UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Interconexión del robot NAO a servicios de simulación en la nube

TRABAJO FIN DE MÁSTER

Máster Universitario en Automática e Informática Industrial

Autor: Carmona Vila, Jahel

Tutor: Blanes Noguera, Juan Francisco

Curso 2018-2019

Resum

En aquest projecte es desenvolupen serveis en el núvol per tal de desacoblar de dins del robot NAO el codi que executa el simulador de diabetes que duu implementat. Es suprimeixen les connexions TCP per protocols de capes superiors en la pila TCP/IP com son HTTP i MQTT que permeten una comunicació més flexible i amb més opcions. A més, es modernitza el codi de manera que sigue més senzilla tant la seva estructura com la seva compilació i execució.

Paraules clau: NAO, simulador, diabetes, HTTP, MQTT

Resumen

En este proyecto se desarrollan servicios en la nube para desacoplar de dentro del robot NAO el código que ejecuta el simulador de diabetes que lleva implementado. Se suprimen las conexiones TCP por protocolos de capas superiores en la pila TCP/IP como son HTTP y MQTT que permiten una comunicación más flexible y con más opciones. Además, se moderniza el código para que sea más sencilla tanto su estructura como su compilación y ejecución.

Palabras clave: NAO, simulador, diabetes, HTTP, MQTT

Abstract

In this project are developed cloud services to disengage the code inside the NAO Robot that starts up the diabetes simulator. TCP connectios are suppressed by higher layer protocols in the TCP/IP stack as HTTP and MQTT are which allow a more flexible communication and with more options. Besides the code is modernized so that this can be simplier in terms of structure, compilation and execution.

Key words: NAO, simulator, diabetes, HTTP, MQTT

Índice general

Índice de figuras

Índice de tablas

CAPÍTULO 1
Introducción

1.1 Motivación

Este proyecto desarrollado parte directamente de «*Desarrollo de un sistema de monitorización y control de un robot simulador de diabetes*», un trabajo anterior a éste, el cual se encuentra en Riunet [3], elaborado por Antonio Bengochea Carrasco y dirigido también por Francisco José Blanes Noguera. Lo que se trata en el actual proyecto es de mejorar la infraestructura anterior teniendo en cuenta las limitaciones sobre las prestaciones que existen hoy en día para el proyecto previo. El centro del proyecto se encuentra dentro del robot NAO, por lo que es el que limita el crecimiento de la aplicación.

NAO es un robot humanoide programable y autónomo, desarrollado por Aldebaran Robotics, una empresa francesa con sede en París subsidiaria del grupo Softbank. Estos robots son ampliamente utilizados en por ejemplo la Robocup, un concurso de robótica a nivel internacional.

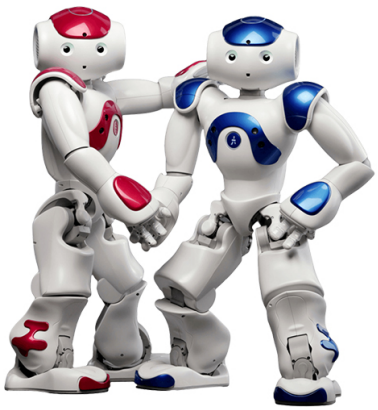

Figura 1.1: Robot NAO en varios colores

En esta hoja técnica [2] se detallan las características técnicas del robot que es objeto de estudio. Con unas dimensiones de 574x275x311mm y un peso de 5.4 kg, el robot consigue su movilidad mediante motores de corriente continua sin escobillas, siendo un total de 26 motores. En la imagen 1.2 podemos ver su esquema

cinemático, en el cual se ven los ángulos de rotación y traslaciones.

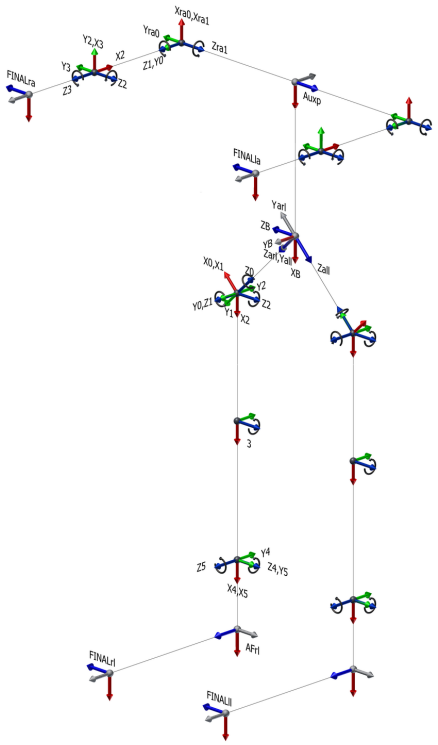

Figura 1.2: Esquema cinemático del robot NAO.

Como se ha mencionado antes, este proyecto parte de un proyecto anterior, cuyas limitaciones del propio robot suponen a su vez límites en el crecimiento de los procesos que lleva implementados. Precisamente, el componente que ejerce de tope es la unidad de cómputo y sus elementos asociados, como las memorias, o lo que el fabricante denomina *Motherboard*. El límite es debido a que el robot incluye un simulador de diabetes que emplean librerías científicas. Esto supone un uso exhaustivo de los recursos de cómputo del robot, siendo que además del propio simulador, también corren otros procesos e hilos. Las características del robot para este aspecto podemos verlas en la tabla 1.1:

Elemento	**Subelemento**	**Prestaciones**
CPU	CPU Processor	Intel ATOM Z530
	Cache memory	512KB
	Clock speed	1.6GHz
	FSB speed	533MHz
RAM		1GB
Flash memory		2GB
Micro SDHC		8GB

Tabla 1.1: Motherboard del robot NAO

Para las aplicaciones de hoy en día, 1GB en memoria principal (RAM) es demasiado poco. Además teniendo en cuenta que aparte de ejecutar el código, que

es muy cargante como se ha mencionado, también ejecuta otros procesos como por ejemplo la comunicación con sus actuadores y sensores mediante *proxyes* como denomina el fabricante, o por ejemplo la comunicación via HTTP, SSH, entre otros. Por otra parte, el procesador Intel ATOM tienen un rendimiento bajo, como se muestra en a figura 1.3. No aparece el modelo de NAO, Atom Z530, pero el rendimiento es similar a los modelos que se muestran en la imagen.

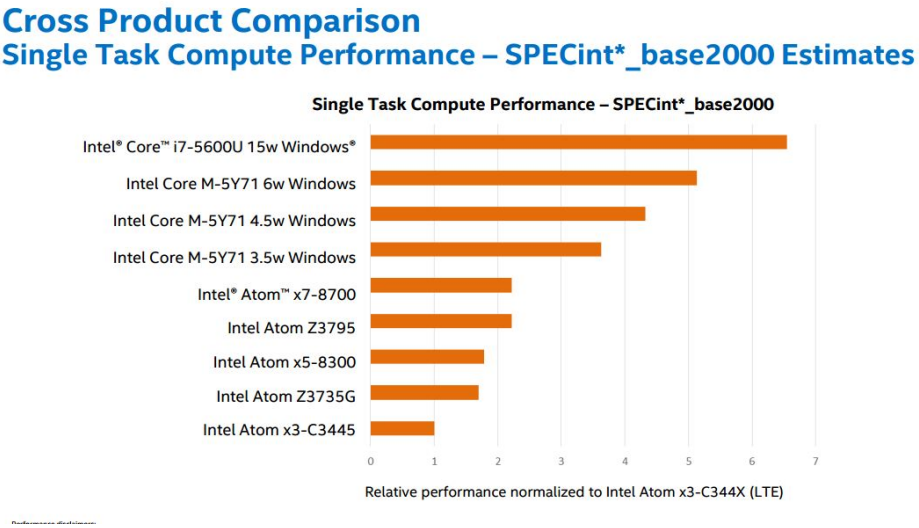

Figura 1.3: Comparativa de rendimiento de procesadores Intel.

Es importante entender que el modelo de comportamiento del páncreas que se sigue para simular la diabetes es el modelo **Hovorka**, cuya implementación puede encontrarse en sus documentos de investigación publicados, como por ejemplo **TODO: CITAR BIEN EL MODELO DE PANCREAS**. Su resolución pasa por ecuaciones diferenciales que se ejecutan periódicamente, lo cual resulta muy pesado para una máquina de las características que se han observado.

1.2 Repaso del proyecto anterior

El proyecto anterior estaba dividido en dos partes. La primera parte es la que actúa de **parte servidora** y se encuentra en el código del robot NAO. Mediante conexiones TCP, atiende peticiones de un cliente Java para obtener datos de la glucosa o para recibir órdenes. Contiene dentro cuatro hilos de ejecución, siendo el primero el hilo principal, que incluye un gestionador de hilos; el segundo el hilo que atiende las conexiones TCP; el tercero aloja el código del simulador, y por último, hay dos hilos excluyentes, es decir, que solamente puede haber uno activo cada vez. Estos son un hilo de escenario, que ejecuta el comportamiento de una persona con diabetes, atendiendo al estado de glucosa que tiene en un mismo instante y con funciones de inyectarse insulina. El otro hilo es un hilo de interacción, en el que solamente recibe órdenes sencillas y responde a ellas.

La otra parte del proyecto anterior es un **cliente Java**, el cual en el presente proyecto se va a dejar prácticamente intacto, pues solo va a sustituirse la parte del

cliente TCP. Por lo demás, la interfaz solicita al servidor datos de glucosa actual al simulador de diabetes y envía órdenes al robot, que pueden ser acciones sobre los hilos o bien órdenes para que el robot ejecute movimientos o diga palabras.

1.3 Objetivo

El objetivo de este proyecto es **desengarzar la parte pesada del código del anterior proyecto para virtualizarla en un servicio en la nube**. Esta parte es el simulador de diabetes, que implementa ecuaciones muy costosas computacionalmente y emplea librerías científicas.

Esto implica una transformación total de la parte del código de NAO, que significa actualizar las partes del proyecto «*Desarrollo de un sistema de monitorización y control de un robot simulador de diabetes*». Se ha eliminado completamente TCP de la estructura del proyecto y se ha sustituido por MQTT o HTTP (REST), según necesidad. Las partes del proyecto por lo tanto necesitan estas modificaciones:

- Aplicación Java: En vez de utilizar una conexión TCP para conectarse al robot, deberá utilizar librerías para hacer llamadas HTTP y un cliente MQTT.
- La parte de NAO: Sufre un cambio radical ya que se debe cambiar de C++ a Python 2.7 **TODO: RELACIONAR CON LA PARTE QUE LO JUSTIFICO**(la versión que admite NAO), y se divide el proyecto en dos partes:

  – Google Cloud Engine: Se debe alquilar un servicio en la nube, en este caso del proveedor Google, y se montará un servidor que atenderá peticiones HTTP. Las responderá con el estado de la glucosa actualmente, ya que contendrá el simulador de diabetes.

  – Código en NAO: Se debe eliminar toda la parte del simulador y TCP. Se añadirán dos librerías de HTTP y un cliente MQTT.

1.4 Estructura de la memoria

Este proyecto se va a dividir en cuatro partes, que corresponden a las tres partes que conforman el proyecto y una cuarta sobre cómo se relacionan entre ellas. Las tres partes son las mencionadas en la sección anterior: Cliente Java, servicio en la nube y código en NAO.

Finalmente, se valorará el resultado del proyecto mediante unas conclusiones y se listarán una serie de posibles mejoras al proyecto actual. También hay un apéndice con información técnica acerca de cómo configurar los diferentes servicios.

CAPÍTULO 2
Servicio en la nube

2.1 Servicios en la nube actualmente

Hoy en día existen empresas que ofrecen lo que se llama *Cloud computing*, o lo que es lo mismo, computación en la nube o servicios en la nube, es un paradigma que permite ofrecer servicios de computación a través de una red, que usualmente es Internet.

La computación en la nube físicamente son servidores encargados de atender las peticiones recibidas en cualquier momento. Sirven a sus usuarios desde varios proveedores de alojamiento repartidos por todo el mundo. Este comportamiento está esquematizado en la figura 2.1. Esta medida reduce los costos, garantiza un mejor tiempo de actividad y que los sitios web sean mucho más seguros.

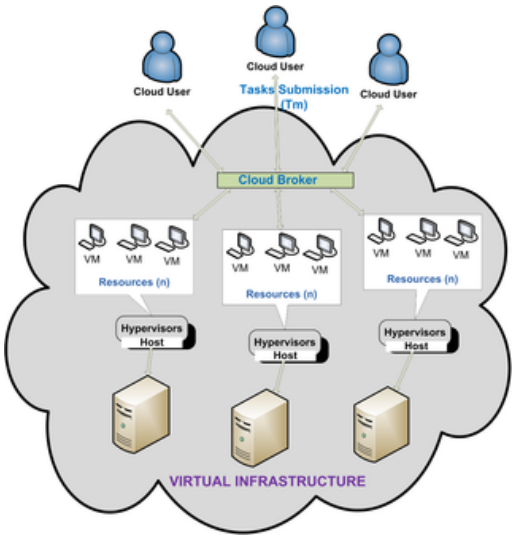

Figura 2.1: Esquema de servicios en la nube.

Los servicios en la nube pueden ofrecer cierto nivel de servicios. Esto permite que, si se ofrece un nivel alto de servicio, el programador se ahorra tener que desarrollar la aplicación, pero la parte personalizable queda reducida. Por contra, si se contrata un nivel de servicio bajo, el programador deberá invertir tiempo en

desarrollar la aplicación sobre el servicio y será mucho más personalizable. Estos tipos de nivel de servicio son:

- **Iaas (Infraestructure as a Service)**: Permite gestionar al desarrollador aspectos como el sistema operativo, qué servidores quiere instalar en su máquina... Pero los aspectos hardware se hacen transparentes al programador. Un ejemplo de esto es el Google Compute Engine, ya que permite crear máquinas virtuales en las que se puede ejecutar cualquier tipo de software dentro.
- **Paas (Platform as a Service)**: Está un nivel por encima de IaaS, porque se hacen transparentes al programador, además de lo anterior, aspectos de sistema operativo, servicios de una máquina virtual... De manera que el desarrollador solamente debe rellenar algunas secciones de código. Un ejemplo de PaaS sería Google App Engine, ya que es una plataforma que permite redireccionar mediante peticiones REST aquellas peticiones a los fragmentos de código que ha escrito el programador.
- **Saas (Software as a Service)**: Cualquier servicio en la web listo para ser usado tal como está. Estos son los que los usuarios conocen más, porque son por ejemplo Google Drive, entre otros.

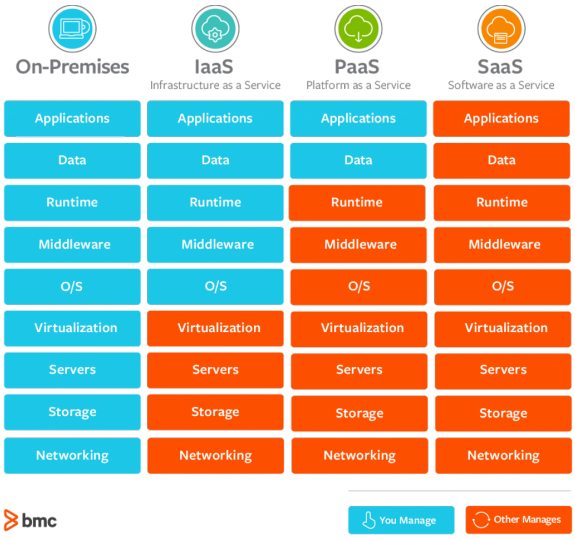

Figura 2.2: Niveles de servicio: IaaS, SaaS y PaaS.

En la figura 2.2 se aprecian las distintas capas que cubre cada uno de los niveles de servicio. Aparece uno no nombrado hasta ahora que es *On-premises*; básicamente, se refiere a cómo serían los servicios localmente.

Hoy en día los mayores proveedores de servicios en la nube que existen son Amazon Web Services, IBM, Oracle, Alibaba, Google Cloud Services y Microsoft Azure. De las compañías mencionadas, las más grande y empleada en el mundo empresarial es Amazon Web Services, pero en este proyecto no se ha seleccionado debido a que la complejidad de su nube es muy alta, ya que ofrecen un total de 2400 aplicaciones como se indica en esta tabla [4]. Se ha escogido una más sencilla como es **Google Cloud Services**, que ocupa el tercer lugar y con una curva de

aprendizaje mucho más rápida que sus anteriores. Además, el plan económico de Google Cloud Services es mucho más conveniente que otros.

2.2 Google Cloud Services

De la gama que ofrece Google Cloud Services, de la cual podemos ver una muestra en la imagen 2.3 se han estado contemplando varias opciones. En un primer lugar, se estuvo barajando trabajar con Google App Engine, que como se ha nombrado antes, es un PaaS. Así, el trabajo a realizar se consideraría notablemente. Pero tuvo que descartarse debido a que ciertas partes como la concurrencia de hilos no eran compatibles con el diseño del PaaS. Se estuvieron mirando otros productos para complementarlo, como son Google Tasks o Google Cloud Storage, pero la combinación no podían generar un servicio de simulación de diabetes en la nube, como era el propósito. Para tener más libertad, se ha escogido para este proyecto un **IaaS**, **Google Compute Engine**, porque tiene mucha flexibilidad para programar los servicios que se deseen.

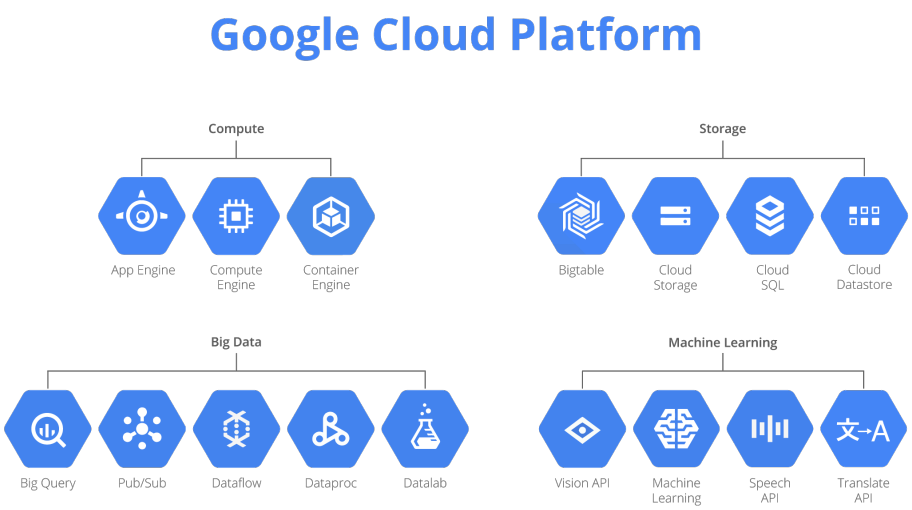

Figura 2.3: Productos de Google Cloud.

Google Compute Engine, o GCE es básicamente una infraestructura que permite levantar **máquinas virtuales** de forma escalable a las cuales se puede acceder via SSH, API, y más formas. Esta máquina virtual, a la hora de crearla, deja escoger qué tipo de memoria principal tiene, el sistema operativo y el número de CPUs (*Central Processing Unit*) y TPUs (*Tensor Processing Unit*); este último componente se emplea para los mecanismos de aprendizaje automático, que en este caso no aplica.

Como se menciona en la tabla 2.1, en la cual podemos ver las características de la máquina virtual empleada para contener el simulador de diabetes, la IP es necesario que sea estática; se necesita que tanto el robot como la aplicación Java

sepan dónde se encuentra el servicio en la nube en todo momento.

Característica	Descripción
Sistema operativo	Ubuntu 19.04
Memoria principal	3,75 GB
Memoria del disco	10 GB
CPU	1 vCPU
IP	Estática (34.76.240.69)

Tabla 2.1: Características de la máquina virtual de GCE

Se han levantado dos servicios dentro de la máquina virtual, para cada uno de los cuales debe entrarse por un puerto diferente. El primero es un *broker* MQTT, el cual se explica en la sección y el segundo el propio servidor para el simulador de diabetes.

2.3 Servidor de diabetes

Del proyecto anterior, se ha quitado la parte del simulador de diabetes y se ha movido a la máquina virtual de Google Compute Engine, como se ha comentado en otras secciones. Pero para acceder a él y poder manejarlo, antes debe superponerse una capa (API, *Application Programming Interface*) para el manejo de peticiones entrantes. En este caso se ha decidido utilizar el modelo REST sobre un servidor HTTP.

2.3.1. HTTP

HTTP, o *HyperText Transfer Protocol*, es el protocolo de red que permite la transferencia de documentos de hipermedia en la red, generalmente entre un navegador y un servidor aunque en este proyecto sea a través de una aplicación Python y Java, para que los humanos puedan leerlos.

Sobre las URLs de los servicios web se pueden hacer muchos tipos de operaciones, que son exactamente: GET, HEAD, POST, PUT, DELETE, TRACE, OPTIONS, CONNECT, PATCH, SEARCH, COPY, LOCK, UNLOCK, MOVE, MKCOL, PROPFIND, PROPPATCH, MERGE, UPDATE y LABEL. Se aprecia que los nombres son cuasi descriptivos por si solos.

Las peticiones y las respuestas de peticiones llevan una cabecera con metadatos para indicar por ejemplo si lo que va a recibirse está en formato XML, JSON... (Se llama MimeType), o el lenguaje de la respuesta, entre otras opciones de metadatos. Después de las cabeceras viene el cuerpo del mensaje en sí, que contiene la información solicitada o de petición.

También está en el protocolo el código de las respuestas. Según el código recibido quieren decir una cosa u otra, pero se agrupan en cinco tipos de formato de código:

- Formato 1xx: Respuestas informativas. Indica que la petición ha sido recibida y se está procesando.
- Formato 2xx: Respuestas correctas. Es el resultado deseado.
- Formato 3xx: Respuestas de redirección. Indica que el cliente necesita realizar más acciones para finalizar la petición.
- Formato 4xx: Errores causados por el cliente. Indica que ha habido un error en el procesado de la petición a causa de que el cliente ha hecho algo mal.
- Formato 5xx: Errores causados por el servidor. Indica que ha habido un error en el procesado de la petición a causa de un fallo en el servidor.

2.3.2. REST

Como se menciona en este artículo [10], REST fue presentado por primera vez en el 2000 por Roy Fielding en la Universidad de California. REST (*Representational State Transfer*) es una arquitectura de programación pensada para servicios web, que se transfieren mediante HTTP. Ha desplazado enormemente a SOAP gracias a su sencillez de uso y de entendimiento. Sigue cuatro principios de diseño:

- Utilice métodos HTTP de forma explícita: El cliente interactúa con el servidor únicamente con métodos HTTP mediante operaciones de crear, leer, actualizar y borrar, que para REST son:
  - Para crear un recurso en el servidor hay que utilizar un POST.
  - Para consultar un recurso hay que utilizar un GET.
  - Para cambiar el estado de un recurso hay que utilizar un PUT.
  - Para borrar un recurso hay que utilizar un DELETE.
- Sea sin estados: Un mensaje no tiene porqué depender de un mensaje anterior para que se entienda, sino que cada mensaje debe tener toda la información necesaria.
- Exponga los URIs (*Uniform Resource Identifier*) como estructuras de directorios.
- Transfiera XML, JavaScript Object Notation (JSON), o ambos.

2.3.3. HTTP y REST en este proyecto

Se ha decidido emplear HTTP y REST en este proyecto porque como se mencionaba antes, su uso es el más extendido hoy en día debido a su sencillez.

Como se aprecia, HTTP y REST por si solos no pueden entenderse, pues HTTP necesita guiarse por el patrón que marca REST y REST necesita que HTTP de cuerpo a su arquitectura. Así que se explican de manera conjunta.

Se ha seguido la estructura de las URIs como REST manda que se monte, es decir, como una estructura de directorios. Cuando se compone toda la URL, y se implementan los métodos, los puntos de acceso al servidor y sus respuestas son los que se indican en las tablas 2.2 y 2.3. En esta tabla se ha recortado la URL y se ha dejado solamente la ruta hasta el recurso. La URL absoluta tiene el prefijo `http://34.76.240.69:8080`, que por ejemplo para acceder al recurso raíz (/) será `http://34.76.240.69:8080/`, o para acceder a /Hilo/, `http://34.76.240.69:8080/Hilo/`. Nótese que solamente se han empleado los métodos HTTP GET y PUT, pues solo se permiten operaciones de lectura y de actualización de recursos.

URL	GET	Respuesta
/	Sí	Es para probar la conectividad, devuelve "Hello World".
/Hilo/	Sí	Devuelve el estado del hilo: PARADO, PAUSADO, CORRIENDO.
/Simulador/Modo/	Sí	Devuelve el modo actual del simulador.
/Simulador/Glucosa/	Sí	Devuelve la glucosa actual del simulador de diabetes.
/Simulador/ DatosSimulacion/	Sí	Devuelve, si lo tiene, el dato de simulación que se ha enviado.

Tabla 2.2: Estructura de URIs y descripción de método HTTP GET sobre el servidor de diabetes.

URL	PUT	Respuesta
/	No	-
/Hilo/	Sí, acepta como parámetros PARADO, PAUSADO, CORRIENDO. sino respuesta 4xx.	Si se ha podido aplicar la operación al hilo, devuelve 200 OK,
/Simulador/Modo/	Sí, admite los valores 1,2,3	Si todo ha ido bien, 200 OK, sino 4xx.
/Simulador/Glucosa/	No	-
/Simulador/ DatosSimulacion/	Sí, admite un JSON con los datos para una nueva simulación.	Si todo ha ido bien, 200 OK, sino 4xx.

Tabla 2.3: Estructura de URIs y descripción de método HTTP PUT sobre el servidor de diabetes.

Para hacer real esta estructura, esta *API*, hay que programarlo en alguna tecnología. Se escoge **Python 2.7**, para mantener la coherencia con el resto del proyecto, el código en NAO, que está obligado a ser en Python 2.7 también, como

se explica más adelante. Antes estaba escrito en C++, pero existen utilidades más modernas y moldeables en Python, por lo que se aprovecha el gran cambio que va a sufrir el simulador para convertirlo a un lenguaje más flexible.

Primero, se utiliza un **servidor WSGI en el puerto 8080** para que la aplicación Python escuche las peticiones entrantes. WSGI significa *Web Server Gateway Interface* y es el conjunto de servidores web que reenvían las peticiones a las aplicaciones web escritas en Python. Tienen dos partes: la primera la parte del servidor en si, que en este caso está construida sobre Nginx, y la segunda, que es la implementación de la API REST en sí. De las posibles opciones que existen en Python, se ha escogido **gevent**, cuyo sitio oficial se encuentra en [9]. Crear una aplicación web con WSGI es muy sencillo y más aún si se combina con Flask.

Flask es un *web framework*, lo que significa un conjunto de herramientas y librerías que hacen muy simple estructurar una aplicación web. Su mayor competidor es Django, pero es más recomendable utilizar Flask junto con Python, por no mencionar que es mucho más simple utilizar Flask. Dado que la API construida no tiene más que un dos clientes, tampoco es necesario algo muy potente, así que la simpleza se valora más. Añadido a Flask, hay una extensión llamada **Flask RESTful** [12] que abstrae un método Flask convencional para convertirlo en una serie de operaciones sobre un recurso de la jerarquía de URIs.

Juntando la máquina de Google Compute Engine y las tecnologías para montar la API REST –WSGI y Flask–, se consigue hacer la puerta de entrada para poder interactuar con el simulador, el cual está encapsulado en un hilo de ejecución. Mediante la API REST se puede activar, pausar, detener o reanudar la ejecución del hilo de ejecución del simulador, así como consultar la glucosa actual en cada momento o enviar datos para simular. Pero lo que falta por convertir es la librería que hace el cálculo de las ecuaciones diferenciales.

2.4 MQTT

2.4.1. Introducción a MQTT

MQTT significa *Message Queue Telemetry Transport*. Es un protocolo del tipo publicador/suscriptor, extremadamente simple y ligero, diseñado para dispositivos limitados o con poco ancho de banda. Los principios de diseño son minimizar el ancho de banda de la red y los requisitos de recursos del dispositivo al mismo tiempo que se intenta garantizar la confiabilidad y cierto grado de garantía de entrega. Estos principios también hacen que el protocolo sea ideal para el emergente mundo de dispositivos conectados «máquina a máquina» (M2M) o «Internet de las cosas» (IoT), y para aplicaciones móviles donde el ancho de banda y la energía de la batería son muy importantes. Además, la versión 3.1.1 se encuentra certificada en la ISO/IEC PRF 20922 [6].

En MQTT hay dos tipos de agentes que intervienen:

- El ***broker***: Es el servidor que contiene las colas de mensajes. Los clientes depositan mensajes en las colas (*publican* estos mensajes) de forma que otros clientes reciben estos mensajes encolados. Solo existe un broker por cada sistema MQTT.
- **Clientes**: Puede haber tantos clientes como se desee. Son los que generan los mensajes que envían al broker, y el broker se encarga de distribuirlos a los clientes que pertoque.

El hecho de enviar un mensaje al broker se llama **publicar**, y para que un cliente pueda recibir los mensajes, debe **estar suscrito** a ellos. La operación de recepción de un mensaje es asíncrona, pues se reciben los mensajes y se ejecuta una llamada como consecuencia (*callback*). Para identificar a qué suscriptores pertenece cada mensaje, se emplean unas etiquetas, que pueden contener niveles de jerarquía, al estilo de los directorios en sistemas operativos. Por ejemplo, si se publica un mensaje *topic1/topic2*, y un suscriptor indica que desea recibir los mensajes sobre esa etiqueta, así se hará. Pero hay dos carácteres especiales. Supóngase la etiqueta de ejemplo anterior:

- **#**: Si un cliente decide suscribirse a *topic1/#*, recibirá todos los mensajes cuya etiqueta comience por *topic1/*.
- **+**: Si algún suscriptor decide suscribirse a *+/topic2*, recibirá todos aquellos mensajes que comiencen por cualquier etiqueta en el primer nivel, pero que en el segundo nivel se llamen *topic2*.

Puede observarse un ejemplo gráficamente en la figura 2.4.

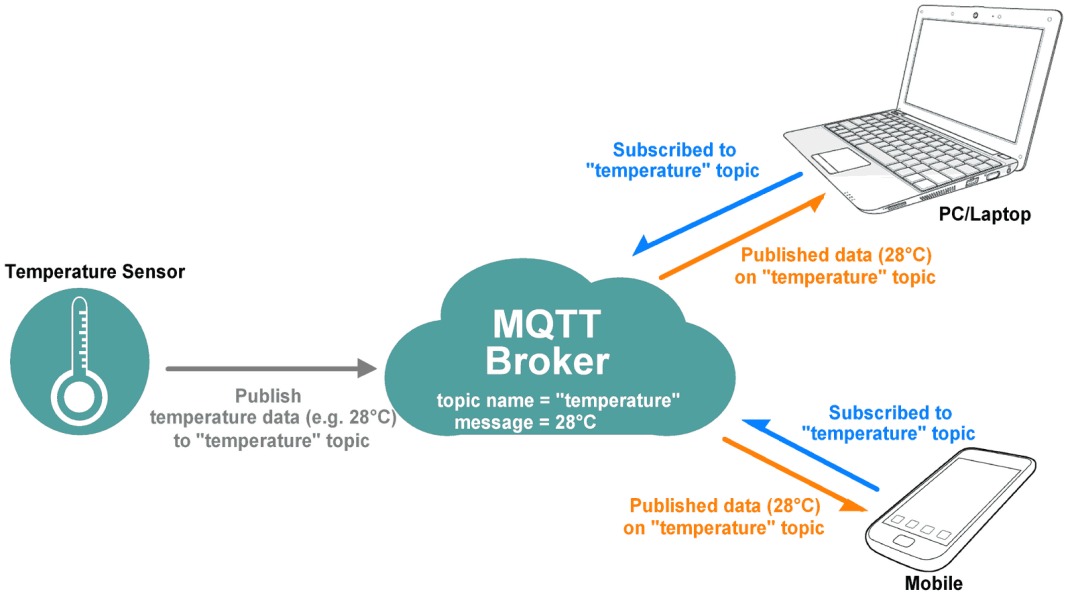

Figura 2.4: Ejemplo del funcionamiento de MQTT.

Existen muchas implementaciones de MQTT: Paho MQTT, Mosquitto, HiveMQ, Adafruit IO, las cuales son las más famosas, pero hay más. Muchas de ellas son de código abierto, otras no, están desarrolladas por diversas empresas o universidades y demás características que se pueden observar en este enlace [7].

2.4.2. MQTT en este proyecto

Para este proyecto se han empleado dos tipos de MQTT del mismo proveedor, Eclipse, que ha desarrollado M2Mqtt, Mosquitto y Paho MQTT. El primero, **Mosquitto**, se ha utilizado para instalar el **broker** en la máquina virtual. Dado que Mosquitto es la única de las tres implementaciones de Eclipse que tiene un broker, no había más opción que instalarlo. Para cliente, solamente tiene integración con el lenguaje de programación C, así que se desarrollan los **clientes** con **Paho MQTT**, que es el segundo componente, pues tiene un nivel de compatibilidad muy alto – se adapta a C, C++, Java, JavaScript, Python y Go.

Dado que el broker Mosquitto se implementa en la máquina virtual, que lleva Ubuntu 19.04 como sistema operativo, la fuente de instalación es los propios repositorios de Ubuntu. Se instala sobre el puerto **TCP 1883** y se emplea autenticación de usuario y contraseña. La conexión es no-TLS (*Transport Layer Security*), pero si se quisiera una conexión cifrada (TLS) debería instalarse en el puerto 8883. Para activar el servidor no hay que hacer más que activar un servicio con *systemctl*.

Los clientes MQTT están tanto en el código NAO como en la aplicación Java, y ambos tienen detallados su funcionamiento en sus secciones pertinentes.

La intención para este sistema es sustituir la conexión TCP que enviaba mensajes periódicamente por una conexión asíncrona como es ésta, mucho más ligera a nivel computacional. Se envían mensajes para el arranque de hilos, órdenes directas al robot, mensajes de estado, entre otros.

2.5 Librería para el cálculo diferencial

Lo último necesario para terminar de migrar el simulador de diabetes a una aplicación web Python es adaptar la librería de cálculo diferencial. Tal como se menciona en la introducción de este proyecto, se sigue un modelo de simulación según Hovorka que emplea ecuaciones ODE (ecuaciones diferenciales ordinarias). En el simulador del anterior proyecto se emplea la librería Alglib, hecha para C++. Al convertir la aplicación a Python, es necesario cambiar también la librería de cálculo.

Alglib [13] tiene una licencia comercial y otra gratuita, esta última para fines no comerciales como es esta aplicación. Empleando la opción gratuita, existe un *wrapper* o una aplicación que engloba a la librería construida para C++, para que pueda utilizarse en Python. En un principio se había pensado en emplear esta opción para respetar los cálculos iniciales pero supone una limitación a la hora de actualizar la librería de la aplicación Python. Esta librería para actualizarse depende de que se actualice su código interno C++ y luego el *wrapper* para ésta.

Por lo tanto se ha migrado la librería de cálculo a **SciPy** [14], una librería para cálculo matemático entre otras funciones, cuyo uso está muy extendido. De hecho, de sus paquetes centrales, existen unos que son muy famosos como NumPy, IPython o Matplotlib.

CAPÍTULO 3
Código en NAO

Idees de estructura: - Explicació del NAO i tipos de llenguatge - Explicació del projecte anterior i arquitectura (dibuixets de TCP) - Explicació de les limitacions de còmput de NAO i de limitacions de TCP - Estructura del meu projecte. MQTT entrar en detall del QoS. (Part Java, Part NAO i part GCloud). - Part Java: Mínim canvi realment. Explicar Paho i que es un client Rest. - Part NAO: Que he reconvertit el codi a Python perquè el NAO te molt mal el C++. - Part GCloud: Aquí explayar-se. - Millores: Prescindir de GCloud i montarse una raspi amb un DNS, afegir més funcions al NAO, mes precisio del simulador aprofitant gcloud, manejar varios NAOs amb una sola app java i un sol gcloud...

CAPÍTULO 4
Aplicación Java

????? ????????????? ????????????? ????????????? ????????????? ?????????????

CAPÍTULO 5
Relación entre los componentes

????? ????????????? ????????????? ????????????? ????????????? ?????????????

5.1 ?? ???? ???? ? ?? ??

????? ????????????? ????????????? ????????????? ????????????? ?????????????

CAPÍTULO 6
Conclusions

6.1 Mejoras

Poder arrancar la MV desde API
Hacer la app Java una app web en el GCE
Controlar mas de un robot con su simulador asignado a la vez

6.2 Conclusiones

??????????????????????????????????????

Bibliografía

[1] Breve explicación de qué es el robot NAO.
Consultado en:
https://es.wikipedia.org/wiki/Nao_(robot)

[2] Características técnicas del Robot NAO.
Consultado en:
https://static1.squarespace.com/static/52e733d8e4b062fc0c603ea8/t/53a13620e4b0d80b1acadb20/1403074080427/nao_datasheet.pdf

[3] Desarrollo de un sistema de monitorización y control de un robot simulador de diabetes. Autor: Antonio Bengochea Carrasco.
Consultado en:
https://riunet.upv.es/bitstream/handle/10251/94065/BENGOCHEA%20-%20Desarrollo%20de%20un%20sistema%20de%20monitorizaci%C3%B3n%20y%20control%20de%20un%20robot%20simulador%20de%20diabetes.pdf?sequence=3

[4] Cuadro comparativo con las empresas de Cloud Computing
Consultado en:
https://recursos.apser.es/hubfs/Infograf%C3%ADas/APS-Cuadro-comparativo-grandes-cloud.pdf

[5] Información sobre MQTT
Consultado en:
http://mqtt.org/

[6] ISO sobre MQTT v3.1.1
Consultado en:
https://www.iso.org/standard/69466.html

[7] Comparación de las implementaciones de MQTT
Consultado en:
https://en.wikipedia.org/wiki/Comparison_of_MQTT_implementations

[8] Principios de HTTP
Consultado en:
https://es.wikipedia.org/wiki/Protocolo_de_transferencia_de_hipertexto

[9] WSGI de Gevent
Consultado en:
http://www.gevent.org/

[10] Principios de servicios RESTful
Consultado en:
https://www.ibm.com/developerworks/ssa/library/ws-restful/index.html

[11] ¿Por qué es Flask una buena elección para web?
Consultado en:
https://www.fullstackpython.com/flask.html

[12] Manual de la extensión de Flask para REST
Consultado en:
https://flask-restful.readthedocs.io/en/latest/

[13] Página principal de Alglib
Consultado en:
http://www.alglib.net/

[14] Página principal de la librería SciPy
Consultado en:
https://www.scipy.org/

APÉNDICE A
Configuració del sistema

????? ????????????? ????????????? ????????????? ????????????? ?????????????

A.1 Fase d'inicialització

????? ????????????? ????????????? ????????????? ????????????? ?????????????

A.2 Identificació de dispositius

????? ????????????? ????????????? ????????????? ????????????? ?????????????

APÉNDICE B
??? ????????????? ????

????? ????????????? ????????????? ????????????? ????????????? ?????????????