←
    // partir de ese contador y que estén almacenados en el ←
    // servidor.
3 })
4 .catch(function(err: Error) {
5   // Caso de error
6 });
```

6. Por último, para enviar un mensaje a un usuario de Telegram, se procede de esta manera mediante el uso del método sendMessage:

```
1 bot.sendMessage(msg.chat.id, "Esto es un texto de prueba", ←
    parametros).then(function(message: TelegramBot.Message) {
2   // Caso de éxito. Se ha logrado enviar el mensaje.
3 })
4 .catch(function(err: Error) {
5   // Caso de error
6 });
```

Esto sería lo básico para interactuar con Telegram a través de la api de bots, pero es posible realizar más tareas dentro del sistema. Los bots y esta biblioteca pueden enviar imágenes, audio, ficheros varios y también leer los ficheros que lleguen por parte de los usuarios.

5.2.5. Api de Tado°



Fig. 17: Logo de Tado°

En el caso de Tado°, disponen de una api privada sin documentar, la cual usan las aplicaciones desarrolladas por dicha empresa. Gracias a las labor de ingeniería inversa realizada por Stephen Phillips, se ha podido realizar el envío de comandos sin necesitar ninguna aplicación oficial. Esa labor ha abierto la puerta a realizar servicios adicionales especiales. Por ejemplo, tenemos el proyecto tado-connect (<https://github.com/Martijn02/tado-connect>), donde configura los termostatos en modo ausente o presente en base a los usuarios conectados a una red wifi como alternativa a la geolocalización a través del smartphone.

En el caso de este proyecto, al igual que con la conexión con Telegram, se ha hecho uso de una biblioteca realizada por Matt Davis llamada node-tado-client (<https://github.com/mattdavis90/node-tado-client>), que evita realizar las peticiones HTTP a mano y dedicar recursos a hacer una tarea que ya cumple este módulo.

A continuación indico los pasos a realizar para poder empezar a usar este módulo:

1. Instalar el paquete a través de npm:

```
1 npm install --save node-tado-client
```

2. Importar el paquete instalado en el código fuente:

```
1 import * as TadoClient from "node-tado-client";
```

3. Crear el objeto TadoClient:

```
1 const tadoClient = new TadoClient();
```

4. Iniciar sesión con las credenciales de Tado°. Si has comprado un termostato de la marca, deberías tener una cuenta creada:

```
1 tadoClient.login("usuario", "contraseña").then(function(response←
    : any) {
2   // Login exitoso
3 })
4 .catch(function(err: Error) {
5   // Login erróneo
6 });
```

5. Por último, puede llamar a los métodos de la api para recibir el tiempo, las temperaturas de las habitaciones, etc.:

```
1 tadoClient.getWeather(home_id).then(function(message: any) {
2   // Caso de éxito. Aquí recibirá el tiempo atmosférico.
3 })
4 .catch(function(err: Error) {
5   // Caso de error
6 });
```

Aparte, aquí indico el listado de métodos disponibles de esta biblioteca:

- **getMe()**: Método que devuelve los datos del usuario, así como de los hogares que tenga configurados.
- **getHome(home_id)**: Método que devuelve los datos del hogar indicado en el home_id.
- **getWeather(home_id)**: Método que devuelve el tiempo atmosférico en la localización del id del hogar indicado.
- **getDevices(home_id)**: Método que devuelve el listado de dispositivos asociados al hogar indicado.
- **getInstallations(home_id)**: Método similar a getDevices pero devolviendo más información.
- **getUsers(home_id)**: Método que devuelve un listado con los usuarios asociados a este hogar y sus dispositivos asociados.
- **getState(home_id)**:
- **getMobileDevices(home_id)**: Indica la lista de dispositivos asociados a ese hogar.
- **getMobileDevice(home_id, device_id)**: Indica el dispositivo móvil asociado.
- **getMobileDeviceSettings(home_id, device_id)**: Indica la configuración de ese móvil asociado.
- **getZones(home_id)**: Devuelve el listado de zonas configuradas en un hogar.
- **getZoneState(home_id, zone_id)**: Devuelve el estado de la zona de un hogar concreto.
- **getZoneCapabilities(home_id, zone_id)**: Indica las capacidades de configuración de una zona.
- **getZoneOverlay(home_id, zone_id)**: Devuelve el ajuste de temperatura manual que se ha configurado para una zona, si este tiene una configuración realizada de forma manual. En caso contrario, devuelve un error 404.

- **getTimeTables(home_id, zone_id):** Indica la programación semanal de una zona.
- **getAwayConfiguration(home_id, zone_id):** Indica la configuración de una zona cuando no hay gente en casa.
- **getTimeTable(home_id, zone_id, timetable_id):**
- **clearZoneOverlay(home_id, zone_id):**
- **setZoneOverlay(home_id, zone_id, power, temperature, termination):** Configura mediante ajuste manual una zona definida. Power debe ser 'on' u 'off'. Temperature debe indicar un número en grados Celsius. Termination debe indicar un número de segundos que dure la configuración o poner valor 'auto' o 'manual'. Si el valor es 'auto', la configuración finaliza cuando cambie el estado en la programación semanal. Si, en cambio, se indica el valor 'manual', la configuración no finalizará hasta que no lo cambie el usuario.
- **identifyDevice(device_id):**
- **apiCall(url, method='get', data={}):** Método de bajo nivel para llamar a futuros endpoints o realizar configuraciones no definidas en la biblioteca. Url indicará la url del endpoint, method indica el HTTP VERB a mandar (GET, POST, PUT, DELETE) y, por último, data indica parámetros a mandar.

6. PRUEBAS

Las pruebas forman una parte importante del desarrollo software. Nos garantiza que cualquier cambio o ampliación en el código no implique la rotura o el comportamiento fuera de lo esperado de un servicio. Para ello, hay que intentar en todo lo posible hacer mecanismos de pruebas automatizados y que cubran todo el código posible. Para este proyecto, se ha usado el framework de desarrollo de pruebas Mocha.

6.0.1. Mocha



Fig. 18: Logo de Mocha

Mocha es un framework para realizar pruebas tanto unitarias como de integración de aplicaciones hechas principalmente con NodeJS, ya que el lenguaje para crear dichas pruebas es JavaScript. Es un framework orientado a facilitar el desarrollo de pruebas para métodos asíncronos (bastante presente en aplicaciones NodeJS).

Para realizar una batería de pruebas con Mocha, disponemos de los siguientes métodos:

- **describe()**: Este método define conceptualmente un grupo de pruebas.
- **context()**: Alias de describe. Realiza el mismo comportamiento que el anterior método.
- **it()**: Define una prueba.
- **before()**: Este método se ejecuta antes de iniciar el grupo de pruebas.
- **after()**: Este método se ejecuta tras realizar todas las pruebas del grupo.
- **beforeEach()**: Este método se ejecuta antes de iniciar cada prueba del conjunto.
- **afterEach()**: Este método se ejecuta tras finalizar cada prueba del conjunto.

Con todo esto es posible realizar la batería de pruebas. A continuación voy a mostrar uno de los grupos de pruebas que tengo para revisar el funcionamiento del servidor de comunicaciones:

```
44         const errMes = new Error('No error was returned.');
45         done(errMes);
46     } else {
47         console.log(util.inspect(response, false, null, true));
48         done();
49     }
50 });
51 });
52 });
```

7. MANUAL DE USUARIO

7.1. Instalación

7.1.1. Prerrequisitos

Para poder trabajar con este framework se necesitan los siguientes requisitos software:

- Sistema operativo GNU/Linux (No puedo garantizar que funcione en otros sistemas operativos pero ningún requisito lo impide)
- NodeJS (versión 8.16).
- Base de datos MongoDB instalado o un acceso a un servidor con dicho motor de base de datos.
- Conexión a Internet (necesario para interactuar con las apis disponibles).

7.1.2. Procedimiento de instalación

Para poner en marcha el servidor de comunicaciones, primero comenzaremos con la instalación de los requisitos:

Instalación de NodeJS Para realizar la instalación de NodeJS puede optar por instalar el paquete que se encuentre en su distribución GNU/Linux o seguir el procedimiento de instalación del propio proyecto: Instalando Node.js usando un gestor de paquetes.

En el caso de Debian, la distribución GNU/Linux que he usado para el desarrollo, el procedimiento es el siguiente:

```

1 # Using Debian, as root
2 curl -sL https://deb.nodesource.com/setup_8.x | bash -
3 apt-get install -y nodejs

```

El script provisto por Nodesource ya realiza todo lo necesario para instalar NodeJS.

Instalación de MongoDB Para realizar la instalación de MongoDB se puede recurrir al paquete que se encuentre en el repositorio de su distribución. Eso sí, no está exento de inconvenientes porque ha habido un cambio de licencia en dicho software y esa nueva licencia (creada por la propia gente de MongoDB) genera conflicto con algunas distribuciones de GNU/Linux (SSPLv1)⁹.

En Debian Stretch y Jessie sigue disponible MongoDB, pero en Debian Buster (la futura versión estable) ya no se encuentra dicho paquete. Para ello toca seguir estos pasos:

⁹ Server Side Public License

- Añadir los repositorios unstable de Debian. Hay que modificar el fichero /etc/apt/sources.list

```

1 deb http://debian.redimadrid.es/debian/ buster main contrib non-free
2 deb-src http://debian.redimadrid.es/debian/ buster main
3
4 deb http://security.debian.org/debian-security buster/updates main
5 deb-src http://security.debian.org/debian-security buster/updates main
6
7 # Hay que añadir esta línea.
8 deb http://debian.redimadrid.es/debian/ unstable main contrib non-free

```

- Después, hay que editar o crear el fichero mypreferences en /etc/apt/preferences.d/ y añadir el siguiente contenido:

```

1 Package: *
2 Pin: release a=testing
3 Pin-Priority: 400
4
5 Package: *
6 Pin: release a=unstable
7 Pin-Priority: 300

```

Con esto hacemos que los paquetes que se encuentran en el repositorio testing (buster) tengan mayor prioridad que unstable. Así evitamos instalar paquetes de unstable más allá del propio MongoDB.

- Por último, ejecutamos el comando apt de esta forma para instalar el paquete mongodb.

```
1 sudo apt install -t unstable mongodb
```

Este procedimiento no está exento de cambios y posiblemente acabe siendo anulado mientras pase el tiempo, ya que la última versión que mantendrá Debian será las que sean anteriores a mongo 4.0 (donde empieza el cambio de licencia). Tampoco está exento la posibilidad de cambios en el software para cambiar de motor de base de datos (véase: Líneas Futuras).

Instalación del servidor de comunicaciones Para realizar la instalación del servidor de comunicaciones debemos seguir estos pasos:

- Primero debemos clonar el repositorio del servidor usando git. Si no está git instalado, en Debian basta con este comando:

```
1 sudo apt install git
```

Después, realizamos la clonación:

```
1 git clone https://github.com/BFMBFramework/ComCenter
```

2. Una vez obtenida la aplicación, tenemos que entrar en la carpeta raíz del proyecto e instalar las dependencias necesarias.

```
1 cd ComCenter
2 npm install
```

3. Una vez realizado esto, ya tenemos instalado el servidor de comunicaciones. Actualmente, los conectores vienen incluidos en el proceso de instalación de dependencias. No hace falta instalarlos aparte.

Configuración del servidor de comunicaciones Para configurar el servidor de comunicaciones, disponen de una configuración de ejemplo en la carpeta raíz del software. Hagan una copia del mismo y editen el fichero.

```
1 cp config-example.json config.json
2 nano config.json
```

Ahora explicamos qué opciones están disponibles en la configuración:

```
1 {
2   "db": {
3     "url": "mongodb://localhost/comcenter",
4     "encKey": "yfbmTM8/lZAH8H6PkGTcyK0H1bMk4k10ovwq1xhX75w=",
5     "signKey": "q8N6L+Lo345JsnU1Sow4sp1Kzo+←
6       XRJPYjT2Uq7mh1OkSi5kFkX0UoIY3etfm4UxtNHaM8xaX2HtkAhV7Gye0KA==←
7   },
8   "tokenConfig": {
9     "secret": "W]5!>v$@NV!bgk8T?{$vP~o' ^9P}S1",
10    "algorithm": "HS512",
11    "expiresIn": "24h"
12  },
13  "servers": [
14    {
15      "type": "tcp",
16      "port": 3000
17    },
18    {
19      "type": "http",
20      "port": 3001
21    },
22    {
23      "type": "https",
24      "port": 3002,
25      "cert": "/home/angel/certificados/certi.pem",
26      "key": "/home/angel/certificados/certikey.key"
27    }
28  ],
29  "logLevel": "info"
30}
```

```

27  {
28    "type": "tls",
29    "port": 3003
30  }
31 ],
32 "modules": [
33   "telegram",
34   "tado"
35 ]
36 }
```

- **db:** Es el objeto donde definimos la configuración de base de datos.

- **db.url:** En este campo se introduce la url de conexión con la base de datos de MongoDB.
- **db.encKey:** La base de datos se encuentra cifrada. Para ello, se necesita una pareja de claves para realizar el cifrado con estas. En este campo se almacena una clave de cifrado de 32 bytes en formato base64 (clave de 256bits).
- **db.signKey:** En este campo se introduce la clave de firma de la base de datos. Se requiere que sea una clave de bytes en formato base64 (clave de 512 bits). Para generar ambas claves se puede usar openssl de la siguiente forma:

```

1 openssl rand -base64 32; openssl rand -base64 64;
2
3 yZ4dEBn2RILEOGA04kg6oIJHsm6QO4kcbT2jnJAaFTo= # Clave de ←
4   cifrado
5 7n24rUz8G8KKojMo3LhUBvv2TUBW7LQPg+←
6   FkyzFtAPSaF3NimFBPhNgWM9s4n5fX
7 251ZD20BTd24OTKSmBSxNQ== # Clave de firma
```

- **tokenConfig:** Es el objeto donde definimos la configuración de los tokens que generará el servidor de comunicaciones.

- **tokenConfig.secret:** En este campo introducimos el secreto que usará jsonwebtoken para generar los tokens con firma. El formato es un string y acepta caracteres alfanuméricos y símbolos.
- **tokenConfig.algorithm:** En este campo indicaremos el algoritmo de firma a usar para la generación de los tokens. Por el momento se ha probado con algoritmos simétricos (HMAC con hash SHA), los cuales son:
 - **none:** El token no se firma con ningún algoritmo.
 - **HS256:** El token se firmará con el algoritmo HMAC con hashing SHA256
 - **HS384:** El token se firmará con el algoritmo HMAC con hashing SHA384
 - **HS512:** El token se firmará con el algoritmo HMAC con hashing SHA512

En el futuro se definirán atributos para los algoritmos de firma asimétricos, ya que estos requieren clave privada y clave pública.

- **tokenConfig.expiresIn:** Campo en el que definimos la duración de los tokens generados.

Si introducimos un valor numérico (no string), se considerará el tiempo de vida en segundos.

Si introducimos un string con el valor numérico seguido de un carácter que pueda reconocerse como una unidad de tiempo (h, m, d, s, ms), se configurará el tiempo de vida del token en base a esa unidad. Por ejemplo, "24h" serían 24 horas y "7d" serían 7 días.

En caso que el string solo tenga caracteres numéricos, se expresará como milisegundos.

- **servers:** Servers es un array en el cual indicamos los servidores JSON-RPC que tendremos operativos, los cuales se configuran de la siguiente forma.

- **type:** En este campo indicamos el protocolo con el que levantaremos el servidor. Puede configurar los siguientes tipos:

- **tcp:** Servidor JSON-RPC usando el protocolo TCP.
- **http:** Servidor JSON-RPC usando el protocolo HTTP.
- **tls:** Servidor JSON-RPC usando el protocolo TLS.
- **https:** Servidor JSON-RPC usando el protocolo HTTPS.

- **port:** Puerto por el que escuchará este servidor.

- **cert:** (*Requerido cuando type es tls o https, ignorado en otros casos*) Ruta de acceso al fichero del certificado que vayamos a usar para securizar las conexiones a este servidor.

- **key:** (*Requerido cuando type es tls o https, ignorado en otros casos*) Ruta de acceso al fichero con la clave privada del certificado que hayamos seleccionado.

En caso de no encontrar el fichero cert y el fichero key en las rutas indicadas (o la omisión de los mismos) impedirá el funcionamiento de los servidores con protocolo tls o https y, por ello, no se iniciarán.

- **modules:** En este array indicaremos los módulos que arrancaremos de BFMBFramework, es decir, los conectores a las apis que necesitemos. Para ello, hay que introducir un string con el nombre del servicio.

Es decir, si queremos iniciar el conector bfmb-telegram-connector, debemos introducir Telegram. Si queremos iniciar el conector bfmb-tado-connector, debemos introducir "Tado"(y así con el resto).

7.1.3. Inicio del servidor

Una vez que tengamos el servidor configurado, podremos iniciar lo con este comando. Desde la carpeta raíz del servidor:

```
1 ./bin/comcenter
```

Esta sería la salida esperada:

```
1 info: Using configuration file from: /home/angel/Proyectos/PFG/←
    ComCenter/config.json
2 info: Welcome to BFMB ComCenter 0.5.0
3 info: Connecting to MongoDB...
4 info: Connected to MongoDB database
5 info: Attaching connectors to Connector Manager
6 info: Raising tcp server on port 3000
7 info: Raising http server on port 3001
8 info: Raising https server on port 3002
9 error: Can't raise https server. No certificates.
10 info: Raising tls server on port 3003
11 error: Can't raise tls server. No certificates.
12 node-telegram-bot-api deprecated Automatic enabling of cancellation ←
    of promises is deprecated.
13 In the future, you will have to enable it yourself.
14 See https://github.com/yagop/node-telegram-bot-api/issues/319. module←
    .js:653:30
```

Como pueden ver, si no se encuentran los certificados que se buscan, saltará un error en los logs para indicar que no se han podido iniciar los servidores tls y https configurados pero no detendrá la aplicación.

7.2. Añadir usuarios al servidor de comunicaciones

Para añadir usuarios al servidor, como no tenemos ninguna aplicación de administrador por el momento, tengo hecho un script en lenguaje Javascript que añade los usuarios que queramos.

```
1 var mongoose = require('mongoose');
2 var config = require('../config.json');
3 var User = require('../dist/schemas/user').User;
4 var Network = require('../dist/schemas/network').Network;
5
6 mongoose.connect(config.db.url, {useNewUrlParser: true});
7
8 if (!process.env.TG_TOKEN) throw new Error("No Telegram token defined←
    .");
9 if (!process.env.TADO_USERNAME) throw new Error("No Tado login data ←
    defined.");
10 if (!process.env.TADO_PASSWORD) throw new Error("No Tado login data ←
    defined.");
11
12 var test = new User({
13   username: 'TEST',
14   password: 'test'
15 });
16
```

```

17 test.save(function (err) {
18   if (err) throw err;
19
20   var tgNetwork = new Network({
21     name: 'Telegram',
22     token: process.env.TG_TOKEN
23   });
24
25   tgNetwork.save(function (err) {
26     if (err) throw err;
27     console.log("Added Telegram network");
28   });
29
30   var tadoNetwork = new Network({
31     name: 'Tado',
32     username: process.env.TADO_USERNAME,
33     password: process.env.TADO_PASSWORD
34   );
35
36   tadoNetwork.save(function (err) {
37     if (err) throw err;
38     console.log("Added Tado Network");
39   })
40
41   test.networks.push(tgNetwork);
42   test.networks.push(tadoNetwork);
43   test.save(function (err) {
44     if (err) throw err;
45     console.log("Added test user.");
46   });
47 });

```

Para añadir el usuario con username 'TEST' y password 'test' basta con ejecutar el script de esta forma.

¹ TG_TOKEN=<token de bot Telegram> TADO_USERNAME=<usuario de Tado°> ↫
TADO_PASSWORD=<contraseña de Tado°> nodejs ./add-mocha-user.js

Para cambiar el username y el password a añadir basta con editar las líneas 13 y 14 del script. No es la mejor forma de configurar usuarios y en el futuro se implementará una aplicación de gestión con el fin de facilitar esta tarea.

7.3. Cómo interactuar con el servidor de comunicaciones

Como se puede ver en la figura 19, nuestro automatismo se conecta únicamente con el servidor de comunicaciones usando para ello el protocolo JSON-RPC, por lo que se puede realizar el bot o automatismo con cualquier lenguaje que tenga un cliente de dicho protocolo.

Para los ejemplos en código, voy a usar TypeScript y usará el cliente jayson.

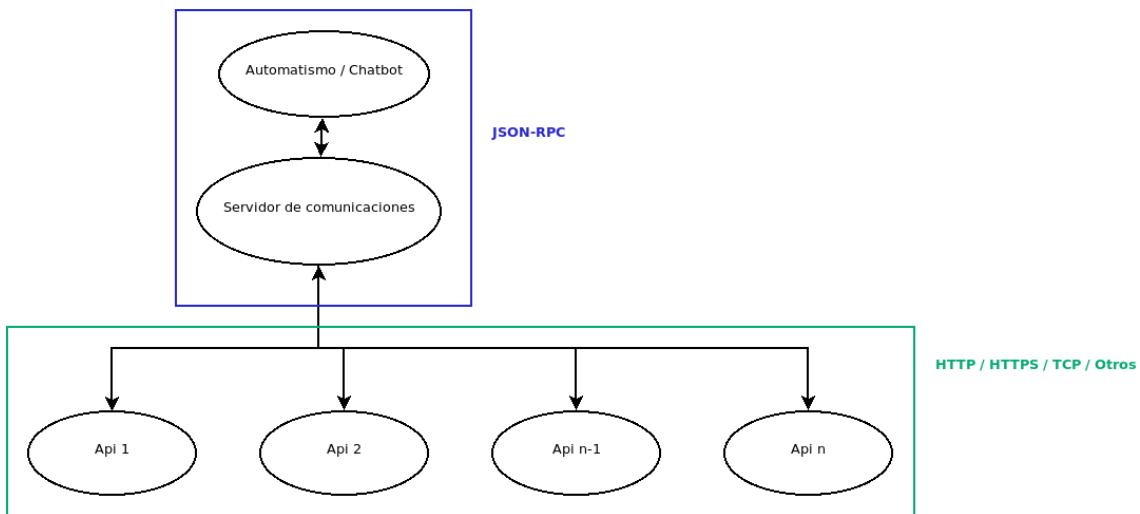


Fig. 19: Comunicación entre sistemas

7.3.1. Realizar login en el servidor

Para poder comunicarse con el servidor, es necesario iniciar sesión en él. Para ello, hay que ejecutar el comando RPC 'authenticate' pasando como opciones el username y el password del usuario del servidor.

```

1 this.jayson.request('authenticate',
2 { username: 'username', password: 'password' },
3 function(err: Error, response: any) {
4     if (err) {
5         // Caso error
6     } else {
7         // Caso éxito
8     }
9 });

```

7.3.2. Obtener información del usuario de la api

Para obtener los datos del usuario de la api que se quisiese acceder, basta con mandar el comando RPC 'getMe'. Como opciones es necesario el token que habremos recibido tras realizar la autenticación con el comando anterior, la red a la que queremos enviar esta orden (recuerde que debe empezar en mayúscula) y las opciones para dicha red (en este caso será un objeto vacío).

```

1 this.jayson.request('getMe',
2 {token: 'token_srv_comunicaciones', network: 'Mired', options: {}},
3 function(err: Error, response: any) {
4     if (err) {
5         // Caso error
6     } else {

```

```
7     // Caso éxito
8   }
9 });
```

7.3.3. Comando de recepción de información

Para realizar una llamada que equivaldría al método HTTP GET, usamos el comando RPC 'receiveMessage'. Como opciones es necesario el token recibido al autenticarse en el servidor de comunicaciones, también hay que indicar la red a la que vamos enviar el comando. Por último, las opciones forman parte de los parámetros necesarios para solicitar la recepción de datos. Por ejemplo, el conector Tadoº necesita las opciones *api_method* y *home_id*. Si no se enviase algún parámetro obligatorio, el error que devolviese el comando lo indicará.

```
1 this.jayson.request('receiveMessage',
2 {token: 'token srv comunicaciones', network: 'Mired',
3 options: {api_method: 'getZones', home_id: 'home_id'}},
4 function(err: Error, response: any) {
5
6});
```

7.3.4. Comando de envío de información

Para realizar una llamada que equivaldría a los métodos POST, PUT y DELETE de HTTP, usamos el comando RPC 'sendMessage'. Como opciones también necesitamos el token recibido (al igual que en los otros métodos), también es necesario indicar la red a la que vamos a enviar el comando. Pôr último, las opciones forman parte de los parámetros necesarios para solicitar el envío de información. Por ejemplo, en el caso de usar el conector de Tadoº, es necesario usar los parámetros *api_method* y *home_id*. También en este caso, si no se llegase a enviar un parámetro que fuese obligatorio, el error devuelto lo indicará.

```
1 this.jayson.request('sendMessage',
2 {token: 'token srv comunicaciones', network: 'Mired',
3 options: {api_method: 'setZoneOverlay', home_id: 'home_id'}},
4 function(err: Error, response: any) {
5
6});
```


8. CONCLUSIONES

Como se ha podido ver en el resumido estado del arte, podemos ver que el campo de los chatbots lleva años desarrollándose, añadiendo nuevos componentes de interacción, como el reconocimiento de voz. Además, se pueden recorrer varios caminos en los chatbots, en los que pueden estar definidos de forma manual o pueden implementar mecanismos de aprendizaje mediante Inteligencia Artificial.

Ya hay ejemplos comerciales de chatbots o de asistentes virtuales (el desarrollo de estos) dentro del ámbito general pero también se están desarrollando chatbots que estén orientados a ciertos servicios específicos como el servicio sanitario.

Aun así, para llegar a ese nivel de complejidad que tienen los asistentes virtuales, se han tenido que dedicar muchos esfuerzos en su desarrollo. La idea de este proyecto es y será la de simplificar dichos desarrollos.

Este proyecto no terminaría en este punto, sino que hay que seguir añadiendo nuevos conectores para otros servicios y, para que se genere un entorno colaborativo, hay que proveer a los colaboradores de una buena base. En las líneas futuras indico una lista de mejoras a futuro de este proyecto, entre las cuales está un cambio de motor de base de datos para aumentar la portabilidad del software o, mejor aún, dar opción a elegir motor de base de datos.

Lo último a tener en cuenta, cosa que desarrollo en la sección de aspectos éticos es que esto es una herramienta para un desarrollo mayor y, como cualquier herramienta, se puede usar para cualquier actividad, ya sea lícita o ilícita. También hay que indicar que el desarrollo de chatbots, sobre todo aquellos con aprendizaje mediante Inteligencia Artificial, les queda mucho para alcanzar un nivel que permita cierta confianza en sus resultados.

9. ASPECTOS ÉTICOS, SOCIALES, PROFESIONALES Y MEDIOAMBIENTALES DEL PROYECTO

Como en todas las herramientas software, no está exento que el mismo sea aprovechado para realizar actividades ilícitas. En el caso de este framework, la responsabilidad de consumir recursos de las apis de forma eficiente recae en el desarrollador del bot y no en el framework. El framework va a ayudar al desarrollador pero no va a pensar y actuar por él.

Con ello, se puede decir claramente que este proyecto se podría usar con fines ilícitos y, si dijera lo contrario en este documento, estaría mintiendo. Cualquier persona con conocimientos de programación puede usar este software para desarrollar, por ejemplo, un automatismo que se dedique a hacer spam a diestro y siniestro (El spam es ilícito porque solamente genera molestias a los destinatarios).

En cuanto al nivel social, este proyecto es Open Source debido a la licencia con la que se ha firmado (MIT), por ello permite la libertad de uso y modificación de este software y cualquiera puede aportar novedades y usarlo en proyectos más grandes como un módulo del mismo. Esto ya genera un valor social por la disponibilidad que hay del código fuente.

A nivel profesional y medioambiental, se espera que este proyecto mejore el rendimiento de producción y, con ello, menor consumo de recursos. Si un desarrollador no tiene que dedicar tiempo en el sistema de comunicación, dedicará sus recursos a otros componentes de importancia del sistema automático / chatbot.

Ser eficiente también es evitar reinventar la rueda y aprovechar lo que ya hay existente. Este proyecto es claro ejemplo, ya que los conectores que realicé para la demostración usan herramientas que ya han desarrollado otras personas. Por ejemplo, el conector que conversa con la api de Tado° usa una biblioteca llamada node-tado-client desarrollado por Matt Davis, quien a su vez se basó en el trabajo de ingeniería inversa que realizó Stephen Phillips.

10. LÍNEAS FUTURAS

Tras el desarrollo actual y los resultados dados, se puede ver que hay distintas líneas futuras a partir de ahora. Algunas de ellas son mejoras del software para permitir una mayor portabilidad y otras pertenecen a posibles desarrollos adicionales.

- Cambiar el motor de base de datos usado en el servidor de comunicaciones.

Actualmente se usa MongoDB por conveniencia en el desarrollo de este proyecto (el uso de la biblioteca mongoose ayudó mucho), pero sería más apropiado usar bases de datos portables como SQLite o NeDB y evitar la instalación de un servidor adicional para el poco volumen de información que tiene que almacenar, aunque eso conlleve la modificación de parte del código fuente para pasar de un motor de base de datos a otro. También se podría optar por desarrollar un modelo universal y que sea el usuario quien elija en base a sus necesidades.

Otro motivo para abrir esta línea de desarrollo es el cambio de licencia de MongoDB en su versión más reciente (4.0). Eso está generando rechazo en la comunidad del software libre y la comunidad opensource, llegando al punto que ciertas distribuciones GNU/Linux (Fedora y Debian) estén eliminando o planteando que se elimine este software de sus repositorios. [14] [16]

- Incorporar un gestor web de usuarios para el servidor de comunicaciones.

Actualmente, se están añadiendo los usuarios y configuraciones a través de un script. La incorporación de un gestor web serviría para facilitar la configuración del mencionado servidor. Se podría plantear incluso desarrollar el gestor como una aplicación por comandos independiente del software servidor.

- Acoplar un motor de reconocimiento de lenguaje natural para la parte bot.

Ahora mismo, el bot realizado para este proyecto no procesa lenguaje natural, sino que se basa en comandos para la realización de órdenes. Se podría incorporar una implementación de ELIZA con un script personalizado para estos casos de uso. En lugar de un script psicoterapeuta, uno que reconozca frases del estilo "Pon la cocina a 23°C" y lo asocie con la orden que debe ejecutar. Se tardaría menos en realizar esto que crear un motor IA desde cero.

- Cambiar la abstracción del servidor de comunicaciones a métodos de api REST (Crear, consultar, editar y eliminar). Sería la equivalencia a las acciones HTTP POST, GET, PUT y DELETE en lugar de tener una abstracción de dos métodos (*receive* y *send*).

- Implementación de un contenedor Docker que permita la implantación de este sistema sin la necesidad de enfocar los esfuerzos en lograr el entorno de trabajo adecuado.

ANEXOS

Códigos de error del servidor de comunicaciones (ComCenter)

Errores genéricos

- **100:** Error en los parámetros del mensaje JSON-RPC para ComCenter.

Autenticación

- **300:** El usuario no existe en la base de datos. Se ha intentado iniciar sesión con un usuario que no existe.
- **301:** No coincide la contraseña. Si el hash de la contraseña introducida no coincide con el hash almacenado, se devuelve este código de error.
- **302:** No se ha encontrado el token en el mensaje JSON-RPC. Se ha intentado ejecutar un método JSON-RPC que requiere autenticación sin la presencia del token.
- **399:** Otros errores de autenticación no definidos.

Sistema de mensajes

- **400:** El usuario no tiene configurada la red solicitada.
- **401:** El conector de red está desactivado en la configuración del servidor.
- **402:** Los parámetros de las opciones enviadas al conector de red correspondiente son incorrectas. / El conector recibe un error.
- **499:** Otros errores del sistema de mensajes.

Listado de software usado para la realización de este PFG

Programación

- **Sublime Text 3:** Editor de texto apto para programar.

Documentación

- **Gummi:** Editor de documentos L^AT_EX.
- **PlantUML:** Generador de diagramas UML a partir de guiones escritos en un lenguaje de marcado propio.
- **Dia:** Editor de diagramas visual.

Código fuente

El código fuente más reciente de este proyecto se puede encontrar en los siguientes repositorios:

- **ComCenter:** <https://github.com/BFMBFramework/ComCenter>
- **BaseConnector:** <https://github.com/BFMBFramework/BaseConnector>
- **TadoConnector:** <https://github.com/BFMBFramework/TadoConnector>
- **TelegramConnector:** <https://github.com/BFMBFramework/TelegramConnector>
- **TadoBot (software demo):** <https://github.com/BFMBFramework/TadoBot>

Referencias

- [1] PÉREZ-DIAZ, D. y PASCUAL-NIETO, I., *Conversational Agents and Natural Language Interaction: Techniques and Effective Practices*, IGI Global, 2011 ISBN: 9781609606183
- [2] CERDAS MENDEZ, D. *Historia de los chatbots y asistentes virtuales*, Planeta Chatbot, <https://planetachatbot.com/evolucion-de-los-chatbots-48ff7d670201>, [Visitado el 4/3/2019]
- [3] MARKOFF, J. *Joseph Weizenbaum, Famed Programmer, Is Dead at 85*, The New York Times, <https://www.nytimes.com/2008/03/13/world/europe/13weizenbaum.html>, [Visitado el 12/3/2019]
- [4] PHILLIPS, S. *The Tado API v2*, SCPHillips.com, <http://blog.scphillips.com/posts/2017/01/the-tado-api-v2/>, [Visitado el 12/3/2019]
- [5] JSON-RPC WORKING GROUP *JSON-RPC version 2.0 specification*, <https://www.jsonrpc.org/specification>, [Visitado el 8/4/2019]
- [6] CLEANTECH INVESTOR *Tado° raises €10m from Target Partners and Shortcut Ventures*, <https://web.archive.org/web/20140821174647/http://www.cleantechinvestor.com/portal/mainmenucomp/companiest/3258-tado/11706-tado-raises.html>, [Visitado el 17/4/2019]
- [7] TADO GMBH *Technical documentation of Tado*, http://www.free-instruction-manuals.com/pdf/pa_1184164.pdf, [Visitado el 19/4/2019]
- [8] JIMÉNEZ RUIZ, L. *Diseño e implementación de etapa de comunicación basada en 6LoWPAN para plataforma modular de redes de sensores inalámbricas*, Archivo documental de la UPM, 2016, http://oa.upm.es/43013/1/TFG_LUIS_JIMENEZ_RUIZ.pdf, [Visitado el 19/4/2019]
- [9] TELEGRAM *Telegram FAQ*, <https://telegram.org/faq>, [Visitado el 19/4/2019]
- [10] TELEGRAM *Telegram Login for Websites*, <https://telegram.org/blog/login>, [Visitado el 19/4/2019]
- [11] J. GUTIÉRREZ, J. *¿Qué es un framework web?*, http://www.lsi.us.es/~javierj/investigacion_ficheros/Framework.pdf, [Visitado el 5/5/2019]
- [12] ÁLVAREZ, M.Á. *Polimorfismo en Programación Orientada a Objetos*, <https://desarrolloweb.com/articulos/polimorfismo-programacion-orientada-objetos-concepto.html>, [Visitado el 12/5/2019]

- [13] LUNDGREN, T. Y OTROS *Jayson is a simple but featureful JSON-RPC 2.0/1.0 client and server for node.js*, <https://github.com/tedeh/jayson/blob/master/README.md>, [Visitado el 8/5/2019]
- [14] MUYLINUX *MongoDB pincha en hueso: nadie acepta su nueva licencia*, <https://www.muylinux.com/2019/01/17/mongodb-rechazo-nueva-licencia/>, [Visitado el 12/5/2019]
- [15] NODESOURCE *README de distributions en GitHub*, <https://github.com/nodesource/distributions/blob/master/README.md#deb>, [Visitado el 12/5/2019]
- [16] DEBIAN BUG REPORTS #916107 - *mongodb: MongoDB should not be part of a stable release*, <https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=916107>, [Visitado el 15/5/2019]
- [17] LOUIS CHATRIOT *NeDB: The JavaScript Database, for Node.js, nw.js, electron and the browser*, <https://github.com/louischatriot/nedb>, [Visitado el 21/5/19]
- [18] CAIXABANK *Web Neo/Banca Digital/CaixaBank*, <https://www.caixabank.es/particular/banca-digital/web-neo.html?loce=sh-part-Neo-5-terrat-BancaDigital-WebCaixabanknow-NA>, [Visitado el 22/5/19]
- [19] AIML FOUNDATION *AIML Foundation*, <http://www.aiml.foundation>, [Visitado el 27/5/19]
- [20] APPLE SUPPORT *Use Siri on all your Apple devices*, <https://support.apple.com/en-us/HT204389>, [Visitado el 1/6/19]
- [21] APPLE DEVELOPER *SiriKit*, <https://developer.apple.com/sirikit/>, [Visitado el 1/6/19]
- [22] BIANCA BOSKER *SIRI RISING: The inside story of Siri's origins - And why she could overshadow the iPhone - Huffpost*, https://www.huffingtonpost.com/2013/01/22/siri-do-engine-apple-iphone_n_2499165.html, [Visitado el 1/6/19]
- [23] GOOGLE *El Asistente de Google /Tu Google personalizado*, https://assistant.google.com/intl/es_es/, [Visitado el 3/6/19]
- [24] AMAZON *Amazon Alexa*, <https://developer.amazon.com/es/alexa>, [Visitado el 3/6/19]
- [25] CHEMA ALONSO (EL LADO DEL MAL) *Un informático en el lado del mal: NegoBot: Una bot para detectar pedófilos en chats*, <http://www.elladodelmal.com/2013/07/negobot-una-bot-para-detectar-pedofilos.html>, [Visitado el 4/6/19]

- [26] LAORDEN, C., GALÁN, P. y OTROS *Negobot: Agente conversacional basado en teoría de juegos para la detección de conductas pedófilas.*, <http://paginaspersonales.deusto.es/claorden/publications/2012/Negobot.pdf>, [Visitado el 4/6/19]
- [27] SOMASEGAR'S BLOG (MSDN) *TypeScript: JavaScript Development at Application Scale*, <https://blogs.msdn.microsoft.com/somasegar/2012/10/01/typescript-javascript-development-at-application-scale/>, [Visitado el 8/6/19]
- [28] TYPESCRIPT Documentation - *TypeScript*, <https://github.com/Microsoft/TypeScript/blob/master/doc/spec.md>, [Visitado el 8/6/19]
- [29] HEAVEN, D. *Your next doctor's appointment might be with an AI*, <https://www.technologyreview.com/s/612267/your-next-doctors-appointment-might-be-with-an-ai/>, [Visitado el 9/6/19]
- [30] ALEXA SKILLS KIT *Build Skills with the Alexa Skills Kit*, <https://developer.amazon.com/docs/ask-overviews/build-skills-with-the-alexa-skills-kit.html>, [Visitado el 11/6/19]
- [31] EISENZOPF, J. *Building your first Action for Google Home (in 30 minutes)*, <https://medium.com/google-cloud/building-your-first-action-for-google-home-in-30-minutes-ecc>, [Visitado el 11/6/19]
- [32] KANBANIZE *Qué es Kanban: Fundamentos*, <https://kanbanize.com/es/recursos-de-kanban/primeros-pasos/que-es-kanban/>, [Visitado el 13/6/19]
- [33] CORIWEB Método Kanban: *¿Qué es y cuáles son sus principios básicos?*, <https://www.coriaweb.hosting/metodo-kanban-cuales-principios-basicos/>, [Visitado el 13/6/19]
- [34] ROUSE, M. *What is MongoDB? - Definition from WhatIs.com*, <https://searchdatamanagement.techtarget.com/definition/MongoDB>, [Visitado el 14/6/19]
- [35] SÁNCHEZ FERNÁNDEZ, J. *Testeando un API REST node.js con Mocha*, <http://xurxodev.com/testeando-un-api-rest-con-mocha/>, [Visitado el 14/6/19]