Escuela Técnica Superior de Sistemas Informáticos

Universidad Politécnica 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

2019

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 reasearch 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 duplicities 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

Índice de figuras

Índice de cuadros

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 Now *(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 ”engañar” 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 Artifical 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. Actualidad: Asistentes virtuales

2.2. Usos de los chatbots aplicados al mundo real

2.2.1. Seguridad

Hablar sobre Negobot (chatbot para detectar presuntos pedófilos)

2.2.2. Sanidad

Hablar sobre sistemas multiagente para detección de ETS.

2.2.3. Banca

Hablar sobre Neo (Caixabank)

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 cuatros 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 includo, 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. Tadoº

Tadoº 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

[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.

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, 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 6LoWPAN[2] 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 1.

También Tadº 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.

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).

[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.

(a) Puente de conexión

(b) Termostato Tado°

(c) Válvula termostática Tado°

Fig. 1: Aparatos de gestión de calefacción de la empresa Tado°

3.3.2. Telegram

Telegram es un servicio de mensajería instantánea y VoIP[3] 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 requiren 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.

- **Chats secretos:** Telegram provee la opción de usar chats secretos entre dos usuarios. Para ello, se utiliza cifrado extremo a extremo[5] Por defecto, Telegram no activa

[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.

[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.

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 terceras, siempre que estas incluyan los componentes necesarios para ello.

4. DISEÑO Y ARQUITECTURA DEL SISTEMA

4.1. Metodología de trabajo

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 çonectores", 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 (Tadoº).

Pueden ver el diagrama de casos de uso en la figura 2.

4.3. 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 ejecujarse en máquinas independientes o todo en una única máquina.

Ahora, vamos a definir en detalle cada aplicación:

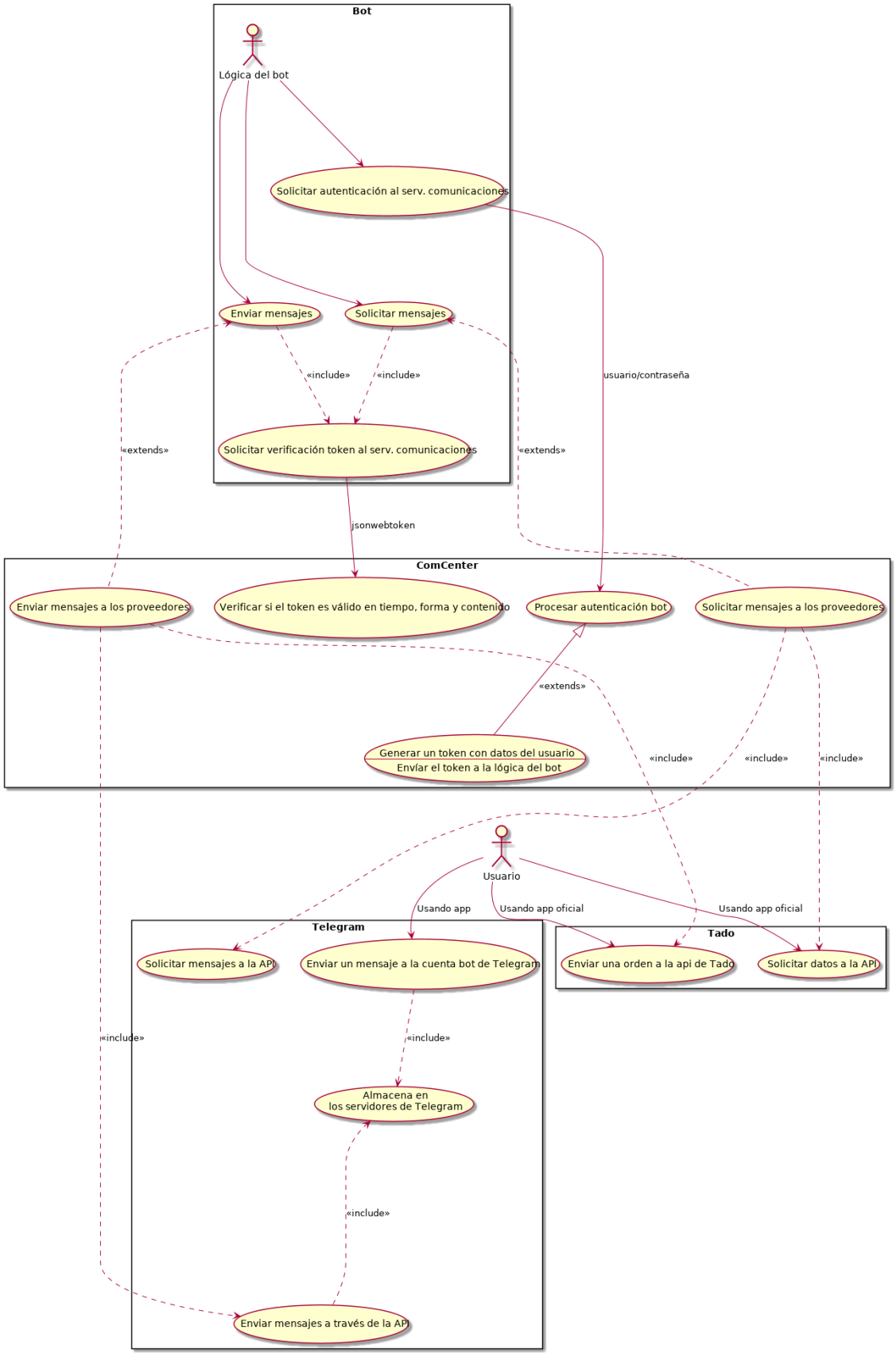

Fig. 2: Diagrama de casos de uso

4.3.1. 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.

4.3.2. 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 CA[6], 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 Tadoº. 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 3.

4.4. Estructura de base de datos

En la figura 4 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:

[6] Autoridad de certificación

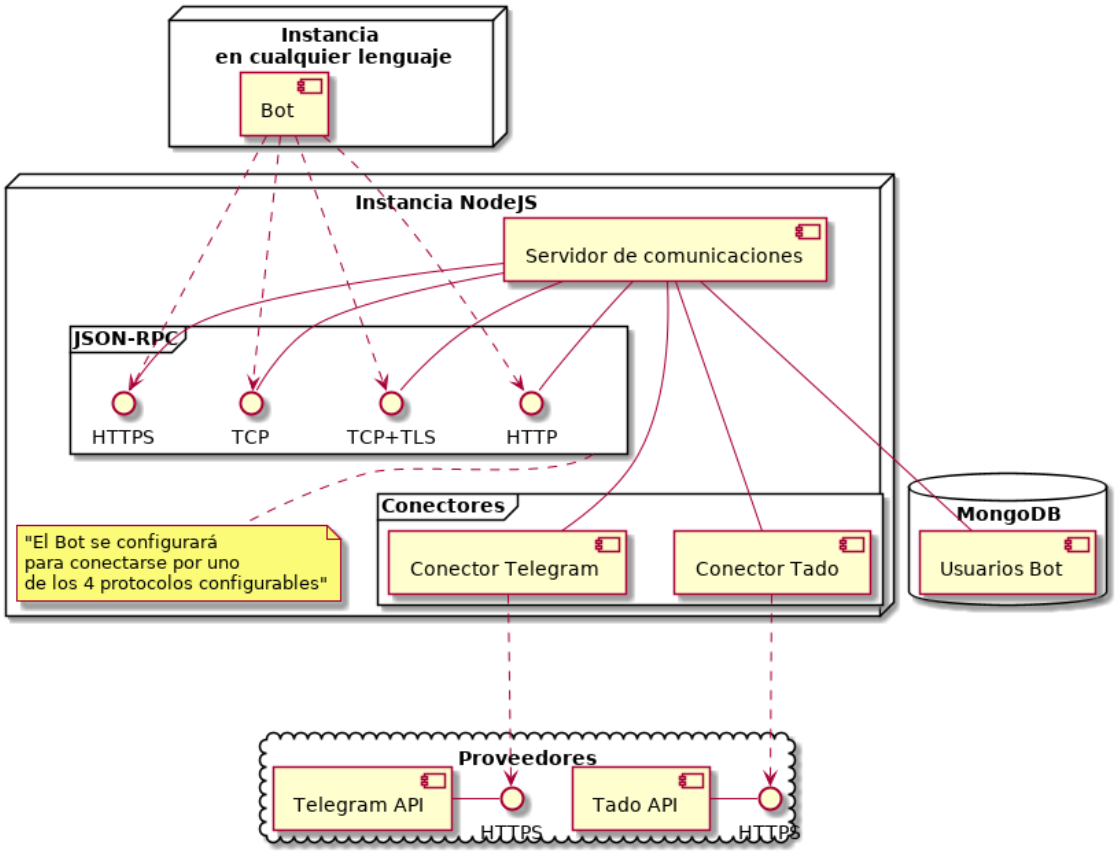

Fig. 3: Esquema de infraestructura (Diagrama de componentes)

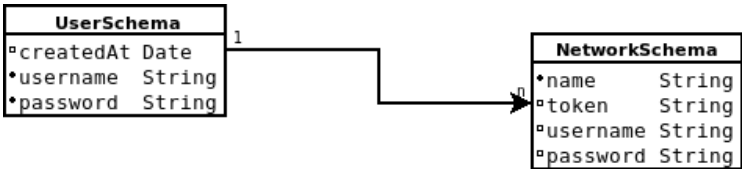

Fig. 4: Estructura de base de datos

- **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 el 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).

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

4.5. 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 5.

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 polimorfismo[7].

- **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 Tadoº.

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:

4.5.1. bfmb-comcenter

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

[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.]

- **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.
- **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.

4.5.2. 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.

4.5.3. 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 (Conection). Estos son clases hija de las clases Connector y Connection del módulo bfmb-base-connector.

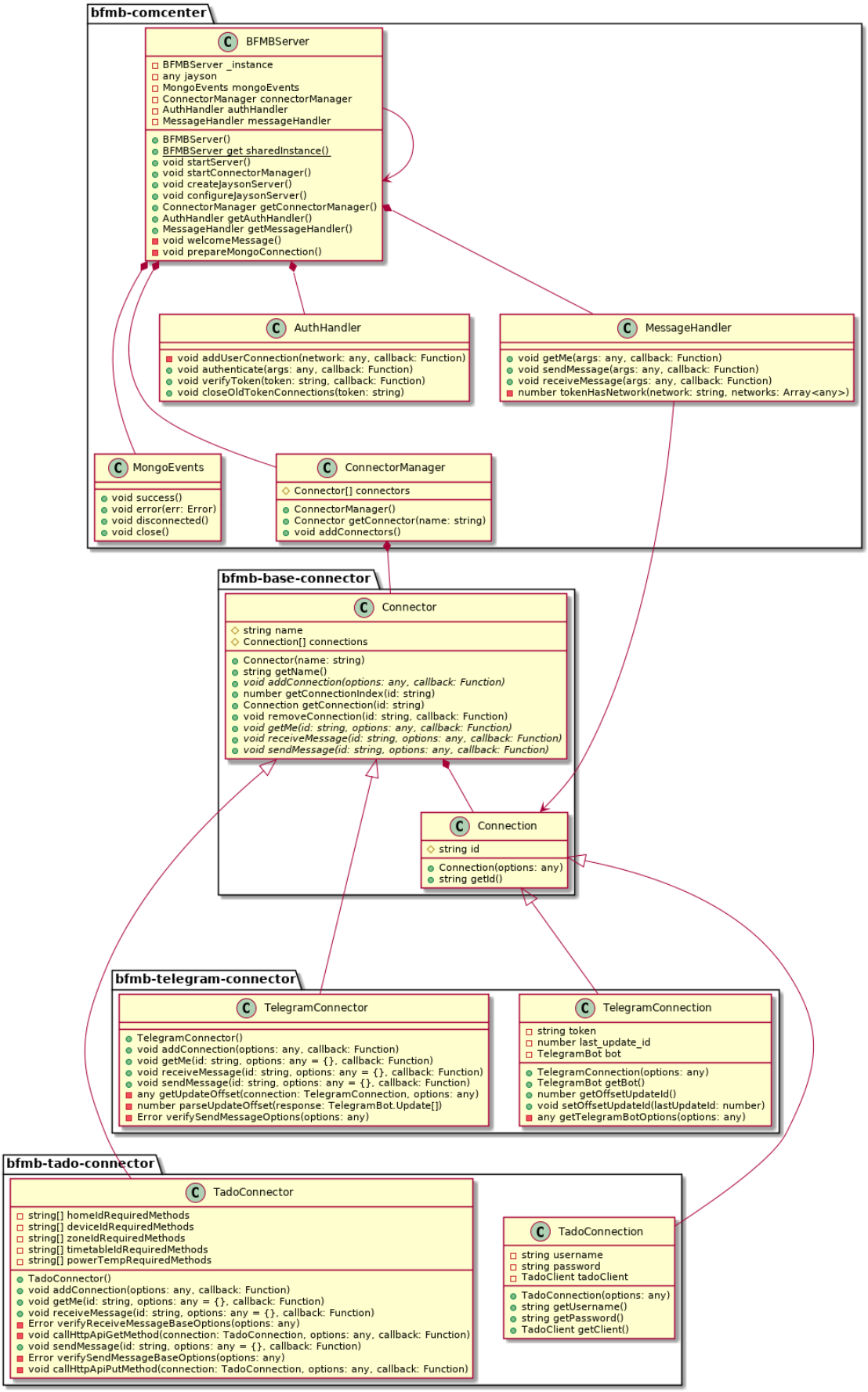

Fig. 5: Diagrama de clases

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.

5.2. Dependencias

5.2.1. MongoDB

MongoDB es un motor de base de datos orientado a documento. Se encuentra clasificado como una base de datos NoSQL, donde se usa una variante de JSON[8] como formato de documento.

5.2.2. NodeJS

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

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 podremos 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.

[8] Javascript Object Notation

5.2.4. Api de Telegram

5.2.5. Api de Tadoº

6. PRUEBAS

6.0.1. Mocha

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. Procedimento 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:

```
# Using Debian, as root
curl -sL https://deb.nodesource.com/setup_8.x | bash -
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

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

```
deb http://debian.redimadrid.es/debian/ buster main contrib non-free
deb-src http://debian.redimadrid.es/debian/ buster main

deb http://security.debian.org/debian-security buster/updates main
deb-src http://security.debian.org/debian-security buster/updates main

# Hay que añadir esta línea.
deb http://debian.redimadrid.es/debian/ unstable main contrib non-free
```

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

```
Package: *
Pin: release a=testing
Pin-Priority: 400

Package: *
Pin: release a=unstable
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.

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

```
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:

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

```
sudo apt install git
```

   Después, realizamos la clonación:

```
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.

```
cd ComCenter
npm install
```

3. Una vez realizado esto, ya tenemos instalado el servidor de comunicaciones. Actualmente, lo 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.

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

Ahora explicamos qué opciones están disponibles en la configuración:

```
{
  "db": {
    "url": "mongodb://localhost/comcenter",
    "encKey": "yfbmTM8/lZAH8H6PkGTcyK0H1bMk4kl0ovwq1xhX75w=",
    "signKey": "q8N6L+Lo345JsnUlSow4sPlKzo+
   XRJPYjT2Uq7mhlOkSi5kFkX0UoIY3etfm4UxtNHaM8xaX2HtkAhV7Gye0KA=="
  },
  "tokenConfig": {
    "secret": "W]5!>v$@NV!bgk8T?{$vP~o'^9P}S1",
    "algorithm": "HS512",
    "expiresIn": "24h"
  },
  "servers": [
    {
      "type": "tcp",
      "port": 3000
    },
    {
      "type": "http",
      "port": 3001
    },
    {
      "type": "https",
      "port": 3002,
      "cert": "/home/angel/certificados/certi.pem",
      "key": "/home/angel/certificados/certikey.key"
    },
    {
      "type": "tls",
      "port": 3003
    }
  ],
  "modules": [
```

```
    "telegram",
    "tado"
  ]
}
```

- **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:

```
openssl rand -base64 32; openssl rand -base64 64;

yZ4dEBn2RILEOGA04kg6oIJHsm6QO4kcbT2jnjAaFTo= # Clave de
    cifrado
7n24rUz8G8KKojMo3LhUBvv2TUBW7LQPg+
    FkyzFtAPSaF3NimFBPhNgWM9s4n5fX
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 json-webtoken para generar los tokens con firma. El formato es un string y acepta carácteres 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 carácteres 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 la 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 iniciarlo con este comando. Desde la carpeta raíz del servidor:

```
./bin/comcenter
```

Esta sería la salida esperada:

```
info: Using configuration file from: /home/angel/Proyectos/PFG/
    ComCenter/config.json
```

```
info: Welcome to BFMB ComCenter 0.5.0
info: Connecting to MongoDB...
info: Connected to MongoDB database
info: Attaching connectors to Connector Manager
info: Raising tcp server on port 3000
info: Raising http server on port 3001
info: Raising https server on port 3002
error: Can't raise https server. No certificates.
info: Raising tls server on port 3003
error: Can't raise tls server. No certificates.
node-telegram-bot-api deprecated Automatic enabling of cancellation of promises is deprecated.
In the future, you will have to enable it yourself.
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 ningúna aplicación de administrador por el momento, tengo hecho un script en lenguaje Javascript que añade los usuarios que queramos.

```
var mongoose = require('mongoose');
var config = require('../../config.json');
var User = require('../../dist/schemas/user').User;
var Network = require('../../dist/schemas/network').Network;

mongoose.connect(config.db.url, {useNewUrlParser: true});

if (!process.env.TG_TOKEN) throw new Error("No Telegram token defined.");
if (!process.env.TADO_USERNAME) throw new Error("No Tado login data defined.");
if (!process.env.TADO_PASSWORD) throw new Error("No Tado login data defined.");

var test = new User({
  username: 'TEST',
  password: 'test'
});

test.save(function (err) {
  if (err) throw err;

  var tgNetwork = new Network({
    name: 'Telegram',
    token: process.env.TG_TOKEN
  });
```

```
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 6, 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 usaré el cliente jayson.

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.

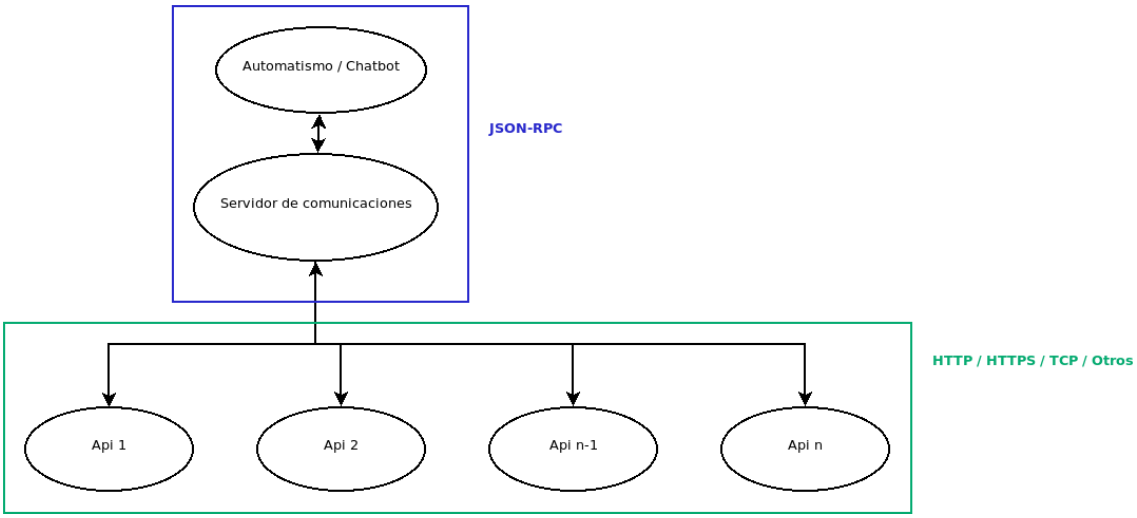

Fig. 6: Comunicación entre sistemas

```
this.jayson.request('authenticate',
 { username: 'username', password: 'password' },
 function(err: Error, response: any) {
      if (err) {
        // Caso error
      } else {
        // Caso éxito
      }
    });
```

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).

```
this.jayson.request('getMe',
 {token: 'token srv comunicaciones', network: 'Mired', options: {}},
 function(err: Error, response: any) {
   if (err) {
     // Caso error
   } else {
     // Caso éxito
   }
});
```

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.

```
this.jayson.request('receiveMessage',
 {token: 'token srv comunicaciones', network: 'Mired',
 options: {api_method: 'getZones', home_id: 'home_id'}},
 function(err: Error, response: any) {

 });
```

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 parámetreos *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.

```
this.jayson.request('sendMessage',
 {token: 'token srv comunicaciones', network: 'Mired',
 options: {api_method: 'setZoneOverlay', home_id: 'home_id'}},
 function(err: Error, response: any) {

 });
```

8. CONCLUSIONES

9. 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 usando en el servidor de comunicaciones.

  Actualmente se usa MongoDB por conveniencia en el desarrollo del proyecto final de grado (el uso de la biblioteca mongoose ayudó mucho), pero sería más apropiado usar bases de datos portables como SQLite 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.

  Otro motivo para abrir esta línea de cambiar la base de datos es porque, en base a informaciones del entorno open source / software libre, se comenta que MongoDB ha intentado modificar su licencia software y está generando rechazo en la comunidad, incluyendo que ciertas distribuciones (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.

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

  Ahora mismo, debido a problemas de tiempo de desarrollo, 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 23ºC” y lo asocie con la orden que debe ejecutar.

- 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*).

ANEXOS

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]