Máster Universitario de Investigación en
Ingeniería de Software y Sistemas Informáticos

Simulación del Internet de las cosas

TRABAJO FIN DE MÁSTER
ITINERARIO DE INGENIERIA DEL SOFTWARE

AUTOR: Fernando Jesús Fernández Carrillo
DIRECTOR: Ismael Abad Cardiel

Curso Académico 2014/2015
Convocatoria Junio

Máster Universitario en Investigación en Ingeniería de Software y Sistemas Informáticos

ITINERARIO: Ingeniería del Software

CODIGO DE ASIGNATURA: 31105128

TÍTULO DEL TRABAJO: Simulación del Internet de las Cosas

TIPO DE TRABAJO: Tipo A, proyecto específico propuesto por un profesor.

AUTOR: Fernando Jesús Fernández Carrillo

DIRECTOR: Ismael Abad Cardiel.

Resumen

El objetivo de este proyecto es adquirir los conocimientos necesarios sobre el área conocida sobre el Internet de las cosas para poder realizar una simulación de un hogar inteligente. Esta simulación permitirá realizar un estudio sobre cómo se comportaría los diferentes nodos al interactuar entre sí y con el usuario.

Este proyecto se puede dividir en dos fases, por un lado está la fase de documentación, donde se realiza un estudio sobre cómo está actualmente el área del Internet de las cosas, centrándonos en las utilidades con más repercusión en la actualidad.

Por otro lado, se realiza una simulación de un hogar inteligente con diferentes tipos de nodos, siendo el comportamiento de algunos de ellos aleatorio. Una vez terminada la simulación, se realiza un análisis de los resultados.

Palabras clave: Internet de las cosas, simulación, *DEUS*, comunicación entre dispositivos.

Abstract

The aim of this project is obtain enough know lodgement about the area knows Internet of things to be able to do a simulation of a smart home. This simulation allow us make a research about how it will be the behaviour of the different nodes when they interact with each other and with the user.

This project can be split in two phases, the phase one is about read documentation, and the aim is make a study about how it actually is the state of Internet of things, making focus on its main utilities.

In the phase two there are a simulation of a smart home with different kind of nodes, with some nodes with random behaviour. When simulation ends, the results will be analysed.

Keywords: Internet of things, simulation, *DEUS*, communication device to device.

Índice

Índice de figuras

1. Introducción

Con la llegada de los teléfonos inteligentes se está avanzado hacia un mundo donde los elementos estén interconectados que ofrecen información útil para las personas en tiempo real. A este se le conoce como Internet de las Cosas [7], a partir de este momento me lo referenciaré como IoT por sus siglas en inglés (*Internet of things*). Gracias a las oportunidades que nos ofrece el tener todos los dispositivos de nuestro entorno conectados entre sí, podemos utilizar la información para saber el estado del tráfico, vigilar el estado de salud en pacientes o controlar todo lo que ocurre en nuestra vivienda.

Es en este último ejemplo donde me voy a centrar en este trabajo, aunque también se hablará de los últimos avances que se han producido en el mundo de la biomedicina y de las ciudades inteligentes.

Es este proyecto se va a crear una simulación de cómo se comportarían los elementos de una vivienda donde se han instalado sensores para controlar elementos como las luces de las habitaciones o la temperatura de la vivienda, para realizar esta simulación utilizaré herramientas que se han desarrollado para simular el mundo del IoT.

1.1. Plan de trabajo

En primer lugar es necesario tener una fase de documentación previa al desarrollo del proyecto, en esta fase he utilizado artículos y libros que ya había utilizado en otras asignaturas del Máster en Ingeniería del Software y Sistemas Informáticos, como por ejemplo computación ubicua, arquitecturas orientadas a servicios y sistemas difusos de apoyo a la toma de decisiones. Con la documentación leída y asimilada es momento de preparar el plan de trabajo que seguiré a la hora de hacer el proyecto que constará de las siguientes fases:

- Realizar una búsqueda de las herramientas de simulación que existen actualmente, buscando además, como se utilizan, para que sistemas están recomendadas, cuáles son sus requisitos y cuáles son sus limitaciones.
- Una vez que se ha seleccionado una herramienta, realizar una pequeña simulación que sirva como toma de contacto con el mundo de la simulación del IoT. No es un requisito indispensable que esta mini simulación esté en la simulación final.
- Definir los elementos de la arquitectura de red que se va a simular. Esta fase del plan de trabajo está muy relacionada con el apartado cuatro de este documento.
- Realizar una simulación. Con los conocimientos adquiridos en la fase de documentación, la herramienta seleccionada y los elementos definidos en la arquitectura, realizar una simulación de una vivienda inteligente donde los elementos se comuniquen entre sí, con la mínima interacción de las personas que viven allí.
- La última fase se corresponde con la fase de pruebas. Hay que detectar los fallos, los errores y el funcionamiento imprevisto para solucionarlo y que le funcionamiento de la simulación sea el más óptimo posible.
- Elaboración de la memoria del proyecto, en ésta se hablará sobre los diferentes sistemas que se pueden simular en el mundo del IoT y los objetivos de cada uno de estos sistemas. También explicaré como he realizado la simulación del comportamiento de una vivienda inteligente y analizaré los resultados obtenidos.

1.2. Conocimientos adquiridos.

La realización de este proyecto me ha aportado conocimientos relacionados con el mundo del IoT, en concreto con la simulación de viviendas inteligentes en las que todo se utiliza de forma automática y autónoma haciendo la vida de las personas que viven allí más fácil.

Además de los conocimiento adquiridos en el área de los hogares inteligentes he aprendido cómo funcionan los otros objetivos que más importancia tienen en la actualidad en el mundo del IoT: biomedicina y ciudades inteligentes.

Además me ha servido para relacionar entre sí los conocimientos adquiridos en las otras asignaturas del máster, por ejemplo, la utilización y el comportamiento del servidor puedo configurarlos gracias a los conocimientos adquiridos en *Arquitecturas orientadas a servicios* y *Sistemas difusos de apoyo a la toma de decisiones*. Gracias a *Computación ubicua* he aprendido cómo deberían ser los nodos para que al usuario no le moleste. Para realizar la simulación he utilizado conocimientos que adquirí en *Generación Automática de Código* ya que los nodos suelen tener un esqueleto común cuya creación se puede automatizar. Por último, la asignatura *Desarrollo de líneas de producto software* me ha servido para tener claro como tendría que crear el sistema para que el software del mismo pueda ser desarrollado en un área comercial.

1.3. Herramientas.

Para la realización de la simulación he utilizado el software de desarrollo *Eclipse* [8], concretamente su versión *Luna*. Con este software he descargado la versión 0.6.0 de la herramienta de simulación *DEUS* [9]. Para poder disponer de este software es necesario un PC con conexión a Internet y con potencia suficiente para que la simulación termine en un tiempo razonable.

1.4. Estructura del documento.

Este documento está dividido en siete apartados que forman la memoria del trabajo fin de máster. Cada uno de los seis primeros apartados está pensado para cubrir cada una de las áreas más importantes definidas en el plan de trabajo.

El apartado uno realiza una breve introducción del mundo del IoT, a continuación, se centra en explicar la estructura del documento, así como el plan de trabajo, los conocimientos adquiridos y las herramientas necesarias para realizar la simulación.

El apartado dos explica los objetivos que me he marcado en la realización de este trabajo y él ámbito donde se sitúa.

El apartado tres explica en qué estado se encuentra actualmente el mundo del IoT, comienza explicando lo que son las redes de sensores, después se explica el modelo del IoT, destacando los apartados más importantes, como el modelo de entidades, modelo de recursos y búsqueda semántica. Para finalizar este apartado se habla de cómo se simulan hoy en día modelos del IoT, en este apartado se explica la simulación en el área de la medicina y en escenarios urbanos.

En el apartado cuatro se realiza una propuesta para una arquitectura de red. Es en este apartado donde me empiezo a centrar en la orientación del IoT en el área de los hogares inteligentes. Comienzo explicando los elementos de la red, que en la simulación se convertirán en nodos y después de explicar los elementos, explicaré los elementos de comunicación. Puesto que es una simulación y su uso no es objetivo de este trabajo, estos

elementos no van a ser simulados, sino que son los que se utilizarían si lleváramos la simulación al mundo real.

El apartado cinco es donde se explica en detalle cómo se ha realizado la simulación. Se explica cómo se crea cada nodo en la simulación, que eventos están relacionados con el funcionamiento de la misma y que procesos son los responsables de relacionar los eventos y los nodos. Una vez que se ha explicado cómo se ha hecho la simulación y se han analizado los resultados obtenidos, se muestran los diagramas de clases de las clases relacionadas con la simulación (no del simulador) y el diagrama de secuencia de los eventos más importantes.

El sexto apartado realiza un resumen de las conclusiones que se pueden extraer a partir del trabajo realizado, y siguiere una serie de líneas por las que podrían seguir futuras iteraciones en el caso de seguir desarrollándolo.

Para finalizar, el apartado siete contiene toda la bibliografía y enlaces de descargas que han sido necesarios para el desarrollo de este trabajo, también incluiré las direcciones de descarga de las herramientas necesarias para la simulación.

2. Ámbito del proyecto y objetivos

Es este apartado voy a listar los objetivos que me he marcado a la hora de realizarlo y que debo cumplir para poder concluir que la realización del mismo ha finalizado con éxito. Una vez que tenga los objetivos listados, explicaré el ámbito en el que se desarrolla este trabajo.

2.1. Objetivos.

Los objetivos a lograr en este proyecto son los siguientes:

- Aprender los distintos protocolos de comunicación que existen en la actualidad y que permiten a los sistemas comunicarse entre sí, sin necesidad de acción humana.
- Aprender cómo funciona el mundo del IoT, comprendiendo en qué estado se encuentra actualmente y cuáles son sus principales áreas de desarrollo.
- Familiarizarse y entender cómo funciona alguna herramienta de simulación del IoT.
- Entender cómo se puede realizar una simulación sencilla relacionada con el mundo del IoT.
- Realizar una simulación de un hogar inteligente en el que los elementos se comuniquen e interactúen entre sí con la mínima intervención humana.

2.2. Ámbito del proyecto

El ámbito de trabajo para este proyecto se centra en la simulación del IoT usando las herramientas que hay con licencia gratuita actualmente. Las simulaciones se van a ejecutar sobre la máquina virtual de Java, lo que proporciona una base para que se pueda repetir la simulación en cualquier sistema operativo.

Al utilizar una herramienta como *DEUS* [14] para realizar la simulación tenemos una base que proporciona características que permiten simular también la simulación del movimiento de los nodos y otros comportamientos que aunque en este trabajo no se utilicen, sí que pueden ser necesarios para realizar evoluciones de la simulación.

3. Estado del arte

En este apartado voy a hablar sobre el estado actual de las redes de sensores en el IoT. Lo he dividido en tres apartados para poder explicar en detalle cada uno de los puntos más importantes en la citada área.

En el primer apartado trataré el tema de las redes de los sensores. Una red de sensores no deja de ser un conjunto de equipos interconectados entre sí, pero son las características de los sensores las que permiten la creación de configuraciones específicas que para obtener los mejores resultados según los objetivos fijados.

En el segundo apartado abordaré como se modelan actualmente las entidades, indicando como se representan sus propiedades y como se relacionan con otras entidades. Un buen modelado de las entidades implica un mejor conocimiento de cómo se comportan las cosas en el mundo real, lo que implica que podremos predecir mejor su respuesta ante situaciones adversas. Además se pueden realizar estudios de cómo se relacionan e interactúan los diferentes objetos entre sí.

Por último, en el tercer apartado, una vez que se han estudiado las redes de sensores y las entidades, se procederá al estudio del siguiente paso, que es simular el IoT antes de implementarlo en el mundo real. Actualmente se dispone de numeroso software que permite de una forma sencilla definir las propiedades de los sensores, una vez definidos estos dispositivos, se podrán simular las redes para estudiar tiempos de respuesta, comportamiento ante errores, etc.

3.1. Redes genéricas de sensores

Actualmente existen gran variedad de dispositivos diferentes, que utilizan un hardware con características diferentes lo que hace muy difícil que se desarrolle software de calidad específico para cada dispositivo. El desarrollo de tanta variedad y la aceptación que está teniendo, orienta el desarrollo hacia la creación de redes de sensores globales [1]. Estas redes están pensadas para que, por ejemplo, distribuyendo diferentes tipos de sensores por una ciudad, los usuarios puedan saber el estado del tráfico, los aparcamientos disponibles, el nivel de contaminación en tiempo real, etc.

Como se ha comentado, para la creación de estas redes de sensores uno de los problemas más importantes que hay que solventar es el desarrollo de aplicaciones que se aísle de tanta heterogeneidad de hardware. Para ello, una de las propuestas que hay es la creación de un software intermedio o *middleware* que se encargue de interpretar los datos que obtenga del sensor y tratarlos para que los reciba el software genérico y en sentido contrario, es decir, recibir las órdenes del software genérico y ser capaz de traducirlas para que el sensor las interprete.

En el artículo [1] se propone utilizar documentos *XML* para crear sensores virtuales con los que poder crear redes de sensores, en los que además de los datos relacionados con las propiedades del sensor, haya que añadir una marca de tiempo que indique cuando fueron tomados estos datos para poder ver su evolución en tiempo real. En la Figura 1 se muestra un ejemplo de cómo sería el código *XML* que se obtendría de un sensor.

```
<sensor1>
   <timestamp>1426631506793</timestamp>
   <propiedad1 value="valor1"/>
   <propiedad2 value="valor2"/>
   <propiedad3 value="valor3"/>
   <propiedad4 value="valor4"/>
</sensor1>
```

Figura 1: Definición de un sensor

La creación de este tipo de redes virtuales es que al utilizar documentos *XML* para simular sensores, se pueden utilizar los principales lenguajes de programación (Java, Ruby, C#, etc) para tratar estos documentos. Utilizar nodos virtuales, permite que se incremente rápidamente el número de nodos sin aumentar el coste, permitiendo así realizar un estudio de cómo se comporta la red si crece tanto en el mundo real.

En este punto, se ha estudiado una manera de simular sensores con diferente hardware y diferente funcionalidad, ahora vamos a estudiar cómo comunicar estos nodos y solucionar el problema de la escalabilidad.

Actualmente, las redes que mejor toleran la escalabilidad son las redes *P2P* (peer-to-peer), en estas redes, todos los sensores se comunican entre sí como iguales, es decir, actuando como cliente y servidor.

Para desarrollar estas redes se propone en [2] definir varias funciones en los nodos que implementen el comportamiento deseado de los nodos. Las funciones que se definen son:

- *CREATE* que convierte un sensor en un nodo recolector de información.
- *JOIN* que une un sensor a un nodo ya creado.
- *MULTICAST* responsable de que el nodo envíe un mensaje a todos los sensores que tiene asociados
- *LEAVE* notifica a los sensores hijos que el nodo al que pertenecen ya no existe y una vez que se lo ha notificado a todos, se auto elimina.

En la Figura 2 se muestra una imagen de ejemplo de una red P2P con nodos y sensores.

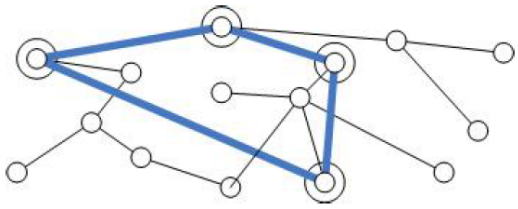


Figura 2: Red P2P con nodos y sensores

Utilizar redes *P2P* permite a los nodos descubrirse entre sí sin necesidad de un índice y utilizar la funcionalidad de los sensores que se requiera en el momento dado. Las redes *P2P* definidas en esta subsección se conocen como redes mixtas ya que, aunque todos los sensores son iguales (obviando las diferencias de su funcionalidad), existen otros nodos que se encargan de gestionar la información y encaminar las consultas que se realizan entre nodos.

3.2. Modelos de Internet de las cosas

En este apartado voy a hablar sobre el modelado de las entidades en el IoT. Además de cómo se modelan las entidades, voy a hablar de cómo se relacionan entre sí, si por asociación, acceso a recursos, etc.

El modelo definido en el IoT presenta una variedad enorme de dispositivos que han de comunicarse entre sí, por este motivo, para modelarlos correctamente, en [3] se fijan una serie de requisitos que se han de cumplir a la hora de realizar un modelado. Estos requisitos son:

- Identificación de los posibles conceptos en el marco de trabajo del IoT y la estructura de su representación.
- Mecanismo de acceso que ofrezcan una interfaz homogénea a todos los dispositivos.
- Automatizar la interoperabilidad entre máquinas para las integraciones con aplicaciones existentes.

El grupo de incubadora del *W3C* de redes semánticas de sensores ha presentado una ontología [6] para describir sensores y redes de sensores. La ontología representa un esquema de modelado de alto nivel que describe el dispositivo que es el sensor en sí mismo, sus capacidades, plataformas y otros atributos relacionados con las redes semánticas de sensores y las aplicaciones web. Las entidades se utilizan como puntos de extensión dentro del modelo de dominio.

Para la comunicación de los servicios en las ontologías se ha definido un lenguaje conocido como *Ontology Web Language for Services (OWL-S).* Este lenguaje proporciona los principales atributos para describir servicios y sus atributos funcionales. Al igual que los Servicios Web tradicionales se pueden publicar en repositorios públicos, los Servicios Web Semánticos se pueden publicar en repositorios específicos. *OWL-S* especifica como el servicio es invocado técnicamente por el servicio consumidor incluyendo la dirección de red del *endpoint* del servicio. Aunque no se está obligado a usarlo, *OWL-S* usa *Web Service Description Language (WSDL)* como mecanismo de granularidad. En la Figura 3 se muestra un ejemplo del lenguaje *OWL-S*.

```
<owl:DatatypeProperty rdf:ID="parameterType">
  <rdfs:domain rdf:resource="#Parameter"/>
  <rdfs:range rdf:resource="&xsd;anyURI"/>
</owl:DatatypeProperty>

<owl:Class rdf:ID="Parameter">
 <rdfs:subClassOf>
   <owl:Restriction>
     <owl:onProperty rdf:resource="#parameterType" />
     <owl:minCardinality rdf:datatype="&xsd;#nonNegativeInteger">
              1</owl:minCardinality>
   </owl:Restriction>
 </rdfs:subClassOf>
</owl:Class>
```

Figura 3: Ejemplo de fichero OWL-S

Para poder realizar un modelado de información es necesario tener claro una serie de conceptos que son de suma importancia en este ámbito:

- Entidad: La entidad constituye las "cosas" del IoT y es el objeto físico que se representa, puede ser cualquier cosa. Es responsable de la interacción entre las personas y/o los agentes software.
- Dispositivo: Es el que enlaza una entidad o una parte del entorno de una entidad para poder monitorizarla.
- Recurso: Es el componente software que proporciona información a la entidad o habilita el control del dispositivo.
- Servicio: El responsable de proporcionar una interfaz estandarizada y bien definida, ofreciendo toda la funcionalidad necesaria para interactuar con entidades y procesos relacionados.

Estos cuatro conceptos pueden parecer algo inconexos y es difícil ver una relación entre ellos, por este motivo voy a utilizar un ejemplo para ayudar a que se comprenda mejor como se relacionan entre sí. Supongamos que la entidad es un sensor de temperatura, es el objeto físico que se instala en las viviendas, en este ejemplo, el dispositivo sería una pantalla que nos muestra continuamente la información de la temperatura en esa habitación. El recurso sería la clase que en el sistema sería equivalente a la entidad, donde las propiedades de la entidad estarían representadas como atributos de la clase, por ejemplo, el valor de la temperatura de la habitación. Por último, el servicio sería una interfaz que tiene toda la funcionalidad disponible de la entidad utilizando el recurso para acceder, en el ejemplo del sensor de temperatura, la interfaz ofrecería la posibilidad de consultar la temperatura que lee la entidad.

En la Figura 4 se muestra como se relacionan estos conceptos entre sí:

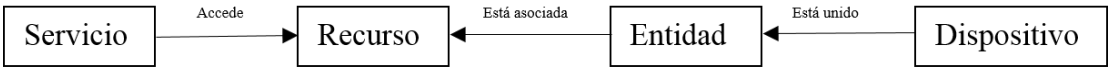


Figura 4: Modelo del Internet de las Cosas

La comunidad de la Web Semántica ha presentado especificaciones formales de definiciones cómo ontologías para modelar la información del dominio, para realizar el modelado utilizan un lenguaje conocido como *Web Ontology Language – Description Logic (OWL-DL).*

Con estos conceptos podemos crear los modelos de entidades, de recursos o de servicios. En los apartados 3.2.1, 3.2.2 y 3.2.3 explicaré como son cada uno de estos modelos.

Después de explicar los modelos, utilizaré los apartados 3.2.4 y 3.2.5 para explicar las diferentes formas de usar los modelos de información. Los modelos de información que se van a explicar son utilizando asociaciones dinámicas y utilizando búsqueda semántica.

3.2.1. Modelos de entidades.

Las entidades pueden tener ciertas caracterizas que necesitan ser tenidas en cuenta a la hora de modelarlas. La representación del lenguaje *OWL-DL* se puede usar para definir los modelos de entidades. Una entidad puede tener ciertas propiedades que incluyen atributos de dominio, además, una instancia de una entidad puede tener múltiples valores

para los atributos de dominio. Cada valor, está representado en un contendor con metadatos asociados a dicho valor. Los metadatos pueden utilizar para especificar unidades de los valores. Además, en los modelos de entidades se puede representar cuando alguno de los valores de los atributos es obtenido de un sistema externo mediante una *URI*.

En la Figura 5 se muestra un ejemplo de un modelo de entidad.

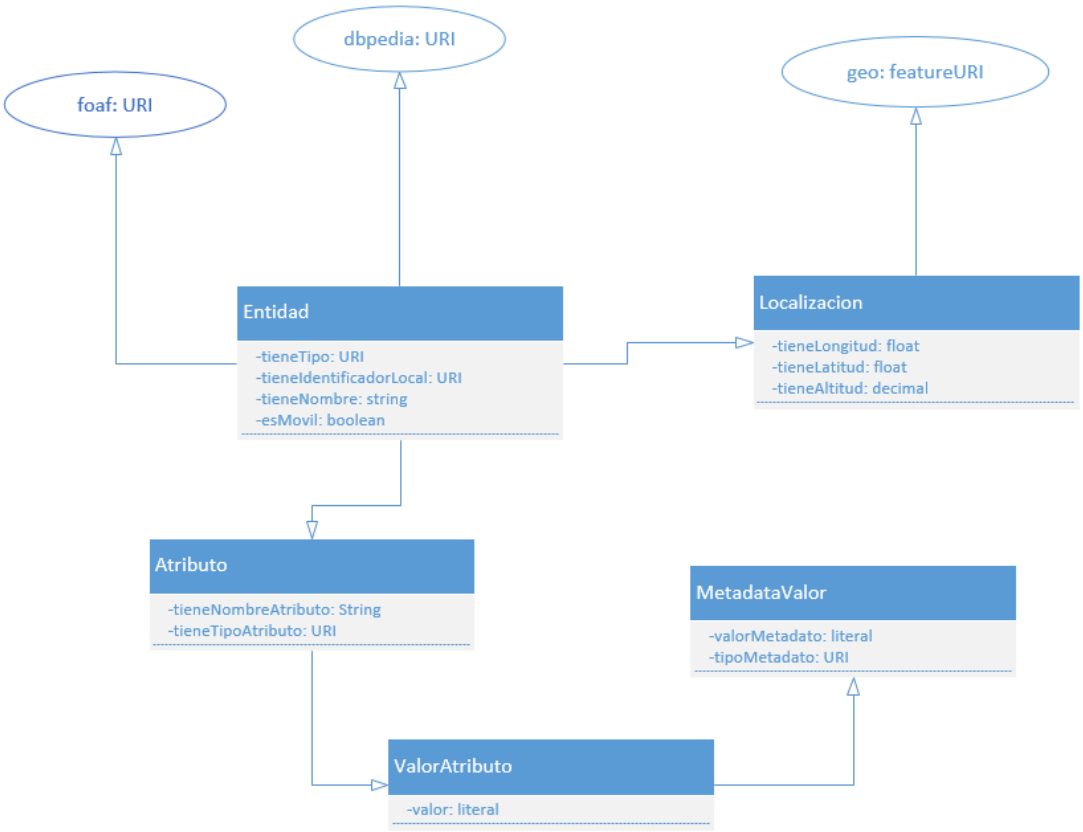


Figura 5: Modelo de entidades

3.2.2. Modelos de recursos.

El recurso es el núcleo del componente software que representa una entidad en el mundo digital. El concepto de recurso tiene los tipos de las propiedades que especifican sus características. Las restricciones funcionales denotan que un recursos solo puede tener un enlace a una instancia. El enlace al tipo de recurso se denota en términos de tipo de propiedad del concepto el tipo de tipo de recurso. Cuando el tipo es un sensor, la propiedad *hasType* se utiliza como enlace a la instancia del sensor que se indique en la ontología.

Los recursos pueden ser enlazados a sistemas externos a la ontología enlazando la *URI* del recurso externo, a los que accede gracias a la interfaz de acceso que tienen definidos todos los recursos en los Servicios Web Semánticos [21].

Además de acceder a recursos externos, el recurso que estamos tratando puede ser accedido por otros recursos externos, para ello, tiene que publicar su propia interfaz de acceso indicando el tipo de interfaz que utiliza. Junto con la interfaz de acceso y el tipo de interfaz, tiene que utilizar la propiedad *hasServiceEndpoint* para enlazar el recurso con la capa de servicio que expone la funcionalidad del recurso en los repositorios del IoT.

En la Figura 6 se muestra un ejemplo de modelo de recursos.

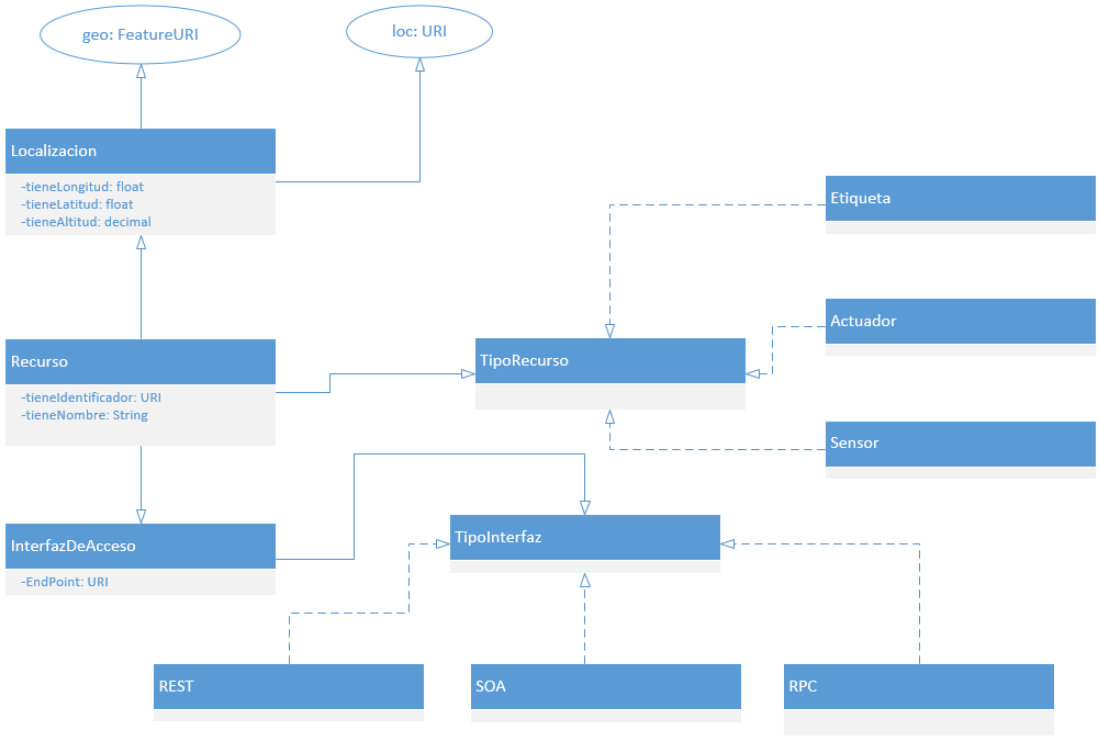


Figura 6: Modelo de recursos

3.2.3. Modelos de servicios.

Los recursos son accedidos por servicios que proporcionan funcionalidad para recopilar información sobre las entidades a las que están asociados.

Para ofrecer está funcionalidad es necesario buscar la terminología relacionada con las entidades de los dominios. La búsqueda devolverá la descripción del servicio que contiene un enlace al recurso que es capaz de proporcionar la información que ha sido solicitada en la búsqueda. Si el servicio necesita alguna información de entrada para ser procesada, este requisito puede ser especificado por las propiedades del recurso. Los atributos de cualquier entidad pueden ser utilizados para describir el significado de los parámetros de entrada y salida.

El recurso es accedido a través de Internet mediante su interfaz, usando, por ejemplo los servicios web. La *URI* del servicio tiene que estar definida en la interfaz de acceso del recurso.

En la Figura 7 se muestra un diagrama del modelo de servicios del IoT.

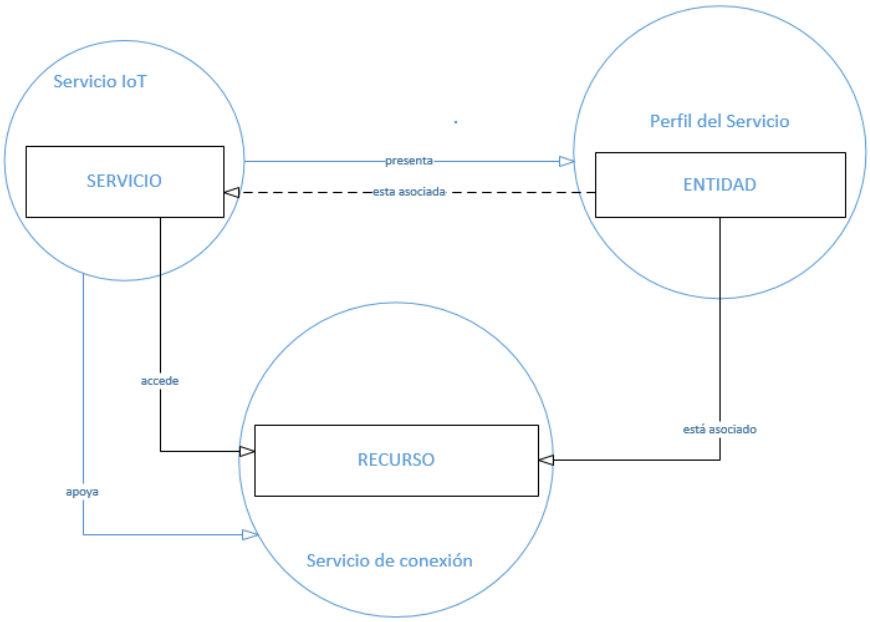


Figura 7: Modelo de servicio para Internet de las Cosas

3.2.4. Asociaciones dinámicas.

En los modelos de información presentados en los apartados 3.2.1, 3.2.2 y 3.2.3, las entidades físicas y los servicios que proporcionan la información no están conectados a través de enlaces físicos, sino que están conectados a través de asociaciones separadas y modeladas.

Tener las asociaciones separadas proporciona un alto nivel de flexibilidad ya que ofrece la posibilidad de que los servicios sean enlazados con varias entidades al mismo tiempo. Al tener varias entidades enlazadas al mismo tiempo, hay que aceptar la posibilidad de que el número de entidades enlazadas cambie con el tiempo, por lo que las asociaciones tienen que ser capaces de cambiar dinámicamente.

Las asociaciones dinámicas pueden ser manejadas por una infraestructura servidora como por ejemplo, la nube, donde las comunicaciones y la computación de recursos son muy numerosos. Además, la privacidad puede estar protegida asociando los servicios a las personas que tienen acceso a la información que suministran. Una asociación también contiene información sobre qué aspecto de la entidad física está asociada con el servicio.

3.2.5. Búsqueda semántica.

Utilizar la información representada en los modelos y usándola en las aplicaciones y servicios del IoT, también se puede encontrar los datos, descubrir las entidades, los recursos y/o los servicios pertinentes. Los datos semánticos pueden ser representados como datos enlazados.

Procesar y razonar a gran escala descripciones semánticas es también otro aspecto importante para hacer que la información esté más disponibles para los usuarios finales.

Los modelos presentados proporcionan tipos de descripciones parecidas, por lo que un método de indexado y búsqueda a gran escala de datos semánticos en el IoT puede ser implementado. Razonando las descripciones de la entidad, del recurso y del servicio en relación con otros datos en el dominio del IoT y los recursos que se describen en el

dominio de la aplicación y los atributos del entorno, también puede utilizarse para la comunicación autónoma y la toma de decisiones.

3.3. Simulación de Internet de las cosas

Una vez que se ha visto cómo se comunican los sensores y con las bases para modelar las redes de sensores en el IoT vamos a analizar cómo se realizan simulaciones de redes de simuladores en el IoT.

Como se podrá comprobar en los apartados 3.3.1 y 3.3.2, aunque las áreas que se van a simular sean muy diferentes, hay multitud de elementos que son comunes, y realmente no se produce una diferenciación hasta que no se concreta en uno de los dos campos. Ambas áreas de simulación presentan los mismos tipos de nodos (finales, de aplicación, de comunicación y de proceso), esto permite que las herramientas de simulación sean intercambiables entre sí, aunque en la actualidad es más común utilizar *MONARC* para simulaciones en el área de la medicina o *DEUS* con *OSMobility* para el área de las ciudades inteligentes, no habría ninguna limitación en intercambiar las herramientas y utilizar *DEUS* en el área de la medicina o *MONARC* en el área del IoT. No obstante, debido a que la herramienta *MONARC* se utiliza comúnmente en el área de la medicina, hay funcionalidad que está más desarrollada en esta herramienta que en *DEUS*.

3.3.1. Simulación del Internet de las cosas en el área de la medicina.

En [4] se ha realizado un estudio sobre cómo simular el IoT en el ámbito del cuidado de pacientes. Esta área es especialmente sensible a la hora de comprobar resultados ya que no se pueden utilizar pacientes para realizar pruebas con resultado incierto. La simulación evalúa varias fases de administración de recursos a través de muchos escenarios donde la experimentación con pacientes reales está muy limitada. Las operaciones del *back-end* se realizan en la nube, aunque los entornos actuales no permiten pruebas extensas en el área del IoT. Para sortear este problema, los autores de [4] identifican los mejores simuladores que podrían procesar dinámicamente la información de los sensores. La solución está basada en la utilización de dispositivos inalámbricos de corto alcance que se conecta a dispositivos como smartphones para transmitir la información en caso de alguna emergencia [16].

Actualmente existen numerosas herramientas para realizar simulaciones, cada una tienes sus ventajas e inconvenientes, además, tienen que cumplir un requisito fundamental en los escenarios contemplados por el IoT, que es la interconexión en tiempo real de dispositivos del IoT. En el artículo citado se realiza un resumen de las más comunes. No obstante, hay una herramienta de simulación que destaca de entre el resto por la cantidad de funcionalidad que puede desarrollar, esta herramienta es *MONARC* [10]. Es un marco de trabajo para simulación dirigido a la optimización de recursos en un entorno de computación distribuida, incluyendo la replicación de información. Este entorno ha sido desarrollado para imitar decisiones de administración de recursos avanzados en diferentes sistemas.

Con el objetivo de seguir la corregir las deficiencias y mejorar la investigación de la simulación de las redes con variaciones topológicas en entornos de la nube, se ha propuesto utilizar la herramienta conocida como *Simulating de Inter-Cloud (SimIC)* [11]. El objetivo de esta herramienta es facilitar la colaboración entre varias nubes para distribuir las peticiones de información a los servicios. Esta herramienta permite entre otras cosas el desarrollo de diseños que pertenecen a varias topologías y entidades en los

escenarios del IoT, soporta la simulación de entornos heterogéneos, incluye soporte para virtualización y es compatible con entornos de computación distribuida que están sujetos a cambios en tiempo real. Hay una herramienta derivada de esta conocida como *SimIoT*[4] que expande la funcionalidad de *SimIC* para dar soporte a los dispositivos del IoT.

SimIoT acepta datos generados por los dispositivos, por ejemplo sensores, que envía automáticamente a la nube. La comunicación entre entidades se realiza mediante el paso de mensajes. Las entidades no tienen que estar en la misma nube, ya que este sistema soporta computación distribuida. En la Figura 8 se muestra el diagrama de actividad del paso de mensajes con el punto inicial y el punto final.

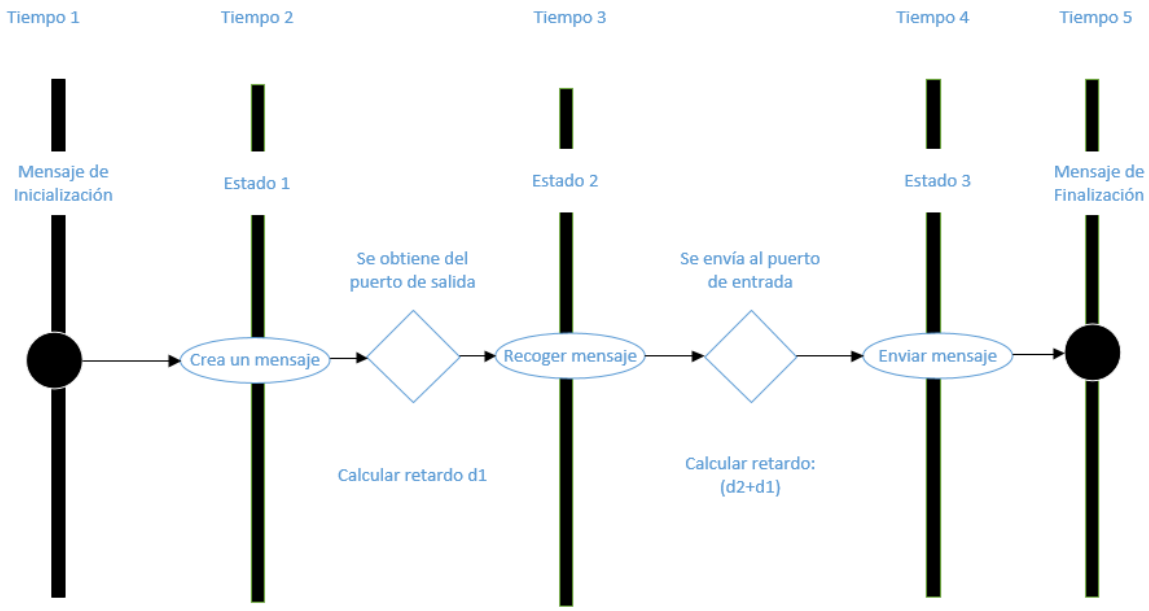


Figura 8: Diagrama de actividad

Como se muestra en la Figura 8, el mensaje es creado en el estado 1, posteriormente es recolectado por el estado 2 que lo envía al estado 3 donde el mensaje termina. En la Figura 9 se muestra la arquitectura a alto nivel de la herramienta *SimIoT*. Esta arquitectura va desde que la información es generada por los sensores a nivel de usuario para ser enviada al bróker que es el responsable de generar la petición de comunicación para procesar la información en la nube.

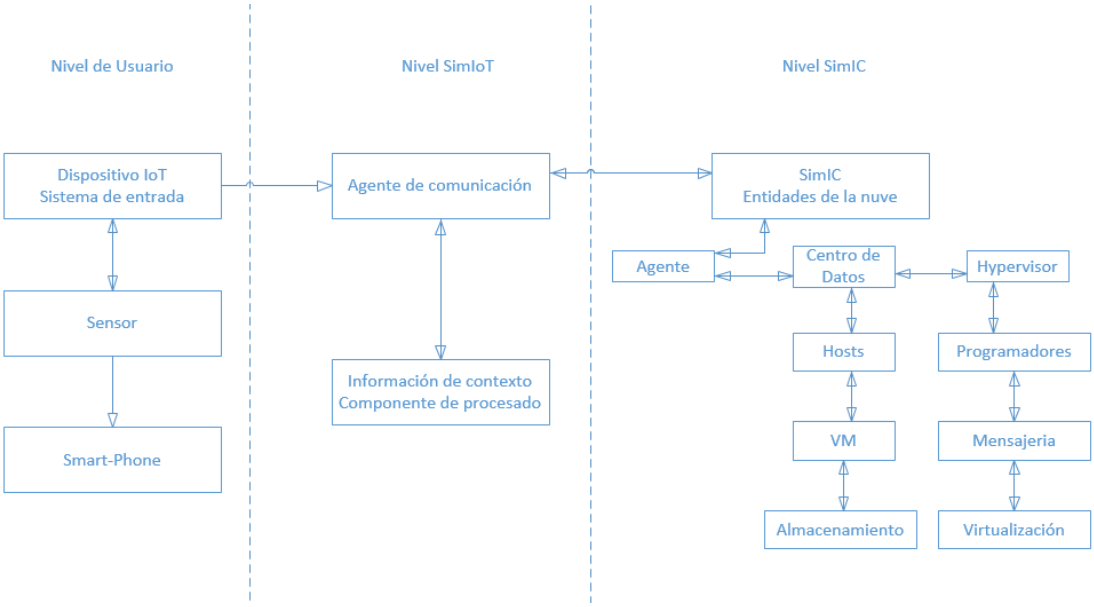


Figura 9: Arquitectura a alto nivel

La arquitectura de *SimIoT* engloba una estructura formada por tres capas:

1º Capa: Contiene las entidades que representan que objetos del sistema están incluidos. Los procesos de virtualización, data center, etc, son inicializados cuando las entidades comienzan la comunicación con la siguiente capa que representa el contenido del IoT.

2º Capa: Incorpora el comportamiento de la herramienta *SimIC* que representa las acciones que tienen lugar con el simulador. Las características principales son la utilización de puertos, funcionalidades y restricciones que manifiestan el comportamiento de la entidad. La comunicación está basada en etiquetas que son asignadas a los mensajes durante el intercambio e identifican el origen del mensaje y la operación requerida.

3º Capa: Incluye las operaciones de rendimiento y de rastreo. Las operaciones de rendimiento incluyen los tiempos de ejecución, los tiempos de cambio de servicio, el tiempo de respuesta del servicio, los niveles de utilización de la máquina virtual y las latencias del servicio. Las métricas están implementadas como en la versión por defecto de *SimIC*. Las operaciones de rastreo monitorizan toda la ejecución del servicio mostrando información de entrada y de salida, eventos de logging, etc.

En la Figura 10 se muestra un ejemplo de las capas de la estructura.

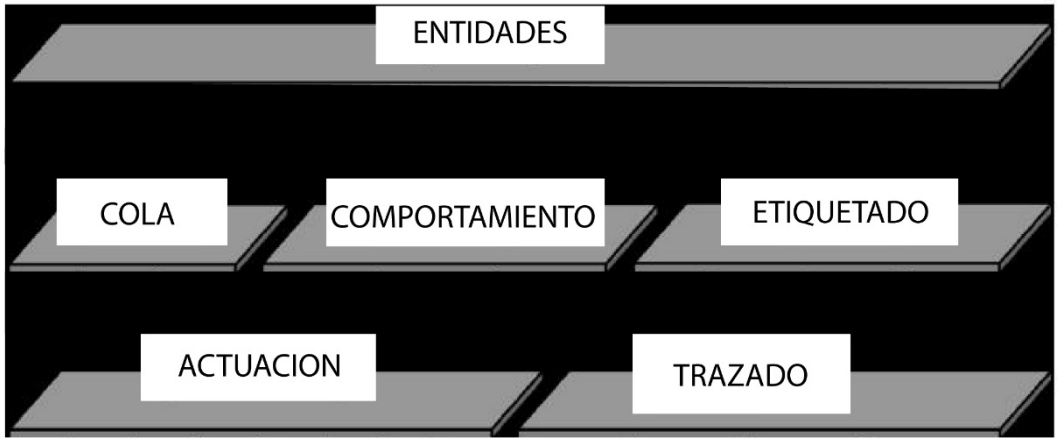


Figura 10: Estructura en capas de SimIoT

Todas las aplicaciones que he comentado en este apartado son necesarias para poder realizar un sistema de monitorización de la salud en situaciones de emergencia.

La utilización de dispositivos de comunicación de corto alcance presenta muchas oportunidades para monitorizar la salud de los pacientes y, utilizando otros dispositivos como los smartphones, enviar la información a servidores remotos, además, los usuarios tienen acceso a la información ofrecida por el entorno de la medicina gracias a sus perfiles personalizados.

El escenario de emergencia definido por [4] comienza cuando un usuario está en una situación crítica y necesita cuidados médicos. El usuario interactúa con el entorno a través de su smartphone mediante el envío de peticiones de información al sistema a través del servicio basado en los sensores. La información habilita al entorno para tomar decisiones inteligentes basadas en los perfiles de salud personales del usuario y la información captada dinámicamente por los sensores. Un proceso de *Business Intelligence* puede ser usado para decidir qué acciones deben ejecutar los sensores en función de las restricciones existentes.

3.3.2. Simulación del Internet de las cosas en escenarios urbanos.

Además del ámbito del cuidado de pacientes, también se están realizando simulaciones en escenarios de entornos urbanos [15] [19]. En esta área también es importante poder realizar simulaciones ya que los proyectos suelen tener un elevado número de sensores, lo que encarece su desarrollo. En [5] se realiza una propuesta de una plataforma de simulación en escenarios de entornos urbanos a gran escala.

Cuando nos referimos al IoT, la mayoría de las veces nos referimos a la conexión de multitud dispositivos entre sí, con limitaciones computacionales y de memoria. Las aplicaciones requieren un gran número de dispositivos equipados con sensores y actuadores con tienen que cooperar y ser coordinados. Los autores de [5] proponen utilizar una plataforma de simulación basada en Java que permita la simulación de dispositivos del IoT con una metodología intuitiva.

La metodología propuesta está basada en el concepto de nodo del IoT, que representa un objeto inteligente dotado con un modelo de movilidad y al menos un modelo de red con el que puede interactuar para y proporcionar información de retorno para mejorar el comportamiento de los nodos simulados. Los modelos de red describen las propiedades de cada interfaz con las que cada nodo del IoT está equipado. En la Figura 11 se muestra un modelo visual de un nodo del IoT. Los modelos de red fueron enlazados a modelos de energía cuando el uso de las interfaces de red empezó a afectar al consumo de energía, que varía dependiendo del tipo de tecnología de acceso a la red.

Un nodo del IoT está caracterizado por tener diferentes interfaces de red, varios protocolos de comunicación y recursos que pueden ser identificados por su *URI*. Es posible simular diferentes aplicaciones que hacen uso de los dispositivos del IoT caracterizados por tener modelos de movilidad, de red y de energía.

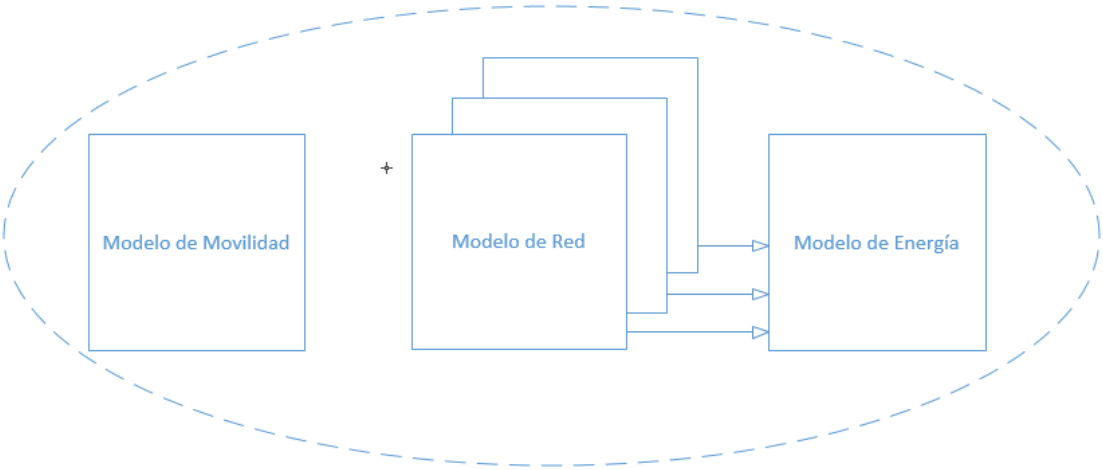


Figura 11: Modelo visual de un nodo en el Internet de las cosas

Ya hemos definido los nodos del IoT, en el simulador, existe una clase de la que pueden heredar los desarrolladores para crear y simular aplicaciones para el IoT. Es esa herencia la que permite aislar la capa de la aplicación de las capas inferiores de la arquitectura. En la Figura 12 se muestra la arquitectura propuesta por los autores de [5]. La capa superior de la arquitectura representa la aplicación que se va a probar, mientras que las capas inferiores están preparadas para la coordinación de los nodos del IoT.

Debido a que el objetivo de la simulación es tener un número elevado de nodos del IoT, se ha elegido utilizar la herramienta *DEUS* [13] debido a que permite una gran escalabilidad y versatilidad, además está desarrollada en Java y es un proyecto libre. La API de *DEUS* [14] permite a los desarrolladores implementar nodos que interactúen en sistemas complejos, eventos, como crear y destruir nodos, y procesos. Existe una extensión de *DEUS* dedicada al modelado de nodos móviles que mantiene sus características y abstracción. Esta extensión se llama *OSMobility*.

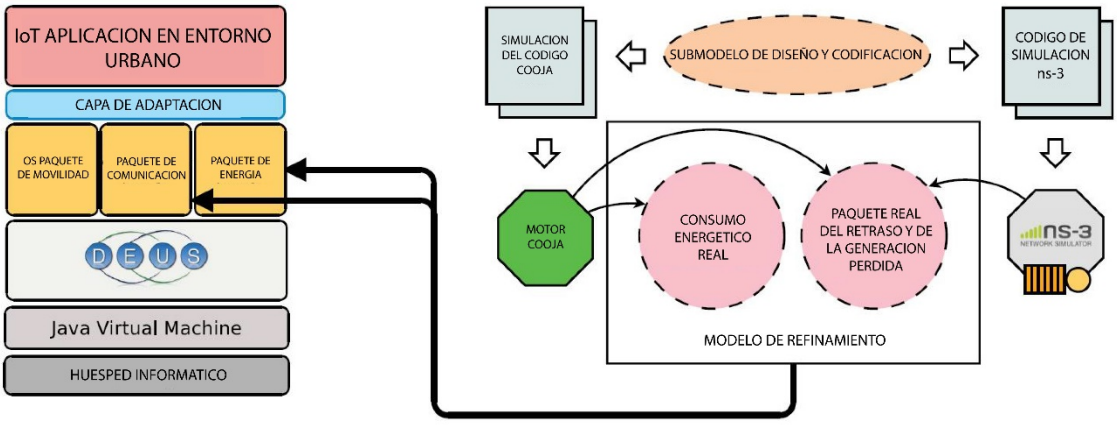


Figura 12: Representación en capas de la metodología propuesta

OSMobility es un entorno de simulación que permite simular el movimiento de diferentes entidades. Está integrado con *OpenStreetMap*, que es una base de datos abierta que proporciona datos geográficos. Una de las principales características de este entorno es que permite definir cualquier modelo de movilidad. La API se usa para calcular la siguiente posición del nodo en función de su posición actual, velocidad, dirección y densidad de nodos a su alrededor. En la propuesta de la simulación del IoT en un entorno

urbano se ha envuelto el nodo del IoT con un nodo del entorno *OSMobility* para aprovechar sus propiedades de geo localización.

Como se ha comentado, un nodo del IoT cosas tiene un modelo de red y un modelo de energía. Es posible describir las capacidades del nodo respecto a las interfaces de red y los protocolos de comunicación y utilizar *DEUS* para simular nodos en sistemas distribuidos que se comunican entre sí. El paquete de comunicaciones proporciona varios modelos de retardo que pueden ser usados para simular la transmisión del mensaje entre nodos de la red. Si el propósito de la simulación es el de medir el retardo medio de la propagación de los mensajes entre nodos que no están directamente conectados, el valor del retardo de cada salto debe ser realista. Para obtener el realismo necesario para darle validez a la simulación, se utilizan dos herramientas especializadas en esta función, el nombre de estas herramientas es *Cooja* y *ns-3*. Además, *Cooja* puede ser utilizado para obtener un modelo realista del consumo de energía.

Cooja es un simulador de red que en realidad es un sistema operativo libre para redes, sistemas con restricciones de memoria y sistemas centrados en dispositivos del IoT que se comunican sin cables y tienen corto alcance. Por su parte, *ns-3* es un simulador de eventos de red para sistemas de Internet. Está pensado para construir un núcleo de simulación sólida que sea fácil de usar y de depurar y que permita la monitorización de todo el flujo de información de la simulación. Soporta comunicaciones basadas en redes IP y en redes no-IP

La integración directa de *DEUS* con *Cooja* y *ns-3*, cuando el primer nodo “llama” a los demás es necesario calcular el retardo cada vez que un nodo se comunica con otro, además hay que tener en cuenta el cálculo del consumo de energía, por estas razones, no es práctica la integración entre estos sistemas ya que se incrementaría el tiempo de simulación. En lugar de la integración directa, es más efectivo y eficiente realizar los siguientes tres pasos:

- Identificar los principales tipos de subsistemas, cada uno caracterizado por sus características de red.
- Utilizando *Cooja* y *ns-3* crear modelos detallados de la simulación de los subsistemas, y midiendo los retrasos caracteristicos de la transmisión y los perfiles de energía.
- Con *DEUS* simular todo el sistema distribuido, incluyendo la comunicación de los eventos, teniendo en cuenta los retrasos de la transmisión y el consumo de energía estimado en el paso anterior.

Para evaluar la plataforma de simulación y su escalabilidad se ha propuesto un escenario y se ha calculado el tiempo para completar la simulación. El escenario es el típico cuando se habla del IoT en entornos urbanos: una infraestructura de parking inteligente.

Se ha definido que el vehículo solicite entradas con un proceso de *Poison* homogéneo cuyo tiempo entre intervalos es exponencialmente distribuido en variables aleatorias cuyo intervalo entre llegadas es de 5 minutos. Cuando vehículo solicita la entrada a un parking, comienza enviando su posición geográfica a la entrada más cercana. A continuación, el sistema de la entrada contacta con los sensores que están bajo su responsabilidad para enviar una respuesta al vehículo. Todos los participantes de la simulación han sido implementados por subclases de nodos del IoT. Estos nodos se comunican utilizando el

protocolo *CoAP*, En este ejemplo, la portabilidad del código es mayor e incluso la implementación puede ser fácilmente transferida a dispositivos reales.

4. Propuesta de arquitectura para la red

Es este apartado voy a diseñar un arquitectura de red, explicando todos los sensores, dispositivos y componentes que son necesarios para después poder simularla. Los elementos que voy a utilizar se pueden agrupar en cuatro categorías: nodos finales, nodos de proceso, nodos de conexión y nodos de aplicación. Estas cuatro categorías están explicadas a continuación:

- **Nodos finales**: Son los dispositivos que se encuentran en el extremo de la red del IoT. Pueden ser sensores, lectores de tarjetas RFID, cámaras, etc.
- **Nodos de proceso**: Estos nodos se pueden comunicar con otros nodos proceso y realizar alguna son los responsables de recibir y procesar la información de los nodos finales. También acción con la información que tienen.
- **Nodos de conexión**: Son los responsables de que se transmita la información entre el nodo final y su nodo proceso o entre nodos proceso.
- **Nodos de aplicación**: Estos nodos son los actuadores que reciben la orden del nodo de proceso y pone en marcha o paran algún proceso real.

En la Figura 13 se muestra una imagen esquemática de una arquitectura de red con los diferentes nodos que se van a utilizar en la simulación.

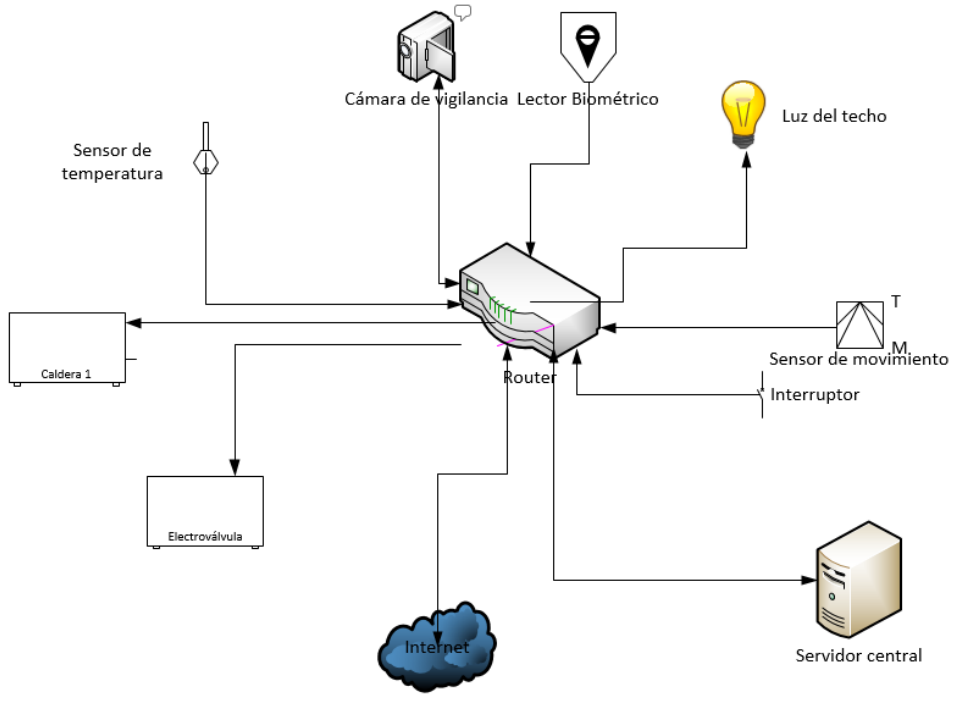


Figura 13: Esquema nodos Internet de las cosas

En el apartado anterior he hablado de simular el IoT en el ámbito de la medicina (apartado 3.3.1) y en ciudades inteligentes (apartado 3.3.2). Otro ámbito muy importante en el IoT es el área de la domótica. Es posible realizar una red de elementos del IoT para controlar aspectos como la calefacción, el encendido de las luces, la seguridad, etc… Es ese ámbito donde voy a centrar este apartado y el siguiente. En este apartado voy a explicar que sensores se utilizarían para transformar una vivienda normal en una vivienda

domótica, y en el apartado cinco, realizaré una simulación de dicha vivienda utilizando los simuladores descritos en el apartado 3.3.

Voy a comenzar explicando cada uno de los dispositivos que voy a explicar, indicando en que categoría de las indicadas anteriormente entraría y cuál sería la función de cada uno de ellos. Una vez definidos los elementos, explicaré los protocolos que se van a utilizar para comunicarse entre ellos.

4.1. Elementos de la red

En la Figura 14 se muestra un plano de una vivienda con los dispositivos que se van a explicar en este apartado. Se han definido cuatro tipos de nodos de aplicación, los interruptores, la caldera, las electroválvulas y las cámaras de seguridad. Como nodo de comunicación tenemos un router y como nodo de proceso se dispondrá de un equipo que realizará las funciones de servidor central. Por último, los sensores finales serán los sensores de movimiento y los sensores de temperatura.

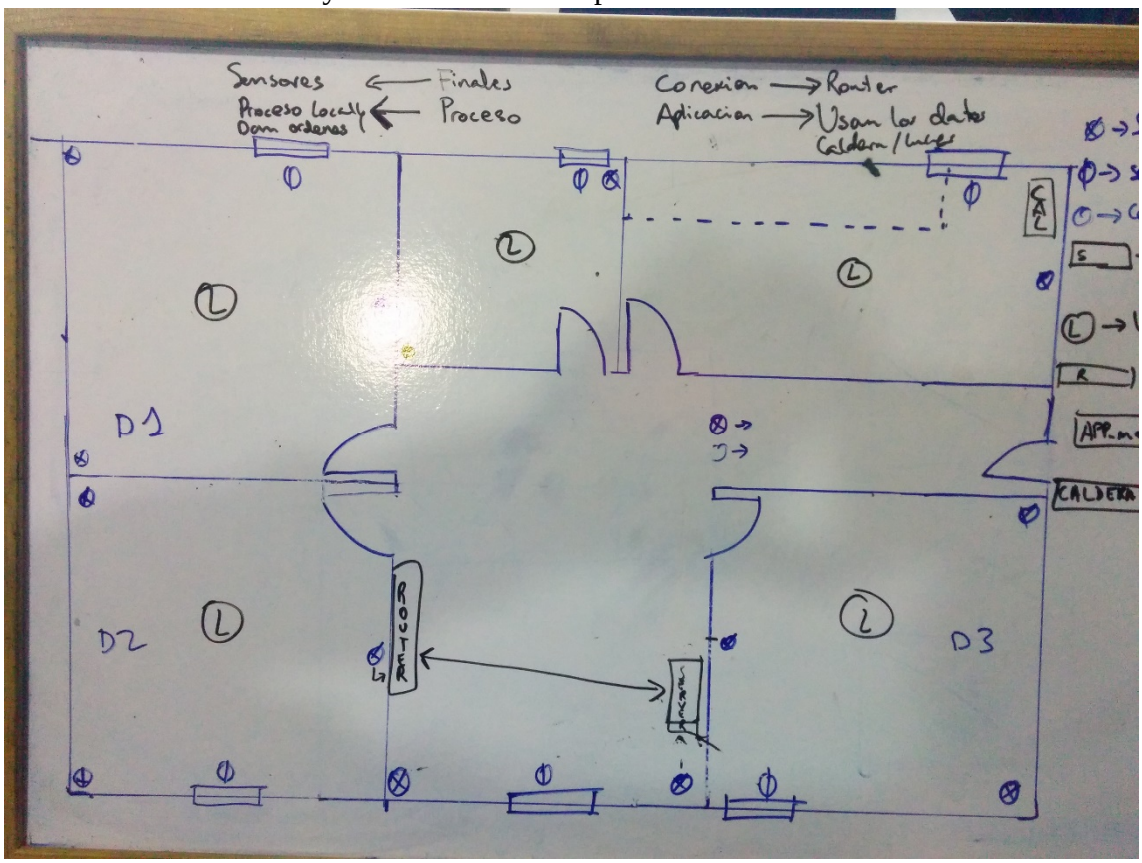


Figura 14: Plano de vivienda

En cada una de las habitaciones de la vivienda se dispondrá de un termómetro cerca de la ventana que, gracias al router, informará permanentemente al servidor central de la temperatura en esa habitación. Cuando la temperatura baje de los grados configurados, se realizará un estudio de si se tiene que electroválvulas se tienen que abrir y si se tiene que activar o no la caldera.

El control de encendido de luces se hace combinando el sensor biométrico que además identifica la persona y los sensores de movimiento y lumínicos. Cuando una persona está en una habitación y la luminosidad dentro de la misma cae hasta un cierto

nivel, el servidor estudia si tiene que encender o no la luz, dependiendo de si hay personas en esa habitación o de si se utilizó el sensor *RFID* para apagar las luces.

Como medida de seguridad, cuando se abre la puerta principal, ese sensor de movimiento enviará los datos al servidor central para que le ordene a la cámara de seguridad que haga una foto de la persona que está entrando. Además, si se pasa el tiempo de seguridad establecido y la persona que ha abierto la puerta no ha usado su sensor biométrico para identificarse, se avisará a los dueños de la vivienda de que hay un posible intruso.

Por último, los dueños de la vivienda podrán acceder al estado de la vivienda en cualquier momento utilizando sus smartphones y sus sensores RFID para identificarse. Además de accedes se permitirá activar o desactivar los nodos de aplicación de forma remota de forma que, por ejemplo, al salir de trabajar puedan activar la calefacción para que la vivienda se haya calentado mientras se hacía el camino de vuelta.

A continuación voy a describir cada uno de los nodos que se han mostrado en la Figura 14:

- **Interruptores**: Estos dispositivos serán los responsables del encendido y del apagado de las luces. Se tienen que poder activar por tres vías, una vía es la manera tradicional, otra vía es mediante un lector RFID que identifique quien enciende y apaga la luz y la última vía es electrónicamente. De activar o desactivar los interruptores por esta vía se encargan los nodos de proceso. Los interruptores entran en la categoría de los nodos de aplicación, ya que utilizan los datos que les aporta el nodo de proceso para actuar. En la Figura 15 se muestra un ejemplo de interruptor que funciona con un sensor RFID.

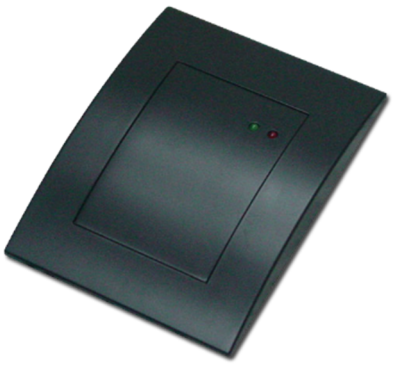


Figura 15: Interruptor Lector RFID

- **Sensores de movimiento y luminosidad**: Estos dispositivos serán los responsables detectar en si en una habitación hay alguien o está vacía, además, también informarán del nivel de luminosidad que hay en la misma. Cuando detectan movimiento envía un aviso al servidor a través del router en el que este estudia si tiene que encender o no las luces del techo enviándole la orden al interruptor. Estos sensores entran en la categoría de nodos finales ya que se encargan de obtener la información del entorno y comunicársela al nodo proceso para que la utilice según indiquen sus directivas. En la Figura 16 se muestra un ejemplo de un sensor de movimiento y luminosidad, en concreto, este sensor no puede comunicarse directamente con un router, por lo que habría que instalar un

nodo de proceso intermedio que traduzca la señal que envía el sensor de movimiento a una señal que sea interpretable por el servidor central.

Figura 16: Sensor de movimiento

- **Electroválvulas**: Estos dispositivos abrirán o cerraran el paso del agua caliente al radiador cuando el servidor central lo ordene. Tiene que haber uno en la entrada de agua de cada radiador para que de esta forma puedan controlarse individualmente y solo se active la calefacción en las zonas en las que hay personas. Cuando todas las electroválvulas están cerradas el servidor tiene que poner la caldera en modo *standBy* ya que no hay ningún motivo para que esté funcionando. Estos dispositivos entran dentro de la categoría de nodos de aplicación ya que no envían al servidor ningún dato y solo actúan cuando reciben la orden. Al tratarse de la calefacción, se le dará la oportunidad a los dueños de la vivienda de que activen las electroválvulas remotamente desde su Smart-Phone. En la Figura 17 se muestra un imagen de cómo es una electroválvula.

Figura 17: Electroválvula

- **Cámara de seguridad**: La cámara de seguridad tomara una fotografía o un video (dependiendo del modelo) cuando se produzca una entrada no autorizada en la vivienda. Para que funcione correctamente hay que coordinar un sensor de movimiento, un receptor RFID y el servidor para que cuando entre alguien por la puerta principal, si no se identifica, el servidor le dé la orden a la cámara de que actúe. Este dispositivo entra dentro de la categoría de nodos de aplicación ya que

se activa en función de los datos que recibe el servidor. En la Figura 18 se muestra una imagen de cómo sería una cámara de seguridad válida para este ejemplo.

Figura 18: Cámara de seguridad

- **Termómetros**: Estos dispositivos se instalarán cerca de las ventanas de cada habitación y serán los responsables de enviar al servidor información acerca de los grados que hay en la habitación. La información se puede enviar de manera continua o en intervalos de tiempo configurables. Dentro de la categorización de los nodos que he indicado para el IoT, estos dispositivos están categorizados como nodos finales, ya que su única responsabilidad es enviarle los datos al servidor central para que éste actúe como dicten las directrices configuradas. En la Figura 19 se muestra una imagen del aspecto que tendría un termómetro válido.

Figura 19: Sensor de temperatura

- **Router**: Este dispositivo será el responsable de comunicar cada sensor y cada actuador con el servidor. Todas las comunicaciones pasarán por él y además permitirá el acceso desde Internet a los clientes que estén debidamente autenticados para que puedan activar los dispositivos que crean oportunos. El router se encuentra categorizado como nodo de comunicación por las razones que he comentado en este párrafo. Es importante que el router disponga de conexión WiFi y de cable ya que no todos los sensores y dispositivos que se van a instalar disponen del mismo tipo de conexión. En la figura 20 se muestra una imagen de un router genérico que sería válido para el propósito de esta red.

Figura 20: Router

- **Caldera**: Este dispositivo es el responsable de calentar el agua que se utilizará en los radiadores de la vivienda cuando se active la calefacción. Tiene dos vías de utilización, por un lado, cuando todas las electroválvulas estén cerradas el servidor central tiene que ser capaz de apagarla para no malgastar energía. Por otro lado, no obstante, si se abre un grifo de agua caliente, aunque esté apagada tiene que ponerse en marcha y calentar el agua para enviarla al grifo que se ha abierto. Esto se consigue gracias a consigue gracias a un medidor de presión que detecta cuando se ha abierto un grifo. En la categoría de nodos del IoT la caldera pertenece a los nodos de aplicación ya que su único funcionamiento es calentar el agua si el servidor central ordena que se active la electroválvula de un radiador. En la Figura 21 se muestra una imagen de una caldera.

Figura 21: Caldera

- **Servidor**: Este dispositivo es el nodo de proceso más importante de la vivienda. A él llega información de todos los sensores que hay en la vivienda y de los sensores biométricos que portan las personas que viven en ella. Con toda esta información, el servidor decide que nodos de aplicación tiene que utilizar y de qué manera. El servidor tiene que tener una capacidad de proceso lo suficientemente elevada como para poder responder a todos las señales que lleguen de los nodos finales. Además de la capacidad de proceso tiene que tener software que permita activar o desactivar manualmente los nodos de aplicación y que permita acceder remotamente, ya sea desde otro equipo de la red o desde fuera de la red local. En la Figura 22 se muestra un ejemplo de cómo sería un servidor válido para el propósito de esta red.

Figura 22: Servidor

- **Lector sensor biométrico:** Durante toda esta sección he ido hablando de los sensores biométricos que portará el usuario. Para utilizarlos correctamente se instalarán en la vivienda varios lectores de los sensores para identificar a la persona de la vivienda que se está identificando. Estos lectores se categorizan dentro de nodos de proceso, ya que leen los datos del sensor biométrico que porta la persona que lo está utilizando y envía la orden al servidor central. En la Figura 23 se muestra un tipo de sensor biométrico, en concreto, un lector de huellas dactilares que servirá para identificar a la persona que entra en la vivienda. Pero no van a ser los únicos sensores biométricos que hay en la casa. Además del lector de las huellas dactilares, deberá de haber otros lectores que a distancia o utilizando algún nodo de comunicación intermedio sean capaces de enviar un aviso de emergencia médica en caso de que alguno de los que están viviendo en la casa necesite la ayuda de un médico.

Figura 23: Lector de huellas dactilares

- **Reloj con sensor biométrico:** Este dispositivo mide en tiempo real las constantes vitales de su portador y se las envía al servidor central. En caso de que se produzca una emergencia médica el servidor enviara una petición de auxilio al centro médico más cercano para que envíen asistencia médica. Este reloj se encuentra dentro del grupo de los nodos finales ya que su única función es la de transmitir información sobre las constantes vitales del portador, y es el servidor el encargado de comunicarse con el centro médico. En la Figura 24 se muestra un ejemplo de un reloj que mide las pulsaciones de la persona que lo lleva puesto.

Figura 24: Reloj biométrico

- **Smartphone/Aplicación móvil:** Este dispositivo lo utiliza un elevado número de personas todos los días y les permite están permanente conectado a su mundo virtual. Los smartphones permiten instalar aplicaciones y es gracias a estas aplicaciones que se permite que los dueños de las viviendas puedan controlar los dispositivos de aplicación de la vivienda. Este dispositivo se encuentra dentro de los nodos de aplicación, ya que permite controlar manualmente los dispositivos. Los smartphones tienen la limitación de la batería, lo que les obliga a conectarse cada pocas horas a la red eléctrica para recargarla. En la Figura 25 se muestra un ejemplo de cómo es un smartphone genérico.

Figura 25: Smartphone

Para finalizar este apartado, se muestra en la Figura 26 un diagrama de arquitectura con los distintos nodos de la simulación.

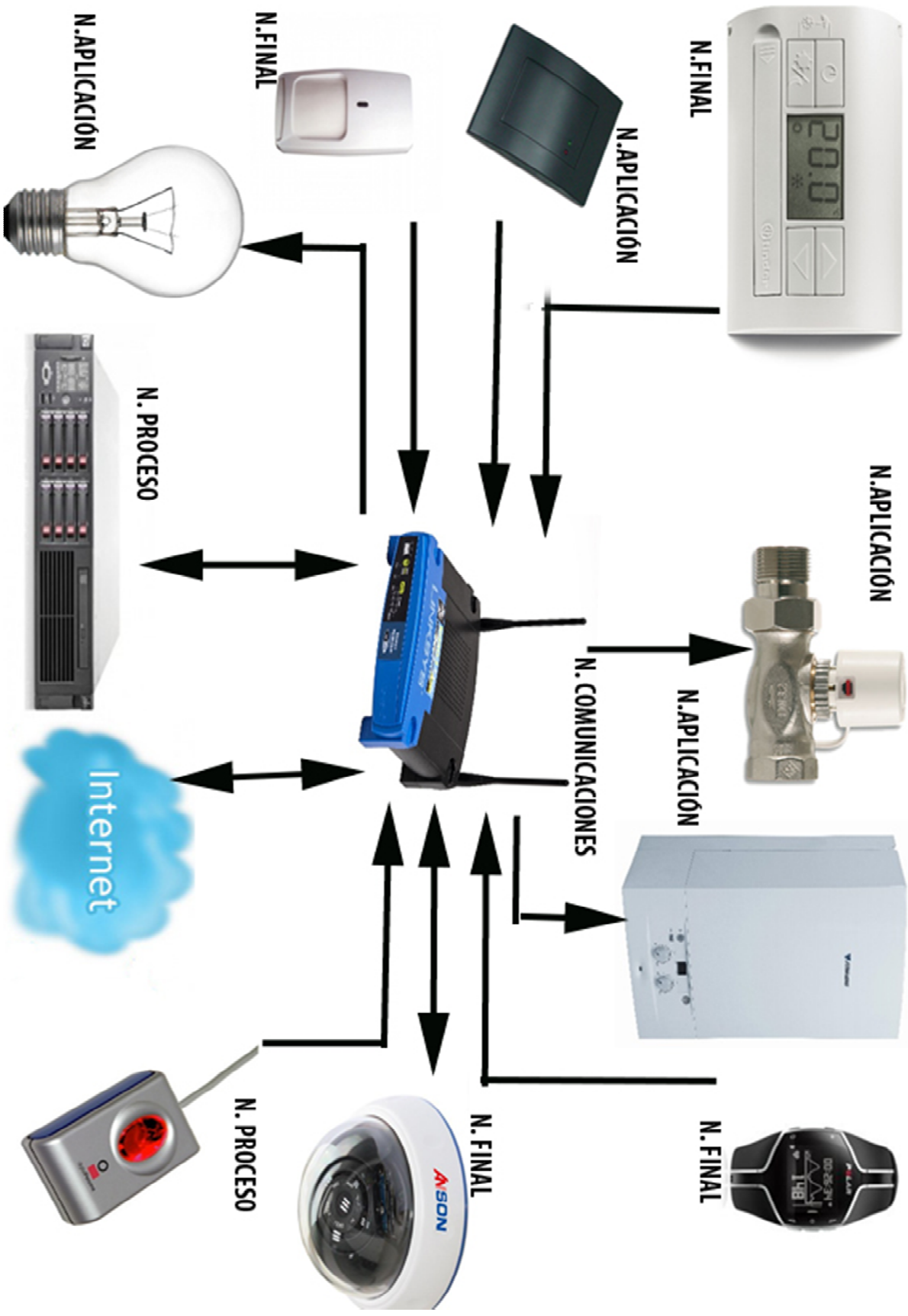


Figura 26: Diagrama de arquitectura con los nodos

4.2. Elementos de comunicación

En este apartado voy a hablar de los protocolos y elementos de comunicación que se implementarán en los distintos nodos. Los protocolos más utilizados dada la naturaleza de la red van a estar sobre *HTTP*, ya que la mayoría de ellos utilizan los servicios web[20]. Los estándares empleados en los servicios web son: *XML*, *SOAP*, *WSDL*, *UDDI*. Además, también se va a utilizar *L2CAP* [17] [18] para la comunicación por bluetooth.

4.2.1. L2CAP

La especificación del control de enlace lógico y el protocolo de la capa de adaptación (conocido como *L2CAP* por sus siglas en inglés) proporciona servicios orientados a conexión y sin conexión a las capas de protocolos superiores con protocolos de multiplexado, segmentación y operaciones de re ensamblado. Este protocolo permite enviar y recibir paquetes de hasta 64Kb de longitud. En la Figura 27 se muestra un ejemplo de dónde se situaría el protocolo *L2CAP* en relación con los otros protocolos.

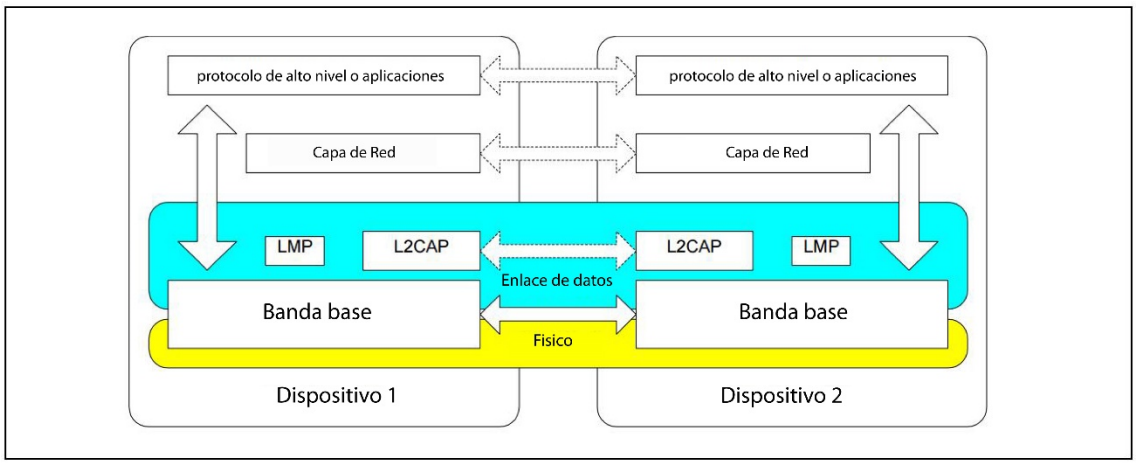


Figura 27: Protocolo L2CAP

El estándar bluetooth define dos tipos de comunicación, síncrona y orientada a conexión (*SCO* por sus siglas en inglés) y asíncrona y no orientada a conexión (*ACL* por sus siglas en inglés. *L2CAP* está definido para usar solamente conexión asíncrona y no soporta conexión síncrona. En los enlaces *ACL* no se permite utilizar paquetes de tipo *AUX1*, esto se debe a que este paquete no soporta ninguna comprobación de integridad de datos, que va en contra del propio *L2CAP* que utiliza comprobaciones de integridad en la banda base para proteger la información transmitida.

La cabecera de los paquetes que se envían en la comunicación utilizando el protocolo *L2CAP* son iguales de un solo segmento o en los paquetes multisegmento, la única diferencia está en que los paquetes multisegmento indican el tipo de paquete que es. En la Figura 28 se muestra un ejemplo de paquetes multisegmento y se indica la diferencia con los paquetes de un solo segmento.

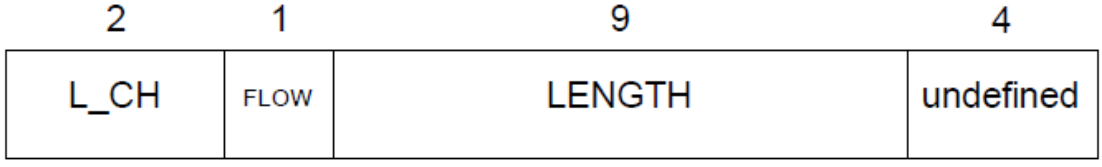


Figura 28: Paquetes multisegmento

Los bytes del campo *L_CH* indican si se está utilizando o no *L2CAP* y puede tomar los valores 10 ó 01 dependiendo de si está iniciando un paquete *L2CAP* o es la continuación de uno anterior. El byte del campo *flow* tomará el valor 0 ó 1 en función de si se van a enviar más paquetes (1) o si por el contrario se el paquete actual es el último que se envía y se deja libre la banda base (0). El campo *Length* indica la longitud de todo el paquete en binario. Por último, el campo *undefined* que solo está presente si el paquete es multisegmento, indica el tipo de paquete que se está enviando.

Los requisitos funcionales para *L2CAP* incluyen protocolos de multiplexado, segmentación y re ensamblado, y administración de grupos. *L2CAP* está sobre el protocolo de banda base y ofrece interfaces a otros protocolos de comunicación como el protocolo del servicio de descubrimiento de bluetooth, y control de telefonía. Aunque se puede utilizar *L2CAP* con fines como la telefonía IP, no es recomendable puesto que o hay control de calidad ya que esa característica pertenece a las comunicaciones síncronas y *L2CAP* no está diseñado para ese tipo de comunicaciones. Además, los requisitos funcionales para la implementación de *L2CAP* deben incluir la simplicidad en el funcionamiento y un bajo coste operativo, ya que las implementaciones deben ser aplicables para dispositivos con recursos computacionales restringidos. *L2CAP* no debe consumir excesiva energía para que no le afecte a la eficiencia energética del dispositivo y además es recomendable mantener el consumo de memoria lo más bajo posible. Por último, la implementación debe ser diseñada para alcanzar la máxima eficiencia posible en el uso del ancho de banda.

A continuación, explico brevemente la causa de cada uno de los requisitos que se han comentado hasta ahora:

- **Protocolo de multiplexado**: *L2CAP* debe soportar un protocolo de multiplexado debido a que el protocolo de banda base no soporta cualquier tipo de identificador indicado en un protocolo que esté en la capa de protocolos a un nivel superior a *L2CAP*.
- **Segmentación y re ensamblado:** Los paquetes definidos en el protocolo de banda base están limitados en el tamaño. Enviar el número máximo de unidades de transmisión asociados con la carga más grande del protocolo banda base, limita el uso eficiente del ancho de banda para la capa superior. Los paquetes *L2CAP* grandes deben segmentarse en múltiples paquetes de banda base más pequeños antes de su transmisión. De manera inversa, cuando se reciben múltiples paquetes de banda base se tiene que poder montar en un solo paquete *L2CAP*. La funcionalidad de segmentación y re ensamblado es necesaria para los protocolos de apoyo que usan paquetes más grandes que los admitidos por la banda base.
- **Calidad del servicio:** La conexión *L2CAP* establece procesos que permiten el intercambio de información relativa a la calidad del servicio esperada entre dos dispositivos bluetooth. Cada implementación de *L2CAP* debe monitorizar los recursos usados por el protocolo y garantizar que la calidad del servicio es la esperada.
- **Grupos:** Muchos protocolos tiene el concepto de grupos de direcciones. El protocolo de banda base soporte el concepto de piconet, un grupo de dispositivos síncronos que trabajan juntos. La abstracción de los grupos de *L2CAP* permite a la implementaciones asignen eficientemente los grupos de protocolos a las

piconets. Sin una abstracción de grupo, los protocolos superiores se estarían exponiendo al protocolo de la banda base directamente.

A pesar de todas las características comentadas, hay algunas que no están dentro del ámbito de responsabilidades de L2CAP:

- *L2CAP* no transporta audio, ya que está designado para enlaces síncronos.
- *L2CAP* no garantiza la integridad de los datos, no hace retransmisiones o cálculos de checksum.
- *L2CAP* no soporta envío multicast.
- *L2CAP* no soporta el concepto de grupo global.

L2CAP está basado en el concepto de *canales.* Cada uno de los puntos finales de un canal *L2CAP* se conoce como *identificador de canal.* Estos identificadores de canal son nombres que representa un punto final del dispositivo. No pueden tomar cualquier valor ya que algunos de ellos están reservados, y otros son ilegales, como por ejemplo asignar el valor 0x0000. Las implementaciones pueden gestionar su asignación de cualquier manera, con la única restricción de que en la misma red, no se reutiliza el mismo identificador de canal en dos dispositivos diferentes.

EL funcionamiento de *L2CAP* orientado a conexión puede ser visto como una máquina de estados. Para realizar la máquina de estados hay que definir los eventos y las acciones que pueden ocurrir usando *L2CAP* y cómo interactúan los eventos con las distintas capas. En la Figura 29 se muestra la interacción de las distintas capas de protocolos.

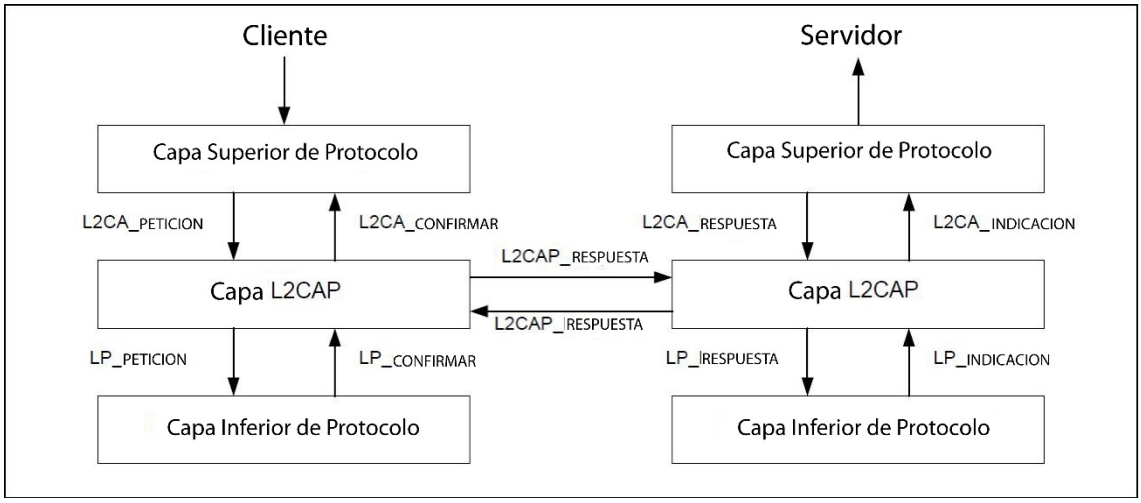


Figura 29: Interacciones entre las capas de protocolos

El funcionamiento de una comunicación es fácil de explicar. La aplicación cliente tiene que poder actuar como cliente y como servidor, esto le permite iniciar y aceptar peticiones según el rol que esté desempeñando. Las interfaces entre las capas son las responsables de adaptar los paquetes para que el protocolo de la capa inferior o superior lo entienda.

Para crear la máquina de estados se ha dicho que hay que definir eventos y acciones. Los eventos son todos los mensajes entrantes a la capa del *L2CAP*, que se dividen en dos categorías: Son indicaciones o confirmaciones si vienen de las capas inferiores o son peticiones y respuestas si vienen de las capas superiores. Las Acciones también pueden ser indicaciones y confirmaciones si vienen de capas superiores o pueden ser peticiones

y respuestas si vienen de las capas inferiores. En la Figura 30 se muestra la máquina de estados de *L2CAP*.

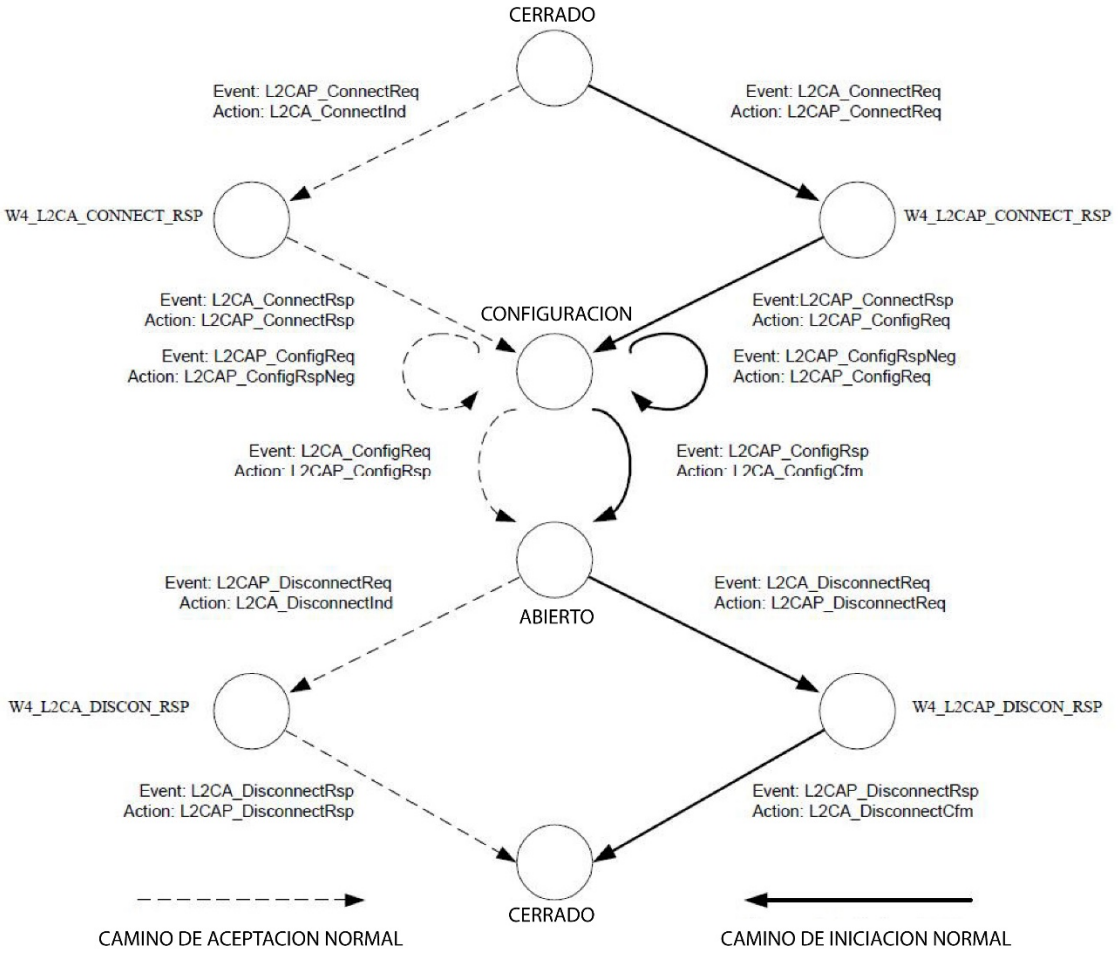


Figura 30: Maquina de estados

4.2.2. XML

XML es el estándar de *Extensible Markup Language*. No es más que un conjunto de reglas para definir etiquetas semánticas que organizan un documento en diferentes partes. *XML* es un metalenguaje que define la sintaxis utilizada para definir otros lenguajes de etiquetas estructurados. Es válido para la representación digital de documentos de cualquier tipo y con cualquier extensión.

Los documentos *XML* tienen que estar estructurados. Esto implica que cada elemento que se defina en el documento *XML* tiene que tener su etiqueta de inicio y su etiqueta de finalización. El usuario tiene total libertad para añadir o quitar elementos, siempre que mantenga la estructura definida en el estándar y respete que un elemento tiene que abrirse y cerrarse con el mismo elemento padre. En la Figura 31 se muestra un ejemplo de cómo sería el código *XML* utilizado para llamar a un servicio web.

```
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope"
   <soap:Header/>
   <soap:Body>
      <icof:registroCliente>
         <identificador>?</identificador>
         <nombre>?</nombre>
      </icof:registroCliente>
   </soap:Body>
</soap:Envelope>
```

Figura 31: Fragmento código XML

Este metalenguaje presenta una serie de características que favorecen su utilización en los servicios web. Las características más destacadas se resumen a continuación:

- No depende de versiones para funcionar en navegadores. Los identificadores se crean de manera simple en el propio elemento y se pueden validar por medio de un validador de documentos.
- Integración de datos de diferentes fuentes. Se pueden enviar y recibir documentos de aplicaciones locales como remotas.
- La extensibilidad del lenguaje permite agrupar información de una gran variedad de aplicaciones, independientemente de si son páginas web o bases de datos.
- Fácil manipulación de los datos.
- La estructuración de la información hace que sea más accesible a los buscadores, lo que favorece que las respuestas de los motores de búsquedas.
- Permite la manipulación de los datos desde un cliente web.
- Los clientes web pueden ser más autónomos para desarrollar tareas que actualmente se ejecutan en el servidor.

El metalenguaje *XML* consta de cuatro especificaciones, es el propio estándar el que sienta las bases sintácticas y el alcance de su implementación:

- **DTD:** Definición de tipo de documento. Este archivo contiene la definición formal del documento y la estructura lógica del mismo. Además, define los elementos de página y los atributos. Debido a que su inclusión es opcional los documentos *XML* se categorizan como *bien formado* si no lleva *DTD* o *validado* si incluye el documento.
- **XLS (eXtensible Stylesheet Language):** Define o implementa el lenguaje de estilo de los documentos escritos para *XML*. Permite modificar el aspecto de un documento. Está basado en el lenguaje de semántica y especificación de estilo del documento y debido a todo lo que este estándar permite realizar es más potente que el *CSS*.
- **XLL (eXtensible Linking Language):** Define el modo de enlace entre diferentes enlaces. Va más allá de los enlaces simples que soporta *HTML*. Soporta enlaces extendidos que tienen características de bidireccionalidad, múltiples o agrupados.
- **XUA (XML User Agent):** Estandarización de navegadores *XML*. Se aplicará a los navegadores para que compartan todas las especificaciones del *XML*.

Con todo lo explicado, parece lógico que se utilice el lenguaje *XML* en los servicios web ya que al ser un estándar abierto y reconocido por muchas tecnológicas hace que muchas aplicaciones web y de escritorio sean compatibles. Además su simplicidad de sintaxis hace que sea muy fácil de escribir y de entender lo que beneficia su utilización y

depuración en servicios web. Por último, *XML* también permite abstraerse del protocolo de transporte, ya que al ser un lenguaje de marcado de texto solo se necesita un protocolo que sea capaz de enviar texto, aunque habitualmente suele utilizarse *HTTP* o su variante segura *HTTPS*.

Los elementos de XML también pueden tener atributos y valores. La posibilidad de definir atributos y elementos permite crear estructuras de datos que serán interpretadas por el servidor y por el cliente del servicio web a la hora de enviar estructuras complejas de datos. Transformar un documento genérico *XML* en datos de una aplicación o en datos específicos del dominio es un aspecto esencial en los servicios web.

La sintaxis usada en las tecnologías de los servicios web específica como los datos se representan genéricamente, define como y con qué calidad del servicio se transmiten los datos y como los servicios son publicados y descubiertos. Hay dos categorías de uso del *XML* en los servicios web, por un lado la representación del formato y por otro lado la especificación de como la aplicación utiliza los datos.

Cuando se reciben datos de un archivo, o de una base de datos, los datos pueden ser convertidos en *XML* usando la información contenido en el esquema. Cuando los documentos *XML* son enviados el esquema puede validar que los datos y los tipos de datos incluidos en el documento son correctos.

En el desarrollo de software tradicional define sus propios tipos de datos y estructuras, pero tiene el problema de que las aplicaciones tienen que comunicarse con otras aplicaciones, para ello se pueden utilizar numerosas librerías *XML* que transforman los datos en documentos *XML* y viceversa. Las librerías más comunes son *DOM* y *SAX*.

La mayor diferencia entre *SAX* y *DOM* es que la API de *DOM* proporciona un modelo genérico de objetos para representar la estructura de documentos y un conjunto de interfaces y estándares para manipularlos. Por el contrario, *SAX* trabaja con eventos, analizando el documento elemento a elemento. La API de *SAX* usa menos memoria y es más eficiente para mensajería y trasformación.

El crecimiento del número de documentos *XML* hace que se necesita un elemento específico para identificar el ámbito del documento. Este elemento es conocido como *namespace*. Los espacios de nombres en *XML* crean prefijos únicos para elementos en documentos separados que son usados juntos. Su uso primario es el de evitar problemas que pueden ser causados cuando el mismo elemento aparece varias veces en documentos relacionados. En los servicios web esto es muy común porque a menudo involucran múltiples documentos *XML* a la vez.

Por último, las instancias de los documentos pueden tener múltiples esquemas asociados, esto provoca la necesidad de crear hojas de transformación para transformar el formato de un esquema a otro. Normalmente se utiliza el estándar *Lenguaje Extensible de Estilos: Transformaciones* (*XSLT* por sus siglas en inglés). La transformación de documentos permite a las instancias transformase del formato específico de una aplicación a otro distinto. Lógicamente, esto significa que el contenido de los documentos *XML* es compatible entre las aplicaciones. En la Figura 32 se muestra un ejemplo de la funcionalidad de las hojas de transformación.

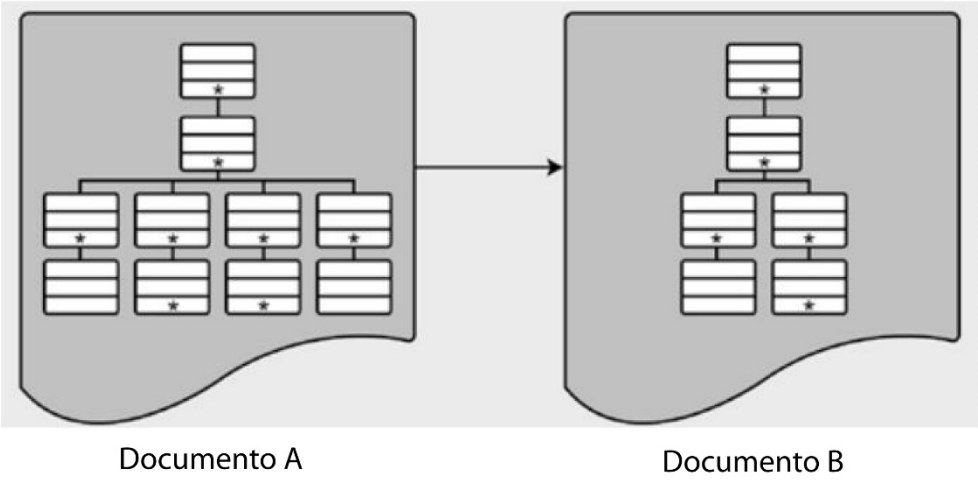


Figura 32: Ejemplo de uso de XSLT

4.2.3. SOAP

El *Protocolo Simple de Acceso a Objetos* (*SOAP* por sus siglas en inglés) está diseñado para la comunicación entre computadores mediante el paso de mensajes. La comunicación se realiza enviando los mensajes en *XML*, describiendo los atributos y los valores de los mismos. *SOAP* especifica tres ítems diferentes: el modelo de paquetes, el mecanismo de serialización y el mecanismo *RPC*. En esta subsección, explicaré cómo funciona *SOAP* y sus principales características.

Un mensaje *SOAP* es una transmisión de un solo sentido que va desde el cliente al servidor. Esta característica presenta el inconveniente de que cuando se utiliza para mensajes *RPCs*, se espera una respuesta en un periodo corto de tiempo, por esta razón, cada vez es más habitual utilizar una implementación de *SOAP* que utilice un patrón de petición y respuesta.

Un mensaje *SOAP* está formado por la cabecera y el cuerpo del mensaje. Ambas cosas juntas se conocen como el envoltorio del mensaje. Este envoltorio debe cumplir las siguientes reglas gramaticales:

- El elemento siempre se llamará *Envelope*.
- Este elemento contendrá todo el mensaje *SOAP*.
- Este elemento puede contener declaraciones del espacio de nombres y atributos adicionales. Todos los atributos contenidos deben especificar su espacio de nombres.

Es en el envoltorio donde se especificará la versión de *SOAP* que se está utilizando. Habitualmente, en el mundo de la informática, se números para identificar versiones, *SOAP* no es así y utiliza un elemento para asociarlo con la *URL* del espacio de nombres de la versión que se vaya a utilizar. Cuando es posible, la aplicación *SOAP* debe responder a la aplicación que ha realizado la llamada con un código de respuesta en el elemento *VersionMismatch*.

En un mensaje *SOAP* la cabecera es voluntaria. Esta parte del mensaje proporciona mecanismos para añadir información sobre el mensaje al mensaje. Cuando se incluye,

este elemento es el primer hijo del elemento *Envelope*. El uso de la cabecera es habitual cuando se quiere realizar autentificaciones o en cuando el protocolo de transporte no tiene mecanismo de respuesta, se puede utilizar el elemento *Header* para simularla. En la Figura 33 se muestra un ejemplo de cómo es un elemento *Header*.

```
<soapenv:Header>
    <wsse:Security soapenv:mustUnderstand="1"
        xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd">
        <wsse:BinarySecurityToken
            EncodingType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#Base64Binary"
            ValueType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509v3"
            wsu:Id="SecurityToken-60498088-e311-4f16-8a91-6c95447cd3e7"
            xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd">MIIHNjCCBh6gAwIBAgIQZEmsJhYl4X9RFOZUqkgOOzANBgk
    </wsse:Security>
    <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
        <SignedInfo>
            <CanonicalizationMethod Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#" />
            <SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1" />
            <Reference URI="#MsgBody">
                <Transforms>
                    <Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#" />
                </Transforms>
                <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
                <DigestValue>4ckx/GszEOYNgsb2Xs+OVDmN3JE=</DigestValue>
            </Reference>
        </SignedInfo>
        <SignatureValue>H/Um4tEHrigANV0L+G0TxeAh3SNsiw6buitX0W6YYtoz36IGH9JnX6cZCX1fEozafY97wr/0mi7H
            ByNlN8afltZ1sSCPrKcfQ4pjbHC7SLSLtFXuoVH+JnHjaaxHDn3GxwQIEn+nxYLSbMdCpbOTWqcB
            m/94LlTNpGHHSHebVv8=</SignatureValue>
        <KeyInfo>

            <KeyValue>
                <RSAKeyValue>
                    <Modulus>rbZPiTDGrWkoaZO2QMPGKfPBClCnU8CTCns/QN4FDO726qesvdCPp0AB2C+2hHbUUaXlmWoBGTsS
                        0Vu9ST8ZDrZmA3biVv/pMU0+GYKBuB9ugLhTSEP7zZrrMi5LAWtMLNVgOuKdGoM302toI64rLkUq
                        69YUBlgYxJ+rR/nrD68=</Modulus>
                    <Exponent>AQAB</Exponent>
                </RSAKeyValue>
            </KeyValue>
        </KeyInfo>
    </Signature>
</soapenv:Header>
```

Figura 33: Ejemplo de cabecera del mensaje SOAP

El cuerpo del mensaje, cuyo elemento se llama *Body*, es el responsable de transportar la información útil para la aplicación. Aunque originalmente este elemento se usaba para concentrar las llamadas *RCP* y devolver la información del error, actualmente se utiliza para pasar otros tipos de datos. En la Figura 34 muestro un ejemplo de cómo sería el cuerpo del mensaje *SOAP*.

```
<soapenv:Body>
   <itv:comprarTasasPorNRC>
      <arg0>
         <aplicacion>MASI</aplicacion>
         <identificador>12345678Z</identificador>
         <importe>9580</importe>
         <nrc>791502000000212345678 9</nrc>
         <entidad>2104</entidad>
         <fechaIngreso>12/12/2015</fechaIngreso>
         <tasasSolicitadas>
            <cantidad>1</cantidad>
            <grupo>1</grupo>
            <tipo>1.1</tipo>
         </tasasSolicitadas>
         <tipoIdentificador>NIF</tipoIdentificador>
      </arg0>
   </itv:comprarTasasPorNRC>
</soapenv:Body>
```

Figura 34: Cuerpo mensaje SOAP

Por último, el elemento *Envelope* puede tener un hijo etiquetado como *Fault*. Este elemento se utiliza para enviar el error en un mensaje. Se recomienda usar este elemento solo en el mensaje de respuesta. Este elemento solo puede aparecer una vez, además tiene cuatro elementos secundarios que pueden utilizarse para ampliar la información:

- **faultcode:** Todos los elementos *Fault* tienen que tener un elemento de este tipo. Puede usarse este código para proporcionar información sobre el tipo de error que se ha producido.
- **faultstring:** Este elemento guarda información en texto plano sobre la causa del error. Al igual que el elemento anterior, todos los elementos *Fault* tienen que tener un elemento hijo de este tipo. No es recomendable usarlo para poner siempre el mismo mensaje de error, ya que, aunque cumpliría con las especificaciones, violaría el objetivo con el que fue creado.
- **faultactor:** Este elemento indica que uno de las aplicaciones que utilizan SOAP escribió el elemento *Fault*. Al crearlo, una aplicación solo necesita incluir este elemento si la aplicación es un intermediario y no es el destino final del mensaje. La aplicación que reciba el mensaje tiene la opción de escribir o no este elemento hacia fuera, indicando la *URI* del origen del fallo en caso de que lo envíe.
- **detail:** Este elemento lleva la información de error específico de la aplicación relacionado con el cuerpo del mensaje. El elemento *detail* se añade cuando la aplicación no puede procesar los datos del elemento *body*, en caso de que el error se produzca en el elemento *header* no se puede utilizar este elemento. Gracias a esta regla, se diferencia si el problema ocurrió en el *body* o en otros lugares.

En la Figura 35 se muestra un ejemplo de cómo sería un elemento *Fault.*

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
   <soapenv:Body>
      <soapenv:Fault>
         <faultcode>soapenv:Server</faultcode>
         <faultstring>Se ha producido un problema interno en la aplicación.
          Por favor, vuelva a intentarlo en unos minutos.
          Si el problema continúa, avise a un administrador.</faultstring>
          <faultactor>http</faultactor>
          <detail/>
      </soapenv:Fault>
   </soapenv:Body>
</soapenv:Envelope>
```

Figura 35: Elemento Fault

4.2.4. WSDL

El *Lenguaje de Descripción de Servicios Web* (*WSDL* por sus siglas en inglés) es un protocolo basado en *XML* que describe los accesos al servicio web. Es un lenguaje propuesto por el *W3C* para la descripción de servicios web que describe la interfaz del servicio en formato *XML*. La principal ventaja que ofrece es que aísla los detalles de la interfaz del protocolo sobre el que se transporta la información. En la Figura 36 se muestra un esquema de cómo funciona *WSDL* en una petición al servicio web.

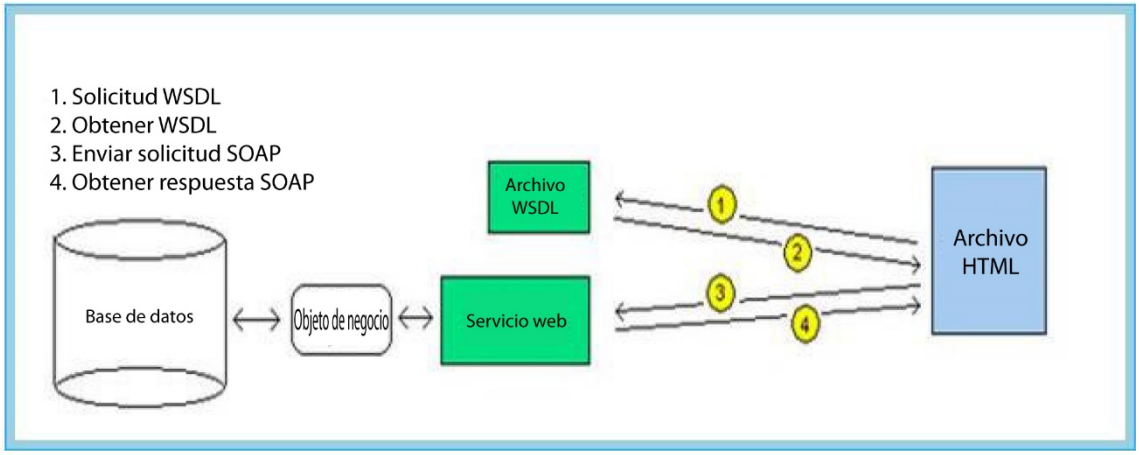


Figura 36: Ejemplo de uso de WSDL

En la figura anterior se describe como lo primero que hace el cliente es obtener el fichero *WSDL*, y una vez que ha estudiado los métodos y los parámetros del servicio web, está preparado para utilizarlo, siendo este último el responsable de enlazar con la parte de negocio de la aplicación y devolver la respuesta. En la Figura 37 muestro un ejemplo de un documento *WSDL* que describe un servicio web con un solo método.

```
<wsdl:definitions xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    <wsdl:types>
        <schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" xm
            <xs:element name="updateLocalizacion" type="tns:updateLocalizacion" />
            <xs:element name="updateLocalizacionResponse" type="tns:updateLocalizacionResponse" />
            <xs:complexType name="updateLocalizacion">
                <xs:sequence>
                    <xs:element minOccurs="0" name="arg0" type="xs:string" />
                    <xs:element minOccurs="0" name="arg1" type="xs:string" />
                    <xs:element minOccurs="0" name="arg2" type="xs:string" />
                </xs:sequence>
            </xs:complexType>
            <xs:complexType name="updateLocalizacionResponse">
                <xs:sequence>
                    <xs:element minOccurs="0" name="return" type="xs:string" />
                </xs:sequence>
            </xs:complexType>
        </schema>
    </wsdl:types>
    <wsdl:message name="updateLocalizacionResponse">
        <wsdl:part element="tns:updateLocalizacionResponse" name="parameters"/>
    </wsdl:message>
    <wsdl:message name="updateLocalizacion">
        <wsdl:part element="tns:updateLocalizacion" name="parameters"/>
    </wsdl:message>
    <wsdl:portType name="WS_iCofrade">
        <wsdl:operation name="updateLocalizacion">
            <wsdl:input message="tns:updateLocalizacion" name="updateLocalizacion"/>
            <wsdl:output message="tns:updateLocalizacionResponse" name="updateLocalizacionResponse"/>
        </wsdl:operation>
    </wsdl:portType>
    <wsdl:binding name="WS_iCofradeServiceSoapBinding" type="tns:WS_iCofrade">
        <soap12:binding style="document" transport="http://schemas.xmlsoap.org/soap/http" />
        <wsdl:operation name="updateLocalizacion">
            <soap12:operation soapAction="urn:UpdateLocalizacion" style="document" />
            <wsdl:input name="updateLocalizacion">
                <soap12:body use="literal" />
            </wsdl:input>
            <wsdl:output name="updateLocalizacionResponse">
                <soap12:body use="literal" />
            </wsdl:output>
        </wsdl:operation>
    </wsdl:binding>
    <wsdl:service name="WS_iCofradeService">
        <wsdl:port binding="tns:WS_iCofradeServiceSoapBinding" name="WS_iCofradePort">
            <soap12:address location="http://porquenohacerlo.es:8080/WS_iCofrade/services/WS_iCofradePort" />
        </wsdl:port>
    </wsdl:service>
</wsdl:definitions>
```

Figura 37: Ejemplo documento WSDL

El protocolo *WSDL* afronta la necesidad de describir las comunicaciones de forma estandarizada, definiendo una gramática *XML* que describe los servicios de red como colecciones de puntos finales de comunicación capaces de intercambiar mensajes. Las definiciones de servicio *WSDL* proporcionan documentación para sistemas distribuidos y sirven como fórmula para automatizar los detalles de la comunicación entre aplicaciones.

Los documentos *WSDL* definen los servicios como colecciones de puntos finales de red o puertos. En *WSDL*, la definición abstracta de puntos finales y de mensajes se separa de la instalación concreta de red o de los enlaces del formato de datos. Esto permite la reutilización de definiciones abstractas: mensajes, que son descripciones abstractas de los datos que se están intercambiando y tipos de puertos, que son colecciones abstractas de operaciones. Las especificaciones concretas del protocolo y del formato de datos para un tipo de puerto determinado constituyen un enlace reutilizable.

Los elementos más importantes del documento *WSDL* son:

- **definitios:** Este elemento está al comienzo del documento y engloba todos los elementos del mismo. Además los espacios de nombres que podrán utilizar todos los elementos hijos.

- **types:** Define los tipos de datos usados en los mensajes. Por diseño del documento, estos elementos pueden ser llevados a un documento *XML* auxiliar conocido como *XML Schema Definition (XSD)*. Si se genera el documento *XSD* habrá que importarlo para poder utilizarlo.
- **message:** Define los métodos y parámetros para realizar la operación. Cada mensaje puede tener varios parámetros o ninguno. Si los tipos de los parámetros no son primitivos, estos tendrán que estar definidos dentro de *types*.
- **portType:** Este elemento define las operaciones que se pueden realizar y los mensajes que involucra. Es aquí donde se define el mensaje de petición y de respuesta.
- **binding:** Se define el formato del mensaje y los detalles del protocolo para cada operación.
- **service:** Define los puertos soportados por el servicio web. Para cada uno de los protocolos aceptados tiene que haber un puerto definiéndolo.

4.2.5. UDDI

El servicio de Descripción, Descubrimiento Universal e Integración (*UDDI* por sus siglas en inglés) es un directorio de servicios web que ofrece información general del proveedor del servicio, su clasificación taxonómica e información técnica del servicio.

UDDI se creó con dos objetivos, por una lado se pretendía que fuera una especie de registro de empresas y que fuera utilizado por estas siguiendo uno de los tres métodos que se indican a continuación:

- **Páginas blancas:** Se utilizan para buscar información acerca de la empresa.
- **Páginas amarillas:** *UDDI* especifica varias taxonomías que las empresas podrían utilizar para clasificarse a sí mismas.
- **Paginas verdes:** Las empresas pueden utilizar este método de búsqueda para encontrar socios comerciales que habían implementado un servicio particular.

El otro objetivo por el que fue diseñado *UDDI* es para mantener la ejecución de aplicaciones distribuidas a largo plazo, de manera que si una empresa propietaria de una aplicación cierra, esta aplicación pueda seguir ejecutándose en otros servidores.

La información contenida en un registro *UDDI* contiene cinco tipos diferentes:

- **businessEntity:** Es la empresa real o una filial de la misma.
- **publisherAssertion:** Es la relación entre varias empresas reales. Deben ser creadas por todas las partes implicadas para ser válidas.
- **bindingTemplate:** Es la especificación de una interfaz del servicio.
- **businessService:** Es el servicio provisto por un negocio. En el mundo del *UDDI* no tiene por qué ser un servicio web, aunque para la finalidad de este trabajo se utilizaran los que ofrecen servicios web.
- **tModels:** Modelos de metadatos. Es un punto de ayuda para los documentos *XML*, tienen como finalidad especificar información acerca de algo vía un servicio, negocio o algo más.

El registro en *UDDI* no suele ser complicado, generalmente solo contiene un nombre y un identificador único que actúa como clave primaria de búsqueda. En la Figura 38 se muestra un ejemplo del código *XML* de un elemento *businessEntity*.

```
<businessEntity businessKey="uuid:C0E6D5A8-C446-4f01-99DA-70E212685A40"
   operator="http://www.ibm.com" authorizedName="John Doe">
   <name>Acme Company</name>
   <description>
      We create cool Web services
   </description>

   <contacts>
      <contact useType="general info">
         <description>General Information</description>
         <personName>John Doe</personName>
         <phone>(123) 123-1234</phone>
         <email>jdoe@acme.com</email>
      </contact>
   </contacts>

   <businessServices>
      ...
   </businessServices>
   <identifierBag>
      <keyedReference tModelKey="UUID:8609C81E-EE1F-4D5A-B202-3EB13AD01823"
   </identifierBag>

   <categoryBag>
      <keyedReference tModelKey="UUID:C0B9FE13-179F-413D-8A5B-5004DB8E5BB2"
   </categoryBag>
</businessEntity>
```

Figura 38: Ejemplo de businessEntity

El elemento *businessEntity* contiene el nombre de la compañía y de manera opcional una descripción y una lista con información de contacto. El elemento *businessService* enumera todos los servicios que han sido asociados con el elemento *businessEntity*. Los dos últimos elementos *identifierBag* y *categoryBag* añaden información adicional sobre la empresa y se utilizan para realizar búsquedas más eficientes y mejorando los resultados de las mismas.

El elemento *publisherAssertion* tiene tres elementos fundamentales que definen la relación entre dos empresas. Los elementos *fromKey* y *toKey* identifican de forma única a las empresas involucradas en el acuerdo. El elemento *keyedReference* tiene como único atributo obligatorio el *keyValue* que es la clave de la relación. En la Figura 39 se muestra un ejemplo de cómo sería el elemento *publisherAssertion*.

```
<element name="publisherAssertion" type="uddi:publisherAssertion" />
<complexType name="publisherAssertion">
   <sequence>
      <element ref="uddi:fromKey" />
      <element ref="uddi:toKey" />
      <element ref="uddi:keyedReference" />
   </sequence>
</complexType>
```

Figura 39: Ejemplo de publisherAssertion

El elemento businessService tiene dos tipos de información principales además de la *serviceKey* única y el nombre. Esta estructura representa un servicio web individual proporcionado por la empresa. Su descripción incluye información sobre cómo acceder al servicio web, que tipo de servicio web es y en que categoría taxonómica se engloba. En la Figura 40 se muestra un ejemplo de un elemento de tipo businessService.

```
<businessService serviceKey="uuid:D6F1B765-BDB3-4837-828D-8284301E5A2A"
   businessKey="uuid:C0E6D5A8-C446-4f01-99DA-70E212685A40">

   <name>Hello World Web Service</name>
   <description>A friendly Web service</description>
   <bindingTemplates>
      ...
   </bindingTemplates>
   <categoryBag />
</businessService
```

Figura 40: Ejemplo de elemento businessService

El elemento *bindingTemplate* es lo que define tanto la ubicación donde se puede encontrar el servicio y lo que hace. Para determinar donde se puede acceder a un servicio se proporciona un punto de acceso, que puede ser una *URL*. Expresar la que realizar un servicio es más complicado ya que no solo se puede ofrecer servicios web y además, los servicios web que se ofrecen debe especificarse de forma clara y sin ambigüedades. El resultado es uqe la *bindingTemplate* incluye un elemento denominado *tModelInstanceDetails* que contiene uno o más elementos denominados t*ModelInstanceInfo* que en última instancia contiene una *URL* que define el servicio, es decir, la *URL* del *WSDL* del servicio web. En la Figura 41 se muestra un ejemplo de un elemento *bindingTemplate*.

```
<bindingTemplate serviceKey="uuid:D6F1B765-BDB3-4837-828D-8284301E5A2A"
   bindingKey="uuid:C0E6D5A8-C446-4f01-99DA-70E212685A40">

   <description>Hello World SOAP Binding</description>
   <accessPoint URLType="http">http://localhost:8080</accessPoint>
   <tModelInstanceDetails>
      <tModelInstanceInfo tModelKey="uuid:EB1B645F-CF2F-491f-811A-4868705F5904">
         <instanceDetails>
            <overviewDoc>
               <description>
                  references the description of the WSDL service definition
               </description>

               <overviewURL>
                  http://localhost/helloworld.wsdl
               </overviewURL>
            </overviewDoc>
         </instanceDetails>
      </tModelInstanceInfo>
   </tModelInstanceDetails>
</bindingTemplate>
```

Figura 41: Elemento bindingTemplate

El elemento *tModels* tiene dos propósitos, por un lado puede ser un indicador de espacio de nombres, es decir ayuda a diferenciar entre los tipos de información que se pueden confundir fácilmente. Por otro lado, *tModel* puede actuar como un identificador de especificaciones. Cada elemento, posee una clave única que constituye la única información requerida. No obstante, en este caso también se puede especificar el elemento *overviewURL* correspondiente al *WSDL* de la interfaz. En la Figura 42 se muestra un ejemplo de cómo sería un elemento *tModel*.

```
<tModel tModelKey="uuid:xyz987..." operator="http://www.ibm.com"
   <name>HelloWorldInterface Port Type</name>
   <description>
      An interface for a friendly Web service
   </description>

   <overviewDoc>
      <overviewURL>
         http://localhost/helloworld.wsdl
      </overviewURL>
   </overviewDoc>
</tModel>
```

Figura 42: Elemento tModel

5. Simulación de funcionamiento

En este apartado voy a simular el funcionamiento de la red de nodos definidos en la sección anterior. Para ello, voy a utilizar la aplicación *DEUS 0.6.0* para crear los nodos, eventos y procesos. Comenzaré explicando las características de cada tipo de nodo simulado, y a continuación, realizaré una simulación con los nodos definidos. La creación de los elementos se hace a partir de un documento XML que se le indica al programa cuando se arranca. Además de definirlos en el documento XML, hay que crear clases con unas características determinadas.

Primero voy a explicar en detalle cada uno de los nodos, mostrando fragmentos del XML y del código Java que los crea, a continuación, explicaré los eventos asociados a cada nodo, y por último explicaré los procesos que relacionan los nodos y los eventos.

5.1. Nodos.

En este apartado voy a explicar los diferentes nodos que he utilizado, así como sus parámetros y comportamiento. Antes de empezar a explicar cada nodo en detalle, se muestra una tabla donde se lista cada nodo de los que se van a simular y su categorización dentro del IoT.

Nodo	**Tipo de nodo**
Sensor de temperatura	Nodo final
Electroválvula	Nodo de aplicación
Caldera	Nodo de aplicación
Interruptor	Nodo de aplicación
Sensor de movimiento	Nodo final
Luz	Nodo de aplicación
Lector biométrico	Nodo de proceso
Router	Nodo de comunicación
Cámara	Nodo final
Reloj	Nodo final
Servidor	Nodo de proceso

Tabla 1: Resumen de nodos simulados

5.1.1. Sensor de temperatura.

Este nodo representa un sensor de temperatura instalado cerca de las ventanas, hay uno en cada habitación y envía periódicamente información al servidor sobre la temperatura de la habitación a la que pertenece.

En el elemento XML se han definido 4 propiedades:

- **valorMinimo**: Esta propiedad indica la temperatura a la que si se llega, el servidor debe ordenar a los elementos necesarios que se activen para subir la temperatura de la habitación. Se establece a 20º.
- **valorMaximo**: Esta propiedad indica la temperatura a la que si se llega, el servidor debe ordenar a los elementos necesarios que se desactiven para que deje de subir la temperatura de la habitación. Se establece a 26º.
- **valorActual**: La temperatura de la habitación en este momento, por defecto se inicia a 25º.

- **localizacion**: Indica la habitación en la que se encuentra el sensor de temperatura. En esta simulación, al haber 6 habitaciones definidas, los valores que puede tomar son: H1, H2, H3, H4, H5 y H6.

En el elemento que se define en el documento XML se indican cada una de las propiedades y la clase Java que simulará este nodo. En la figura 43 se puede ver un ejemplo de cómo se define el sensor de temperatura para la habitación H1.

```
<aut:node handler="it.unipr.ce.dsg.deus.nodos.temperatura.SensorTemperatura"
    id="sensorTemperaturaH1">
    <aut:params>
        <aut:param name="valorMinimo" value="20" />
        <aut:param name="valorMaximo" value="26" />
        <aut:param name="valorActual" value="25"/>
        <aut:param name="localizacion" value="H1"/>
    </aut:params>
</aut:node>
```

Figura 43: Sensor de temperatura en XML

Para definir el comportamiento del sensor de temperatura, se utilizará una clase Java que herede directa o indirectamente de la clase *Node* que se define en la aplicación. Para esta simulación he decidido que los nodos hereden de la clase *Peer* (que hereda de la clase *Node*) para disponer de las características que se ofrecen para comunicar dos elementos. En este caso, se utilizará para enviar información al servidor a través del router.

En la clase que define el nodo he creado las mismas propiedades que el elemento XML, de manera que al instanciar la clase, cargue los valores indicados y se pueda trabajar con ellos. En la figura 44 se ve la declaración de la clase y su constructor.

```
public class SensorTemperatura extends Peer {

    private int valorMinimo;
    private int valorMaximo;
    private int valorActual;
    private String location;

    public SensorTemperatura(String id, Properties params,
            ArrayList<Resource> resources) throws InvalidParamsException {
        super(id, params, resources);
        valorMinimo = new Integer((String)params.get("valorMinimo"));
        valorMaximo = new Integer((String)params.get("valorMaximo"));
        valorActual = new Integer((String)params.get("valorActual"));
        location = (String)params.get("localizacion");
    }
```

Figura 44: Sensor de temperatura en Java

5.1.2. Electroválvula

Este nodo se utiliza para habilitar o no que la caldera envíe agua caliente a una habitación Se instará cerca del radiador y permitirá o bloqueará el paso del agua caliente desde la caldera. Tiene dos atributos:

- **Estado**: Indica si está abierta o cerrada. Cuando está abierta, el agua caliente llega al radiador y aumenta la temperatura de la habitación. Los valores que se indican

en el documento XML son “0” para cuando está cerrada y “1” para cuando está abierta.

- **Localización**: Indica la habitación en la que se encuentra la electroválvula. Al igual que en el sensor de temperatura, los valores de este atributo pueden ser: H1, H2, H3, H4, H5 y H6.

En la figura 45 se muestra como se define la electroválvula en el documento XML. Por defecto, todas las electroválvulas están cerradas y no se abrirán hasta que el servidor envíe la orden.

```
<aut:node handler="it.unipr.ce.dsg.deus.nodos.electrovalvula.Electrovalvula"
    id="electrovalvulaH1">
    <aut:params>
        <aut:param name="estado" value="0"/>
        <aut:param name="localizacion" value="H1"/>
    </aut:params>
</aut:node>
```

Figura 45: Electroválvula en XML

Para definir el nodo en la simulación se va crear la clase indicada en el código XML, haciendo que herede de la clase *Peer*, que a su vez hereda de la clase *Node*, lo que facilita que se conecte con el router para recibir las órdenes del servidor. Para poder controlar su comportamiento en la clase se han creado los atributos *estado* y *localizacion* para que vinculan con las propiedades definidas en el documento XML. En la Figura 46 se ve la definición de la clase y su constructor.

```
public class Electrovalvula extends Peer {

    private String estado;
    private String localizacion;

    public Electrovalvula(String id, Properties params,
            ArrayList<Resource> resources) throws InvalidParamsException {
        super(id, params, resources);
        estado = (String)params.get("estado");
        localizacion = (String)params.get("localizacion");
    }
```

Figura 46: Electroválvula en Java

5.1.3. Caldera.

Este elemento es el responsable de calentar el agua y distribuirla por todas las habitaciones del piso. Se activará o desactivará en función de la orden que reciba del servidor. Debido a esta forma de funcionar, solo es necesario un atributo para la caldera:

- **Estado:** Indica si la caldera está encendida o apagada. Por defecto estará apagada y cambiará el estado cuando el servidor mande la orden correspondiente. Si atributo vale “0” significa que está apagada y si vale “1” significa que está encendida.

En la figura 47 se muestra como se define la caldera en el documento XML.

```
<aut:node handler="it.unipr.ce.dsg.deus.nodos.caldera.Caldera"
    id="caldera">
    <aut:params>
        <aut:param name="encendida" value="0" />
    </aut:params>
</aut:node>
```

Figura 47: Caldera en XML

Para definir el nodo en la simulación se va a definir la clase indicada en el nodo XML, esta clase hereda de *Peer* para poder aprovechar las características de conexión que ofrece. Además, la clase tendrá un solo atributo, al igual que su equivalente en XML que identifica si la caldera está encendida o apagada. Inicialmente la caldera está apagada, y no es hasta que recibe la orden de encendido del servidor hasta que se conecta. De igual forma, cuando desde el servidor se envía la orden de apagarla, la caldera se apaga. En la Figura 48 se ve la definición de la clase y su constructor.

```
public class Caldera extends Peer{

    private String encendida;

    public Caldera(String id, Properties params,
            ArrayList<Resource> resources) throws InvalidParamsException {
        super(id, params, resources);
        encendida = (String)params.get("encendida");
    }
```

Figura 48: Caldera en Java

5.1.4. Interruptor.

Este nodo envía al servidor la orden para que realice un cambio de estado en el estado de otro nodo del sistema identificado como *Luz*. El interruptor tiene dos atributos:

- **Estado:** Indica el último envío de una orden al servidor. Si vale “0” indica que se mandó la orden de apagado y si vale “1”, significa que se mandó la orden para que se encendiera la luz de la habitación.
- **Localizacion:** Indica la habitación en la que se encuentra el interruptor, y por lo tanto, la luz que tendrá que encenderse o apagarse. Al igual que en el sensor de temperatura o la electroválvula, los valores de este atributo pueden ser: H1, H2, H3, H4, H5 y H6.

En la figura 49 se muestra como se define el interruptor en el documento XML. Por defecto, todos los interruptores se inician con el estado “0” y no se actualizará a estado “1” hasta que en la simulación se realice el cambio antes de enviar la orden.

```
<aut:node handler="it.unipr.ce.dsg.deus.nodos.interruptor.Interruptor"
    id="interruptorH1">
    <aut:params>
        <aut:param name="estado" value="0"/>
        <aut:param name="localizacion" value="H1"/>
    </aut:params>
</aut:node>
```

Figura 49: Interruptor en XML

Para definir el nodo en la simulación se va a definir la clase indicada en el nodo XML, esta clase hereda de *Peer* para poder aprovechar las características de conexión que ofrece. Esta clase tiene los mismos atributos que se indican en el documento XML y se utilizan con el mismo criterio. En la Figura 50 se muestra la definición de la clase y su constructor.

```
public class Interruptor extends Peer {

    private String estado;
    private String localizacion;

    public Interruptor(String id, Properties params,
            ArrayList<Resource> resources) throws InvalidParamsException {
        super(id, params, resources);
        estado = (String)params.get("estado");
        localizacion = (String)params.get("localizacion");
    }
```

Figura 50: Interruptor en Java

5.1.5. Sensor de movimiento

Este nodo es el responsable de enviar al servidor información sobre si hay alguien en la habitación y la luminosidad de la misma, para que en caso de que sea necesario, el servidor envíe la orden de encender la luz. El sensor de movimiento tiene tres atributos:

- **Luminosidad:** Indica en un valor de 1 a 100 el nivel de luminosidad de la habitación. Cuando el valor es inferior a 50, el servidor estudia si tiene que enviar la orden de encender la luz o no.
- **MovimientoDetectado:** Esta propiedad toma valores de “0” o “1” en caso de que detecte movimiento. Cuando se detecta movimiento se envía la información que detecte el sensor en ese momento y en caso de que se envíen datos pero no haya habido detección de movimiento se enviará un “0”.
- **Localizacion:** Indica la habitación en la que se encuentra el sensor de movimiento, y por lo tanto, la luz que tendrá que encenderse o apagarse. Al igual que en el interruptor, los valores de este atributo pueden ser: H1, H2, H3, H4, H5 y H6.

En la figura 51 se muestra el código XML necesario para crear un sensor de movimiento. Por defecto, la luminosidad está a 0 y no se ha detectado movimiento en la habitación indicada.

```
<aut:node handler="it.unipr.ce.dsg.deus.nodos.movimiento.SensorMovimiento"
    id="sensorMovimientoH1">
    <aut:params>
        <aut:param name="luminosidad" value="0"/>
        <aut:param name="movimientoDetectado" value="0"/>
        <aut:param name="localizacion" value="H1"/>
    </aut:params>
</aut:node>
```

Figura 51: Sensor de movimiento en XML

Para definir el nodo en la simulación se va a definir la clase indicada en el nodo XML, esta clase hereda de *Peer* para poder aprovechar las características de conexión que ofrece. Al ser la clase *Peer* una clase que hereda de *Node*, el sistema se comporta como

le corresponde dentro de la simulación. Esta clase tiene los mismos atributos que se indican en el documento XML y se utilizan con el mismo criterio. En la Figura 52 se muestra la definición de la clase y su constructor.

```
public class SensorMovimiento extends Peer {

    private String luminosidad;
    private String movimientoDetectado;
    private String localizacion;

    public SensorMovimiento(String id, Properties params,
            ArrayList<Resource> resources) throws InvalidParamsException {
        super(id, params, resources);
        localizacion = (String)params.get("localizacion");
        luminosidad = (String)params.get("luminosidad");
        movimientoDetectado = (String)params.get("movimientoDetectado");
    }
```

Figura 52: Sensor de movimiento en Java

5.1.6. Luz

Este nodo es el responsable de permitir que se enciendan o se apaguen las luces en cada habitación, por lo que es necesario que uno de sus atributos sea la localización del mismo. El encendido o no de la luz está condicionado por el interruptor y el sensor de movimiento, teniendo más preferencia el interruptor que el sensor de movimiento. Los atributos que forman este nodo son:

- **Estado:** Indica si la luz está encendida o apagada. Cuando la luz esté apagada esta propiedad valdrá “0” y si está encendida valdrá “1”. Por defecto, se iniciará en estado apagada.
- **Localizacion:** Indica la habitación en la que se encuentra la luz. Al igual que en el interruptor, los valores de este atributo pueden ser: H1, H2, H3, H4, H5 y H6.

En la Figura 53 se muestra el código XML necesario para que se cree este nodo en la simulación.

```
<aut:node handler="it.unipr.ce.dsg.deus.nodos.luces.Luces" id="LuzH1">
    <aut:params>
        <aut:param name="estado" value="0"/>
        <aut:param name="localizacion" value="H1"/>
    </aut:params>
</aut:node>
```

Figura 53: Luz en XML

Al igual que los otros nodos de la simulación, este nodo también va a heredar de la clase *Peer* en su implementación en Java, ya que para recibir las órdenes del servidor necesita aprovechar las características de conexión que ofrece la clase *Peer*. Esta clase tiene los mismos atributos que se indican en el documento XML y se utilizan con el mismo criterio. En la Figura 54 se muestra la definición de la clase y su constructor.

```java
public class Luces extends Peer {

    private String estado;
    private String localizacion;

    public Luces(String id, Properties params,
            ArrayList<Resource> resources) throws InvalidParamsException {
        super(id, params, resources);
        estado = (String)params.get("estado");
        localizacion = (String)params.get("localizacion");
    }
```

Figura 54: Luz en Java

5.1.7. Lector biométrico

Este nodo se instalará en la puerta de entrada de la vivienda y servirá para que la persona que acaba de entrar se identifique, si pasados 30 segundos desde que la cámara de la entrada detecta movimiento, no se ha utilizado este dispositivo, el servidor enviará una alerta a un sistema externo informando de que se ha producido una entrada no autorizada en la vivienda. Este sistema puede utilizarse para controlar el acceso a otras habitaciones por lo que es necesario que el dispositivo disponga de un atributo que identifique en que habitación se encuentra. Los atributos de este nodo son:

- **Localizacion:** Indica en que habitación se encuentra el sensor biométrico. En esta simulación, solamente toma el valor H1 ya que solo hay 1.
- **idLeido:** Este atributo contiene información sobre la persona que se ha identificado con este dispositivo. Debido a la diversidad de dispositivos que se pueden utilizar y a que la información que envía cambia en función de cada dispositivo, en esta simulación se mandará un número que identifica a la persona que entra.

En la Figura 55 se muestra el código XML necesario para que se cree este nodo en la simulación.

```xml
<aut:node handler="it.unipr.ce.dsg.deus.nodos.lector.SensorBiometrico"
    id="lectorH1">
    <aut:params>
        <aut:param name="idLeido" value="0"/>
        <aut:param name="localizacion" value="H1"/>
    </aut:params>
</aut:node>
```

Figura 55: Sensor biométrico en XML

Como en los otros nodos de la simulación, este nodo también va a heredar de la clase *Peer* en su implementación en Java, ya que para enviar la información al servidor necesita aprovechar las características de conexión que ofrece la clase *Peer*. Esta clase tiene los mismos atributos que se indican en el documento XML y se utilizan con el mismo criterio. En la Figura 56 se muestra la definición de la clase y su constructor.

```java
public class SensorBiometrico extends Peer {

    private String idLeido;

    private String localizacion;

    public SensorBiometrico(String id, Properties params,
            ArrayList<Resource> resources) throws InvalidParamsException {
        super(id, params, resources);
        idLeido = (String)params.get("idLeido");
        localizacion = (String)params.get("localizacion");
    }
```

Figura 56: Sensor biométrico en Java

5.1.8. Router

Este elemento es crucial en la simulación, gracias a este nodo se pueden comunicar los distintos elementos con el servidor y permite al servidor enviar órdenes a los nodos para indicarles que se enciendan, apaguen, comiencen a grabar, etc… Debido a que la única responsabilidad de este nodo es la de transmitir información entre nodos, no es necesario crearle ningún atributo en el documento XML, tal y como se ve en la figura 57.

```xml
<aut:node handler="it.unipr.ce.dsg.deus.nodos.router.Router"
    id="router"/>
```

Figura 57: Router en XML

Este nodo también va a heredar de la clase *Peer* en su implementación en Java, ya que para enviar la información al servidor necesita aprovechar las características de conexión que ofrece la clase *Peer*. Esta clase tiene los mismos atributos que se indican en el documento XML y se utilizan con el mismo criterio. En la figura 58 se muestra la definición de la clase y su constructor.

```java
public class Router extends Peer {

    public Router(String id, Properties params,
            ArrayList<Resource> resources) throws InvalidParamsException {
        super(id, params, resources);
    }
```

Figura 58: Router en Java

5.1.9. Cámara

Este dispositivo está situado enfrente de la puerta de la vivienda y se utiliza en conjunto con el sensor biométrico para detectar cuando entra una persona autorizada o cuando entra un intruso. Los atributos de la cámara son:

- **InformacionCaptada:** Este atributo representa las imágenes captadas por la cámara que serán recibidas por el servidor. En esta simulación se van a tratar como una cadena de caracteres, no obstante, si la simulación se implantara en el mundo real, habría que cambiar el tipo de información del programa que se instalaría en el servidor.
- **Estado:** Este atributo representa si la cámara está activa (grabando) o no. De forma que si el valor de este atributo es “0” significa que la cámara aunque esté activa, no está grabando y que si este atributo vale “1” se está grabando lo que capta la cámara.

- **movimientoDetectado:** Este atributo valdrá "1" cuando la cámara detecte movimiento en la puerta de entrada y mande la información al servidor para que realice todo el protocolo necesario para detectar si quien ha entrado es un intruso o no.

En la Figura 59 se muestra el código XML necesario para que se cree este nodo en la simulación.

```
<aut:node handler="it.unipr.ce.dsg.deus.nodos.camara.Camara"
    id="camara">
    <aut:params>
        <aut:param name="informacionCaptada" value="0" />
        <aut:param name="estado" value="0" />
        <aut:param name="movimientoDetectado" value="40" />
    </aut:params>
</aut:node>
```

Figura 59: Cámara en XML

Este nodo también va a heredar de la clase *Peer* en su implementación en Java, ya que para enviar la información al servidor necesita aprovechar las características de conexión que ofrece la clase *Peer*. Esta clase tiene los mismos atributos que se indican en el documento XML y se utilizan con el mismo criterio. En la figura 60 se muestra la definición de la clase y su constructor.

```
public class Camara extends Peer {

    private String informacionCaptada;
    private String estado;
    private String movimientoDetectado;

    public Camara(String id, Properties params,
            ArrayList<Resource> resources) throws InvalidParamsException {
        super(id, params, resources);
        informacionCaptada = (String)params.get("informacionCaptada");
        estado = (String)params.get("estado");
        movimientoDetectado = (String)params.get("movimientoDetectado");

    }
```

Figura 60: Cámara en Java

5.1.10. Reloj

Este nodo es un dispositivo que va con el usuario y que necesita de otro dispositivo auxiliar para cumplir su función, generalmente, se utilizará un teléfono inteligente como apoyo. Para esta simulación, se ha omitido el paso intermedio del teléfono y se ha configurado para que se conecte directamente al router. Los atributos que se van a controlar en el reloj son:

- **Tension:** Este atributo representa el valor de la tensión arterial de la persona que lleva puesto el reloj. Si la tensión cae por debajo de 80, se mandará un mensaje a un servicio externo de emergencias.
- **Pulsaciones:** Este atributo representa el número de pulsaciones por minuto de la persona que lleva puesto el reloj. Si las pulsaciones caen por debajo de 40, se mandará un mensaje a un servicio externo de emergencias.

- **idPersona:** Identifica de manera concreta y sin error a la persona que lleva el reloj puesto.

En la figura 61 se muestra el código XML necesario para que se cree este nodo en la simulación.

```
<aut:node handler="it.unipr.ce.dsg.deus.nodos.reloj.Reloj"
    id="reloj">
    <aut:params>
        <aut:param name="idPersona" value="0" />
        <aut:param name="tension" value="0" />
        <aut:param name="pulsaciones" value="40" />
    </aut:params>
</aut:node>
```

Figura 61: Reloj en XML

Este nodo también va a heredar de la clase *Peer* en su implementación en Java, ya que para enviar la información al servidor necesita aprovechar las características de conexión que ofrece la clase *Peer*. Esta clase tiene los mismos atributos que se indican en el documento XML y se utilizan con el mismo criterio. En la Figura 62 se muestra la definición de la clase y su constructor.

```
public class Reloj extends Peer {

    private String idPersona;
    private String tension;
    private String pulsaciones;

    public Reloj(String id, Properties params,
            ArrayList<Resource> resources) throws InvalidParamsException {
        super(id, params, resources);
        idPersona = (String)params.get("idPersona");
        tension = (String)params.get("tension");
        pulsaciones = (String)params.get("pulsaciones");
    }
```

Figura 62: Reloj en Java

5.1.11. Servidor

El servidor es la central de toda la simulación, es el responsable de recibir información de los nodos finales y, procesando esa información, transmitir a los nodos de aplicación las órdenes con las que ha sido configurado. Para esta simulación no se ha creado ninguna base de datos, en un sistema real, además de los elementos que he comentado en este apartado, habría que tener una base de datos en la que el servidor almacenara toda la información de lo que está ocurriendo en la vivienda, de la información que recibe y de las órdenes que envía.

En la Figura 63 se muestra el código XML necesario para que se cree este nodo en la simulación.

```
<aut:node handler="it.unipr.ce.dsg.deus.nodos.servidor.Servidor"
    id="servidor"/>
```

Figura 63: Servidor en XML

Este nodo también va a heredar de la clase *Peer* en su implementación en Java, ya que para enviar la información al servidor necesita aprovechar las características de conexión que ofrece la clase *Peer*. Esta clase tiene los mismos atributos que se indican en el documento XML y se utilizan con el mismo criterio. En la Figura 64 se muestra la definición de la clase y su constructor.

```
public class Servidor extends Peer{

    public Servidor(String id, Properties params,
            ArrayList<Resource> resources) throws InvalidParamsException {
        super(id, params, resources);
    }
```

Figura 64: Servidor en Java

5.2. Eventos

Los eventos se los responsables de que la simulación se lleve a cabo. Al definir un evento tenemos que definir qué clase java va a ser la responsable de su comportamiento y que tipo de evento va a ser. Debido a que en la simulación estoy usando *DEUS* es necesario crear eventos que instancien los nodos. Además de los eventos de creación de nodo, se crearán eventos para definir el comportamiento de los nodos en la simulación de la vivienda. En este apartado, explicaré un evento de creación de nodos y los eventos de enviar información y recibir órdenes que se gestionan con el servidor. Antes de entrar en detalle a explicar cada uno de los eventos, se mostrará una tabla resumen que lista cada uno de los eventos y los nodos con los que está involucrado.

Evento	Nodos involucrados
Creación de nodos	Sensor de temperatura, electroválvula, caldera, interruptor, sensor de movimiento, luz, lector biométrico, router, cámara, reloj y servidor.
Envío información del reloj	Router, reloj y servidor.
Envío información del sensor biométrico	Lector biométrico, router, cámara y servidor.
Envío información de cámara	Lector biométrico, router, cámara y servidor.
Envío información del sensor de movimiento.	Interruptor, sensor de movimiento, luz, router y servidor.
Enviar información del interruptor	Interruptor, sensor de movimiento, luz, router y servidor.
Enviar información del sensor de temperatura	Sensor de temperatura, electroválvula, caldera, router y servidor
Transmisión de información del router.	Sensor de temperatura, electroválvula, caldera, interruptor, sensor de movimiento, luz, lector biométrico, router, cámara, reloj y servidor.

Tabla 2: Relación eventos y nodos

5.2.1. Eventos de creación de nodos

Cada nodo tiene que estar asociado al menos con un evento manejado por *BirthEvent*. Este evento representa el nacimiento de un nodo de simulación. Durante su ejecución, una instancia del nodo asociado al evento es creado. Para todos los nodos de la simulación

se ha dejado la instancia por defecto ya que no es necesario crear ninguna configuración adicional, no obstante, aunque todos los eventos de nacimiento estén vacíos, se ha creado un evento por cada tipo de nodo, al que se llamará con cada uno de los nodos listados anteriormente.

En la figura 65 se muestra un ejemplo de cómo se ha definido en el documento XML el evento del nacimiento de los nodos de temperatura, la creación de los eventos de nacimiento de los otros nodos, es equivalente.

```
<aut:event id="birthTemperatura" handler="it.unipr.ce.dsg.deus.impl.event.BirthEvent"
    schedulerListener="it.unipr.ce.dsg.deus.nodos.temperatura.SensorTemperaturaBirthSchedulerListener"/>
```

Figura 65: Nacimiento de nodos en XML

En la figura 66 muestro una imagen de la clase Java indica en el elemento XML, como ya he comentado, esa clase está vacía ya que no es necesario cambiar nada de la configuración por defecto.

```
public class SensorTemperaturaBirthSchedulerListener
            implements SchedulerListener {

    @Override
    public void newEventScheduled(Event parentEvent, Event newEvent) {
    }

}
```

Figura 66: Evento de nacimiento en Java

5.2.2. Envío de información del reloj

Este evento se encarga de leer la información del reloj que lleva el usuario en su muñeca y, a través del router, mandarla al servidor. Los datos que se envían son las pulsaciones por minuto, la tensión arterial y el identificador de la persona que lleva el reloj. En la figura 67 se muestra un ejemplo de cómo se define el evento en el documento XML.

```
<aut:event id="enviarInformacionServerReloj" handler="it.unipr.ce.dsg.deus.nodos.reloj.RelojEvent"
    schedulerListener="it.unipr.ce.dsg.deus.nodos.reloj.EnviarInformacionRelojServer"/>
```

Figura 67: Evento de envío de información del reloj al servidor

En el código de la simulación, la clase encargada de enviar la información tiene que componer el mensaje que será enviado al servidor, para ello, primero busca los nodos que son instancia del servidor y del router, a continuación busca los nodos relacionados con el evento para componer el mensaje utilizando las propiedades del nodo *Reloj*. El mensaje se tendrá como nodo origen el reloj seleccionado, como nodo destino, el servidor, y utilizará la instancia del router para transmitir el mensaje. En la figura 68 se muestra una imagen del código que se ejecuta el evento de envío de información al servidor.

```
@Override
public void run() throws RunException {
    Servidor server = (Servidor) Engine.getDefault().getConfigNodes().get(buscarNodoServidor());
    Router router = (Router) Engine.getDefault().getConfigNodes().get(buscarNodoRouter());
    for(int i=0; i<getParentProcess().getReferencedNodes().size(); i++) {
        Reloj reloj = (Reloj) getParentProcess().getReferencedNodes().get(i);
        reloj.setPulsaciones(obtenerPulsaciones());
        reloj.setTension(obtenerTension());
        getLogger().info("Se va a enviar un mensaje del reloj con las pulsaciones: "
                + ""+reloj.getPulsaciones()+" y la tension "+reloj.getTension());
        reloj.sendMessage(server, router);
        associatedNode = reloj;
    }
}
```

Figura 68: Envío de información del reloj al servidor en Java

Cuando el servidor recibe esta información la analiza, y si las pulsaciones o la tensión arterial están fuera del rango establecido, se envía una señal de auxilio a un sistema externo que contacta con las autoridades sanitarias.

5.2.3. Envío de información del sensor biométrico

Este evento se encarga de leer la información de la persona que acaba de entrar en la vivienda. Normalmente, este evento no se espera hasta que la cámara haya captado movimiento, en ese momento, se lanzará una cuenta atrás en la que si no se recibe información de este sensor o si la que se recibe es errónea, el servidor se encargará de enviar la información a un sistema externo para informar de una entrada no autorizada. En la figura 69 se muestra el código del documento XML que crea este evento.

```
<aut:event id="enviarInformacionServerBiometrico" handler="it.unipr.ce.dsg.deus.nodos.lector.LectorEvent"
    schedulerListener="it.unipr.ce.dsg.deus.nodos.lector.EnviarInformacionBiometricoServer"/>
```

Figura 69: Evento de envío de información del sensor biométrico al servidor

En el código de la simulación, la clase *LectorEvent* tiene que encargarse de generar un mensaje que se enviará al servidor a través del router, para ello, lo primero que hace es buscar las instancias del servidor, del router y del lector biométrico para componer el mensaje que se va a enviar. En la figura 70 se muestra una imagen del código que se ejecuta el evento de envío de información al servidor.

```
@Override
public void run() throws RunException {
    Servidor server = (Servidor) Engine.getDefault().getConfigNodes().get(buscarNodoServidor());
    Router router = (Router) Engine.getDefault().getConfigNodes().get(buscarNodoRouter());
    SensorBiometrico lector =
            (SensorBiometrico) Engine.getDefault().getConfigNodes().get(buscarNodoSensorBiometrico());
    lector.setIdLeido(obtenerDatosLeidos());
    lector.sendMessage(server, router);
    associatedNode = lector;
}
```

Figura 70: Evento de envío de información del sensor biométrico al servidor

Cuando el servidor recibe esta información, comprueba si el identificador que ha enviado el sector está en la lista de autorizados envía una alarma.

5.2.4. Envío de información de la cámara

La cámara envía continuamente al servidor la información que está captando, no obstante, cuando detecta movimiento, se le envía al servidor la información de que ha captado movimiento y fotos tomadas de la entrada de la vivienda que el servidor se encarga de almacenar. La cámara posee una propiedad que indica si se está grabando lo

que está captando o no, cuando se detecta movimiento, automáticamente comienza a grabar hasta que el servidor envía la orden de que deje de grabar. En la figura 71 se muestra el código del documento XML que crea este evento.

```
<aut:event id="enviarInformacionCamaraServer" handler="it.unipr.ce.dsg.deus.nodos.camara.CamaraEvent"
    schedulerListener="it.unipr.ce.dsg.deus.nodos.camara.EnviarInformacionCamaraServer"/>
```

Figura 71: Evento de envío de información de la cámara al servidor

Para crear el mensaje que se le enviará al servidor es necesario encontrar las instancias del servidor, del router y de la cámara. Una vez localizados los tres nodos, se comprueba si la cámara ha captado información y tanto si se ha captado como si no, se envía la información al servidor para que la analice. En la figura 72 se muestra una imagen del código que se ejecuta el evento de envío de información al servidor.

```
@Override
public void run() throws RunException {
    Servidor server = (Servidor) Engine.getDefault().getConfigNodes().get(buscarNodoServidor());
    Router router = (Router) Engine.getDefault().getConfigNodes().get(buscarNodoRouter());
    Camara camara = (Camara) Engine.getDefault().getConfigNodes().get(buscarNodoCamara());
    camara.setConnected(false);
    camara.setEstado("0");
    camara.setInformacionCaptada(obtenerMovimientoDetectado());
    camara.setMovimientoDetectado(camara.getInformacionCaptada());
    associatedNode = camara;
    if(camara.getId().equals("0")) {
        getLogger().info("La camara no ha captado movimiento.");
    }
    else {
        camara.setConnected(true);
        getLogger().info("La camara ha captado movimiento.");
        getLogger().info("Enviando información captada");
    }
    camara.sendMessage(server, router);
}
```

Figura 72: Envío de información de la cámara al servidor.

Si al analizar la información que ha recibido el servidor no hay movimiento, se ignora el mensaje. En caso contrario, se lanza una cuenta atrás para el envío del mensaje de alarma que no será cancelado a no ser que se reciba información del sensor biométrico. Si se recibe a tiempo el mensaje del lector biométrico con una identificación aceptada, se cancela la cuenta atrás y se envía una orden a la cámara para que deje de grabar, además, se eliminará el archivo generado por la grabación de la cámara.

5.2.5. Transmisión de información del router

El evento de la transmisión del mensaje del router no está implementado en la clase *RouterEvent*, aunque es necesaria para la gestión de los mensajes. Este evento se encarga de cuando recibe un mensaje enviarlo al nodo que le corresponde en función del nodo al que esté asociado y que actúa como nodo destino. En la figura 73 se muestra el código del documento XML que crea este evento.

```
<aut:event id="transmitirMensajeRouter" handler="it.unipr.ce.dsg.deus.nodos.router.RouterEvent"
    schedulerListener="it.unipr.ce.dsg.deus.nodos.router.TransmitirMensaje"/>
```

Figura 73: Evento de transmisión de mensaje en el router

El funcionamiento de este nodo es muy sencillo, simplemente se comprueba la instancia del nodo asociado, y en función de la misma se utiliza una función u otra. En la

figura 74 se muestra una imagen del código que se ejecuta el evento de envío de información a los diferentes nodos destino.

```
@Override
public void run() throws RunException {
    if(getAssociatedNode() instanceof Router) {
        Router router = (Router) getAssociatedNode();
        router.transmitirMensaje(msg);
    }
    else {
        if(getAssociatedNode() instanceof Servidor) {
            Servidor servidor = (Servidor) getAssociatedNode();
            servidor.recibirMensaje(msg);
        }
        else if(getAssociatedNode() instanceof Electrovalvula) {
            Electrovalvula electrovalvula = (Electrovalvula) getAssociatedNode();
            electrovalvula.recibirOrden(msg);
        }
        else if(getAssociatedNode() instanceof Caldera) {
            Caldera caldera = (Caldera) getAssociatedNode();
            caldera.recibirOrden(msg);
        }
        else if(getAssociatedNode() instanceof Luces) {
            Luces luz = (Luces) getAssociatedNode();
            luz.recibirOrden(msg);
        }
    }
}
```

Figura 74: Seleccionar destino del mensaje.

5.2.6. Envío de información del sensor de movimiento.

Este evento se encarga de enviar la información de si hay movimiento o no en una habitación, y luminosidad de la misma. Cuando este nodo detecta movimiento, genera un mensaje que le envía al servidor para que decida si tiene que encender o no la luz de la habitación. En la figura 75 se muestra el código del documento XML que crea este evento.

```
<aut:event id="enviarOrdenMovimientoServer" handler="it.unipr.ce.dsg.deus.nodos.movimiento.MovimientoEvent"
    schedulerListener="it.unipr.ce.dsg.deus.nodos.movimiento.EnviarMovimientoYServidor"/>
```

Figura 75: Evento de envío de información del sensor de movimiento al servidor

Al detectar el movimiento, el sensor crea un mensaje con la luminosidad de la habitación, el aviso de que ha detectado movimiento y la habitación en la que se encuentra. Para enviar el mensaje, primero hay que localizar la instancia del servidor y del router, ya que la información se va a enviar al servidor a través del router. Una vez que están todos los nodos localizados, se realiza el envío. En la figura 76 se muestra una imagen del código que se ejecuta el evento de envío de información al servidor.

```
@Override
public void run() throws RunException {
    Servidor server = (Servidor) Engine.getDefault().getConfigNodes().get(buscarNodoServidor());
    Router router = (Router) Engine.getDefault().getConfigNodes().get(buscarNodoRouter());
    for(int i=0; i<getParentProcess().getReferencedNodes().size(); i++) {
        SensorMovimiento sMov = (SensorMovimiento) getParentProcess().getReferencedNodes().get(i);
        sMov.setLuminosidad(obtenerLuminosidad(sMov.getLocalizacion()));
        sMov.setMovimientoDetectado(obtenerMovimientoDetectado(sMov.getLocalizacion()));
        sMov.sendMessage(server, router);
        associatedNode = sMov;
        getLogger().info("Enviando mensaje desde el "
                + "sensor de movimiento de la habitacion " +sMov.getLocalizacion());
    }
}
```

Figura 76: Envío de información del sensor de movimiento al servidor

5.2.7. Enviar de información del interruptor.

Este evento es el responsable de que cuando se accione un interruptor se envíe información al servidor para que actualice el estado de la luz de la habitación en la que se encuentra. En la figura 77 se muestra el código del documento XML que crea este evento.

```
<aut:event id="enviarOrdenInterruptorServer" handler="it.unipr.ce.dsg.deus.nodos.interruptor.InterruptorEvent"
    schedulerListener="it.unipr.ce.dsg.deus.nodos.interruptor.EnviarInterruptorYSensor"/>
```

Figura 77: Evento de envío de información del interruptor al servidor

Para ello, se genera un mensaje que, a través del router, le información de que se ha producido un cambio de estado. El evento se encarga de producir el cambio de estado y a partir de ese cambio de estado se realiza el envío del mensaje que encenderá o apagará la luz de la habitación. En la figura 78 se muestra una imagen del código que se ejecuta el evento de envío de información al servidor.

```
@Override
public void run() throws RunException {
    Servidor server = (Servidor) Engine.getDefault().getConfigNodes().get(buscarNodoServidor());
    Router router = (Router) Engine.getDefault().getConfigNodes().get(buscarNodoRouter());
    for(int i=0; i<getParentProcess().getReferencedNodes().size(); i++) {
        Interruptor interruptor = (Interruptor) getParentProcess().getReferencedNodes().get(i);
        if(interruptor.getEstado().equals("0")) {
            interruptor.setEstado("1");
        }
        else {
            interruptor.setEstado("0");
        }
        interruptor.sendMessage(server, router);
        associatedNode = interruptor;
        getLogger().info("Se ha realizado un cambio de estado en el servidor,"
                + " el nuevo estado es: "+interruptor.getEstado());
    }
}
```

Figura 78: Evento de envío de mensaje del interruptor al servidor

Cuando el servidor recibe el mensaje, cambia el estado de la luz, además tiene en cuenta de que si recibe información de movimiento en la habitación pero el interruptor fue el último en mandar el cambio de estado para apagar la luz, no se encienda hasta que el interruptor no vuelva a enviar el cambio de estado para permitir que el sensor de movimiento encienda las luces.

5.2.8. Enviar de información del sensor de temperatura.

Por último, el proceso de envío de información del sensor de temperatura es el responsable de enviar al servidor la temperatura actual de la habitación, el identificador de la habitación en la que se encuentra el sensor, el valor mínimo aceptado en la habitación y el valor máximo de la temperatura. En la figura 79 se muestra el código del documento XML que crea este evento.

```
<aut:event id="enviarTempServer" handler="it.unipr.ce.dsg.deus.nodos.temperatura.SensorTemperaturaEvent"
    schedulerListener="it.unipr.ce.dsg.deus.nodos.temperatura.EnviarSensorTemperaturaYServidor"/>
```

Figura 79: Evento de envío de información del sensor de temperatura al servidor.

Al igual que en los otros eventos, para enviar el mensaje con la información es necesario localizar la instancia del servidor, del router y del sensor de la habitación de la que se vaya a enviar la información. Con esa información, se genera un mensaje que recibirá el router y reenviará al servidor. En la figura 80 se muestra una imagen del código que se ejecuta el evento de envío de información al servidor.

```
@Override
public void run() throws RunException {
    Servidor server = (Servidor) Engine.getDefault().getConfigNodes().get(buscarNodoServidor());
    Router router = (Router) Engine.getDefault().getConfigNodes().get(buscarNodoRouter());
    for(int i=0; i<getParentProcess().getReferencedNodes().size(); i++) {
        SensorTemperatura temperatura = (SensorTemperatura) getParentProcess().getReferencedNodes().get(i);
        temperatura.setValorActual(modificarTemperaturaActual(temperatura));
        getLogger().info("Se va a enviar la temperatura de la habitacion "
                +temperatura.getLocation()+" que es "+temperatura.getValorActual());
        temperatura.sendMessage(server, router);
        associatedNode = temperatura;
    }
}
```

Figura 80: Evento de envío de temperatura de la habitación

Cuando el servidor recibe el mensaje del sensor de temperatura, comprueba que la temperatura es inferior al límite permitido, en ese caso envía a la caldera la orden de que se encienda sin tener en cuenta de si ya está encendida o no, y además, localiza el nodo de la electroválvula de la habitación indicada y la envía la orden de que se active para permitir al agua caliente de la caldera que caldee la habitación. Si por el contrario, la temperatura ha superado el máximo permitido, se le enviará la orden a la electroválvula de que cierre el flujo de agua caliente, y si la electroválvula a la que se ha enviado la orden de que se cierre es la última que estaba abierta, se le enviará a la caldera la orden de que se apague.

5.3. Procesos

Los procesos se encargan de relacionar uno o varios eventos con uno o varios nodos. Cada proceso tiene que definir un manejador para regular su comportamiento de manera que los eventos se produzcan con la periodicidad deseada. Además de los nodos y los eventos relacionados, cada proceso deberá definir cuantas veces se va a ejecutar y de manera opcional, si se añade algún retraso entre la ejecución de las repeticiones de los eventos. Por último, es necesario indicar un identificador para poder localizarlo cuando se programe su ejecución en la simulación. Antes de entrar en detalle a explicar cada uno de los procesos, se mostrará una tabla resumen que lista cada uno de los procesos, los eventos que simulan y los nodos con los que están involucrados.

Procesos	Evento	Nodos involucrados
Crear reloj	birthReloj	Reloj.
Crear eventos reloj	Envío información del reloj, transmisión de información del router.	Router, reloj y servidor.
Crear sensor biométrico	birthBiometrico	Sensor biométrico.
Crear eventos del sensor biométrico	Envío información del sensor biométrico, transmisión de información del router.	Lector biométrico, router, cámara y servidor.
Crear cámara	birthCamara	Cámara.
Crear eventos de la cámara	Envío información de cámara, transmisión de información del router.	Lector biométrico, router, cámara y servidor.
Crear router	birthRouter	Router.

Crear eventos del router	Transmisión de información del router.	Sensor de temperatura, electroválvula, caldera, interruptor, sensor de movimiento, luz, lector biométrico, router, cámara, reloj y servidor.
Crear sensor de movimiento	birthMovimiento	Sensor de movimiento.
Crear luces	birthLuz	Luz.
Crear eventos del sensor de movimiento	Envío información del sensor de movimiento, transmisión de información del router.	Interruptor, sensor de movimiento, luz, router y servidor.
Crear sensor de temperatura	birthTemperatura	Sensor de temperatura.
Crear electroválvula	birthElectrovalvula	Electroválvula.
Crear interruptor	birthInterruptor	Interruptor.
Crear eventos de los interruptores	Enviar información del interruptor, transmisión de información del router.	Interruptor, sensor de movimiento, luz, router y servidor.
Crear caldera	birthCaldera	Caldera
Crear eventos del sensor de temperatura	Enviar información del sensor de temperatura, transmisión de información del router.	Sensor de temperatura, electroválvula, caldera, router y servidor
Crear servidor	Birth2	Servidor.

Tabla 3: Relación entre procesos, eventos y nodos

5.3.1. Crear reloj.

El proceso para crear un reloj se ejecuta una vez e involucra al nodo *reloj* y al evento *birthReloj*. Este proceso solo ejecuta una vez porque el nodo solo se crea al comienzo de la ejecución y no es necesario que se vuelva a crear durante toda la simulación. En la figura 81 se muestra el código XML del proceso que enlaza el evento y el nodo indicado.

```
<aut:process handler="it.unipr.ce.dsg.deus.impl.process.RectangularPulsePeriodicProcess"
    id="crearReloj">
    <aut:params>
        <aut:param name="period" value="1" />
        <aut:param name="startVtThreshold" value="0" />
        <aut:param name="stopVtThreshold" value="1" />
    </aut:params>
    <aut:nodes>
        <aut:reference id="reloj" />
    </aut:nodes>
    <aut:events>
        <aut:reference id="birthReloj" />
    </aut:events>
</aut:process>
```

Figura 81: Proceso de creación del nodo reloj

5.3.2. Crear eventos del reloj.

Este proceso es el responsable de que se envíe la información del reloj al servidor. Se van a realizar 1000 envíos y en cada uno de ellos se enviarán todos los datos que el reloj es capaz de leer. En la figura 82 se muestra el código XML del proceso que enlaza el evento y el nodo indicado.

```
<aut:process handler="it.unipr.ce.dsg.deus.impl.process.RectangularPulsePeriodicProcess"
    id="crearEventosReloj">
    <aut:params>
        <aut:param name="period" value="1" />
        <aut:param name="startVtThreshold" value="0" />
        <aut:param name="stopVtThreshold" value="1000" />
    </aut:params>
    <aut:nodes>
        <aut:reference id="reloj" />
    </aut:nodes>
    <aut:events>
        <aut:reference id="enviarInformacionServerReloj" />
    </aut:events>
</aut:process>
```

Figura 82: Proceso de creación de eventos del reloj

5.3.3. Crear sensor biométrico.

El proceso para crear un sensor biometrico se ejecuta una vez e involucra al nodo *lectorH1* y al evento *birthBiometrico*. Este proceso solo ejecuta una vez porque el nodo solo se crea al comienzo de la ejecución y no es necesario que se vuelva a crear durante toda la simulación. En la figura 83 se muestra el código XML del proceso que enlaza el evento y el nodo indicado.

```
<aut:process handler="it.unipr.ce.dsg.deus.impl.process.RectangularPulsePeriodicProcess"
     id="crearSensorBiometrico">
    <aut:params>
        <aut:param name="period" value="1" />
        <aut:param name="startVtThreshold" value="0" />
        <aut:param name="stopVtThreshold" value="1" />
    </aut:params>
    <aut:nodes>
        <aut:reference id="lectorH1" />
    </aut:nodes>
    <aut:events>
        <aut:reference id="birthBiometrico" />
    </aut:events>
</aut:process>
```

Figura 83: Proceso de creación del sensor biométrico

5.3.4. Crear eventos del sensor biométrico.

Este proceso es el responsable de que se envíe la información del sensor biométrico al servidor. Se van a realizar 1000 envíos y en cada uno de ellos se enviarán todos los datos que el sensor biométrico es capaz de leer. A pesar de que se realicen esa cantidad de envíos el servidor solo los tendrá en cuenta si ocurren después de que la cámara haya captado movimiento. En la figura 84 se muestra el código XML del proceso que enlaza el evento y el nodo indicado.

```
<aut:process handler="it.unipr.ce.dsg.deus.impl.process.RectangularPulsePeriodicProcess"
     id="crearEventosSensorBiometrico">
    <aut:params>
        <aut:param name="period" value="1" />
        <aut:param name="startVtThreshold" value="0" />
        <aut:param name="stopVtThreshold" value="1000" />
    </aut:params>
    <aut:nodes>
        <aut:reference id="lectorH1" />
    </aut:nodes>
    <aut:events>
        <aut:reference id="enviarInformacionServerBiometrico" />
    </aut:events>
</aut:process>
```

Figura 84: Proceso de creación de eventos del sensor biométrico.

5.3.5. Crear cámara.

El proceso para crear una cámara se ejecuta una vez e involucra al nodo *camara* y al evento *birthCamara*. Este proceso solo ejecuta una vez porque el nodo solo se crea al comienzo de la ejecución y no es necesario que se vuelva a crear durante toda la simulación. En la figura 85 se muestra el código XML del proceso que enlaza el evento y el nodo indicado.

```
<aut:process handler="it.unipr.ce.dsg.deus.impl.process.RectangularPulsePeriodicProcess"
    id="crearCamara">
    <aut:params>
        <aut:param name="period" value="1" />
        <aut:param name="startVtThreshold" value="0" />
        <aut:param name="stopVtThreshold" value="1" />
    </aut:params>
    <aut:nodes>
        <aut:reference id="camara" />
    </aut:nodes>
    <aut:events>
        <aut:reference id="birthCamara" />
    </aut:events>
</aut:process>
```

Figura 85: Proceso de creación de cámara

5.3.6. Crear eventos de la cámara.

Este proceso es el responsable de que se envíe la información de la cámara al servidor. Se van a realizar 1000 envíos y en cada uno de ellos se enviarán todos los datos que la cámara es capaz de procesar. En la figura 86 se muestra el código XML del proceso que enlaza el evento y el nodo indicado.

```
<aut:process handler="it.unipr.ce.dsg.deus.impl.process.RectangularPulsePeriodicProcess"
        id="crearEventosCamara">
    <aut:params>
        <aut:param name="period" value="1" />
        <aut:param name="startVtThreshold" value="0" />
        <aut:param name="stopVtThreshold" value="1000" />
    </aut:params>
    <aut:nodes>
        <aut:reference id="camara" />
    </aut:nodes>
    <aut:events>
        <aut:reference id="enviarInformacionCamaraServer" />
        <aut:reference id="recibirInformacionCamaraServer" />
    </aut:events>
</aut:process>
```

Figura 86: Proceso de crear eventos de la cámara

5.3.7. Crear router.

El proceso para crear un router se ejecuta una vez e involucra al nodo *router* y al evento *birthRouter*. Este proceso solo ejecuta una vez porque el nodo solo se crea al comienzo de la ejecución y no es necesario que se vuelva a crear durante toda la simulación. En la figura 87 se muestra el código XML del proceso que enlaza el evento y el nodo indicado.

```
<aut:process handler="it.unipr.ce.dsg.deus.impl.process.RectangularPulsePeriodicProcess"
     id="crearRouter">
    <aut:params>
        <aut:param name="period" value="1" />
        <aut:param name="startVtThreshold" value="0" />
        <aut:param name="stopVtThreshold" value="1" />
    </aut:params>
    <aut:nodes>
        <aut:reference id="router" />
    </aut:nodes>
    <aut:events>
        <aut:reference id="birthRouter" />
    </aut:events>
</aut:process>
```

Figura 87: Proceso de creación del router

5.3.8. Crear eventos del router.

Este proceso es el responsable de que se envíe la información de la cámara al servidor. Se van a realizar 10000 envíos porque es necesario que se ejecuten por cada uno de los 1000 eventos que se realizan de las comunicaciones de los otros nodos con el servidor. En la figura 88 se muestra el código XML del proceso que enlaza el evento y el nodo indicado.

```
<aut:process handler="it.unipr.ce.dsg.deus.impl.process.RectangularPulsePeriodicProcess"
    id="crearEventosRouter">
    <aut:params>
        <aut:param name="period" value="1" />
        <aut:param name="startVtThreshold" value="0" />
        <aut:param name="stopVtThreshold" value="1000" />
    </aut:params>
    <aut:nodes>
        <aut:reference id="router" />
    </aut:nodes>
    <aut:events>
        <aut:reference id="transmitirMensajeRouter" />
    </aut:events>
</aut:process>
```

Figura 88: Proceso crear eventos del router

5.3.9. Crear sensor de movimiento.

El proceso para crear un sensor de movimiento se ejecuta una vez e involucra a los nodos *sensorMovimientoH** y al evento *birthMovimiento*. Este proceso solo ejecuta una vez, aunque el evento con el que está relacionado se ejecuta seis veces, una para cada nodo. El motivo por el que el proceso solo se ejecuta una vez es porque cada uno de los nodos solo se crea al comienzo de la ejecución y no es necesario que se vuelvan a crear durante toda la simulación. En la figura 89 se muestra el código XML del proceso que enlaza el evento y el nodo indicado.

```
<aut:process handler="it.unipr.ce.dsg.deus.impl.process.RectangularPulsePeriodicProcess"
     id="crearSensorMovimiento">
    <aut:params>
        <aut:param name="period" value="1" />
        <aut:param name="startVtThreshold" value="0" />
        <aut:param name="stopVtThreshold" value="1" />
    </aut:params>
    <aut:nodes>
        <aut:reference id="sensorMovimientoH1" />
        <aut:reference id="sensorMovimientoH2" />
        <aut:reference id="sensorMovimientoH3" />
        <aut:reference id="sensorMovimientoH4" />
        <aut:reference id="sensorMovimientoH5" />
        <aut:reference id="sensorMovimientoH6" />
    </aut:nodes>
    <aut:events>
        <aut:reference id="birthMovimiento" />
    </aut:events>
</aut:process>
```

Figura 89: Proceso de creación de sensores de movimiento

5.3.10. Crear luces.

El proceso para crear una luz se ejecuta una vez e involucra a los nodos *LuzH** y al evento *birthLuz*. Este proceso solo ejecuta una vez, aunque el evento con el que está relacionado se ejecuta seis veces, una para cada nodo. El motivo por el que el proceso solo se ejecuta una vez es porque cada uno de los nodos solo se crea al comienzo de la ejecución y no es necesario que se vuelvan a crear durante toda la simulación. En la figura 90 se muestra el código XML del proceso que enlaza el evento y el nodo indicado.

```
<aut:process handler="it.unipr.ce.dsg.deus.impl.process.RectangularPulsePeriodicProcess"
    id="crearLuces">
    <aut:params>
        <aut:param name="period" value="1" />
        <aut:param name="startVtThreshold" value="0" />
        <aut:param name="stopVtThreshold" value="1" />
    </aut:params>
    <aut:nodes>
        <aut:reference id="LuzH1" />
        <aut:reference id="LuzH2" />
        <aut:reference id="LuzH3" />
        <aut:reference id="LuzH4" />
        <aut:reference id="LuzH5" />
        <aut:reference id="LuzH6" />
    </aut:nodes>
    <aut:events>
        <aut:reference id="birthLuz" />
    </aut:events>
</aut:process>
```

Figura 90: Proceso de creación de luces.

5.3.11. Crear eventos de sensor de movimiento.

Este proceso es el responsable de que se envíe la información del sensor de movimiento al servidor. Se van a realizar 1000 envíos desde cada nodo y en cada uno de ellos se enviarán todos los datos que el sensor de movimiento es capaz de procesar. En la figura 91 se muestra el código XML del proceso que enlaza el evento y el nodo indicado.

```
<aut:process handler="it.unipr.ce.dsg.deus.impl.process.RectangularPulsePeriodicProcess"
    id="crearEventosSensorMovimiento">
    <aut:params>
        <aut:param name="period" value="1" />
        <aut:param name="startVtThreshold" value="0" />
        <aut:param name="stopVtThreshold" value="1000" />
    </aut:params>
    <aut:nodes>
        <aut:reference id="sensorMovimientoH1" />
        <aut:reference id="sensorMovimientoH2" />
        <aut:reference id="sensorMovimientoH3" />
        <aut:reference id="sensorMovimientoH4" />
        <aut:reference id="sensorMovimientoH5" />
        <aut:reference id="sensorMovimientoH6" />
    </aut:nodes>
    <aut:events>
        <aut:reference id="enviarOrdenMovimientoServer" />
    </aut:events>
</aut:process>
```

Figura 91: Proceso de creación de eventos del sensor de movimiento

5.3.12. Crear sensor de temperatura.

El proceso para crear un sensor de temperatura se ejecuta una vez e involucra a los nodos *sensorTemperaturaH** y al evento *birthTemperatura*. Este proceso solo ejecuta una vez, aunque el evento con el que está relacionado se ejecuta seis veces, una para cada nodo. El motivo por el que el proceso solo se ejecuta una vez es porque cada uno de los nodos solo se crea al comienzo de la ejecución y no es necesario que se vuelvan a crear durante toda la simulación. En la figura 92 se muestra el código XML del proceso que enlaza el evento y el nodo indicado.

```
<aut:process handler="it.unipr.ce.dsg.deus.impl.process.RectangularPulsePeriodicProcess"
     id="crearSensorTemperatura">
    <aut:params>
        <aut:param name="period" value="1" />
        <aut:param name="startVtThreshold" value="0" />
        <aut:param name="stopVtThreshold" value="1" />
    </aut:params>
    <aut:nodes>
        <aut:reference id="sensorTemperaturaH1" />
        <aut:reference id="sensorTemperaturaH2" />
        <aut:reference id="sensorTemperaturaH3" />
        <aut:reference id="sensorTemperaturaH4" />
        <aut:reference id="sensorTemperaturaH5" />
        <aut:reference id="sensorTemperaturaH6" />
    </aut:nodes>
    <aut:events>
        <aut:reference id="birthTemperatura" />
    </aut:events>
</aut:process>
```

Figura 92: Proceso de creación de sensores de temperatura.

5.3.13. Crear electroválvula.

El proceso para crear una electroválvula se ejecuta una vez e involucra a los nodos *electrovalvulaH** y al evento *birthElectrovalvula*. Este proceso solo ejecuta una vez, aunque el evento con el que está relacionado se ejecuta seis veces, una para cada nodo. El motivo por el que el proceso solo se ejecuta una vez es porque cada uno de los nodos solo se crea al comienzo de la ejecución y no es necesario que se vuelvan a crear durante toda la simulación. En la figura 93 se muestra el código XML del proceso que enlaza el evento y el nodo indicado.

```
<aut:process handler="it.unipr.ce.dsg.deus.impl.process.RectangularPulsePeriodicProcess"
     id="crearElectrovalvula">
    <aut:params>
        <aut:param name="period" value="1" />
        <aut:param name="startVtThreshold" value="0" />
        <aut:param name="stopVtThreshold" value="1" />
    </aut:params>
    <aut:nodes>
        <aut:reference id="electrovalvulaH1" />
        <aut:reference id="electrovalvulaH2" />
        <aut:reference id="electrovalvulaH3" />
        <aut:reference id="electrovalvulaH4" />
        <aut:reference id="electrovalvulaH5" />
        <aut:reference id="electrovalvulaH6" />
    </aut:nodes>
    <aut:events>
        <aut:reference id="birthElectrovalvula" />
    </aut:events>
</aut:process>
```

Figura 93: Proceso de creación de electroválvulas.

5.3.14. Crear interruptor.

El proceso para crear un interruptor se ejecuta una vez e involucra a los nodos *interruptorH** y al evento *birthInterruptor*. Este proceso solo ejecuta una vez, aunque el evento con el que está relacionado se ejecuta seis veces, una para cada nodo. El motivo por el que el proceso solo se ejecuta una vez es porque cada uno de los nodos solo se

crea al comienzo de la ejecución y no es necesario que se vuelvan a crear durante toda la simulación. En la figura 94 se muestra el código XML del proceso que enlaza el evento y el nodo indicado.

```
<aut:process handler="it.unipr.ce.dsg.deus.impl.process.RectangularPulsePeriodicProcess"
    id="crearInterruptor">
    <aut:params>
        <aut:param name="period" value="1" />
        <aut:param name="startVtThreshold" value="0" />
        <aut:param name="stopVtThreshold" value="1" />
    </aut:params>
    <aut:nodes>
        <aut:reference id="interruptorH1" />
        <aut:reference id="interruptorH2" />
        <aut:reference id="interruptorH3" />
        <aut:reference id="interruptorH4" />
        <aut:reference id="interruptorH5" />
        <aut:reference id="interruptorH6" />
    </aut:nodes>
    <aut:events>
        <aut:reference id="birthInterruptor" />
    </aut:events>
</aut:process>
```

Figura 94: Proceso de creación de interruptores

5.3.15. Crear eventos de los interruptores.

Este proceso es el responsable de que se envíe la información del interruptor al servidor. Se van a realizar 1000 envíos desde cada nodo y en cada uno de ellos se enviarán todos los datos que el sensor de movimiento es capaz de procesar. En la figura 95 se muestra el código XML del proceso que enlaza el evento y el nodo indicado.

```
<aut:process handler="it.unipr.ce.dsg.deus.impl.process.RectangularPulsePeriodicProcess"
    id="crearEventosInterruptor">
    <aut:params>
        <aut:param name="period" value="1" />
        <aut:param name="startVtThreshold" value="0" />
        <aut:param name="stopVtThreshold" value="1000" />
    </aut:params>
    <aut:nodes>
        <aut:reference id="interruptorH1" />
        <aut:reference id="interruptorH2" />
        <aut:reference id="interruptorH3" />
        <aut:reference id="interruptorH4" />
        <aut:reference id="interruptorH5" />
        <aut:reference id="interruptorH6" />
    </aut:nodes>
    <aut:events>
        <aut:reference id="enviarOrdenInterruptorServer" />
    </aut:events>
</aut:process>
```

Figura 95: Proceso de creación de eventos de los interruptores

5.3.16. Crear caldera.

El proceso para crear una caldera se ejecuta una vez e involucra al nodo *caldera* y al evento *birthCaldera*. Este proceso solo ejecuta una vez porque el nodo solo se crea al comienzo de la ejecución y no es necesario que se vuelva a crear durante toda la

simulación. En la figura 96 se muestra el código XML del proceso que enlaza el evento y el nodo indicado.

```
<aut:process handler="it.unipr.ce.dsg.deus.impl.process.RectangularPulsePeriodicProcess"
    id="crearCaldera">
    <aut:params>
        <aut:param name="period" value="1" />
        <aut:param name="startVtThreshold" value="0" />
        <aut:param name="stopVtThreshold" value="1" />
    </aut:params>
    <aut:nodes>
        <aut:reference id="caldera" />
    </aut:nodes>
    <aut:events>
        <aut:reference id="birthCaldera" />
    </aut:events>
</aut:process>
```

Figura 96: Proceso de creación de caldera

5.3.17. Crear eventos del sensor de temperatura.

Este proceso es el responsable de que se envíe la información del sensor de temperatura al servidor. Se van a realizar 1000 envíos desde cada nodo y en cada uno de ellos se enviarán todos los datos que el sensor de temperatura es capaz de procesar. En la figura 97 se muestra el código XML del proceso que enlaza el evento y el nodo indicado.

```
<aut:process handler="it.unipr.ce.dsg.deus.impl.process.RectangularPulsePeriodicProcess"
    id="crearEventosSensorTemperatura">
    <aut:params>
        <aut:param name="period" value="1" />
        <aut:param name="startVtThreshold" value="0" />
        <aut:param name="stopVtThreshold" value="1000" />
    </aut:params>
    <aut:nodes>
        <aut:reference id="sensorTemperaturaH1" />
        <aut:reference id="sensorTemperaturaH2" />
        <aut:reference id="sensorTemperaturaH3" />
        <aut:reference id="sensorTemperaturaH4" />
        <aut:reference id="sensorTemperaturaH5" />
        <aut:reference id="sensorTemperaturaH6" />
    </aut:nodes>
    <aut:events>
        <aut:reference id="enviarTempServer" />
    </aut:events>
</aut:process>
```

Figura 97: Proceso de creación de eventos del sensor de temperatura

5.3.18. Crear servidor

El proceso para crear el servidor se ejecuta una vez e involucra al nodo *servidor* y al evento *birthC2*. Este proceso solo ejecuta una vez porque el nodo solo se crea al comienzo de la ejecución y no es necesario que se vuelva a crear durante toda la simulación. En la figura 98 se muestra el código XML del proceso que enlaza el evento y el nodo indicado.

```
<aut:process handler="it.unipr.ce.dsg.deus.impl.process.RectangularPulsePeriodicProcess"
    id="crearServer">
    <aut:params>
        <aut:param name="period" value="1" />
        <aut:param name="startVtThreshold" value="0" />
        <aut:param name="stopVtThreshold" value="1" />
    </aut:params>
    <aut:nodes>
        <aut:reference id="servidor" />
    </aut:nodes>
    <aut:events>
        <aut:reference id="birth2" />
    </aut:events>
</aut:process>
```

Figura 98: Proceso de creación del servidor

5.4. Engine

Este elemento es el nodo principal de la simulación usada en *DEUS*. Es el responsable de gestionar el tiempo de simulación, la semilla de creación que se usa en la aleatoriedad y el paradigma de creación de nodos. En el documento XML solo se instancian los procesos, e internamente, al estar enlazados los nodos y los eventos con los procesos se crea toda la simulación. En la figura 99 se muestra el código XML usado para la simulación de esta arquitectura.

```
<!-- COMIENZO -->
<aut:engine maxvt="2000000" seed="1234567890" prng="ISAAC">
    <aut:logger level="INFO" />
    <aut:processes >
        <aut:reference id="crearServer" />
        <aut:reference id="crearSensorTemperatura" />
        <aut:reference id="crearCaldera" />
        <aut:reference id="crearElectrovalvula"/>
        <aut:reference id="crearInterruptor"/>
        <aut:reference id="crearLuces"/>
        <aut:reference id="crearRouter"/>
        <aut:reference id="crearCamara"/>
        <aut:reference id="crearSensorBiometrico"/>
        <aut:reference id="crearSensorMovimiento"/>
        <aut:reference id="crearReloj"/>
        <aut:reference id="crearEventosSensorTemperatura" />
        <aut:reference id="crearEventosElectrovalvula"/>
        <aut:reference id="crearEventosInterruptor"/>
        <aut:reference id="crearEventosLuz"/>
        <aut:reference id="crearEventosRouter"/>
        <aut:reference id="crearEventosCamara"/>
        <aut:reference id="crearEventosSensorBiometrico"/>
        <aut:reference id="crearEventosSensorMovimiento"/>
        <aut:reference id="crearEventosReloj"/>
        <aut:reference id="crearEventosServidor"/>
    </aut:processes>
</aut:engine>
```

Figura 99: Nodo Engine

Después de que el archivo de configuración haya sido parseado, la configuración de los objetos de simulación obtenidos son pasados a la clase *Engine* que es la responsable de iniciar los eventos a ejecutar. Los eventos insertados en la cola de simulación son procesados individualmente hasta que el máximo tiempo de simulación es alcanzado o la

cola está vacía. En cada ciclo, el primer evento de la cola es eliminado, el tiempo virtual de la simulación es actualizado y el evento ejecutado. Si el evento tiene asociados eventos, estos se ejecutarán inmediatamente después de la ejecución en el mismo orden que se definieron en el archivo de configuración.

5.5. Diagrama de clases.

En este apartado voy a mostrar el diagrama de clases de la simulación. En este diagrama solo se mostrarán las clases que se ha creado y que están relacionadas de alguna manera con las clases *Node, Event* y *Process*. Debido a que un solo diagrama de clases sería demasiado confuso en la relación de los nodos y eventos con el servidor, he optado por dividir el diagrama de clases en cuatro, por un lado voy a mostrar el diagrama de clases relacionado con los eventos de climatización, a continuación mostraré el diagrama de clases relacionado con los eventos de luminosidad, después mostraré el diagrama de clases de los eventos relacionados con el acceso a la vivienda y por último mostraré el diagrama de clases relacionado con el reloj y los eventos relacionados con la salud del usuario.

En la figura 100 muestro el diagrama de clases de los eventos relacionados con la climatización. En esa figura se ve que el servidor adquiere un papel central provocado por el evento lanzado en la clase *SensorTemperaturaEvent*. Cuando el evento de envío de información de la temperatura es procesado, le llega al servidor la información y se encarga de comunicarse con los nodos *Caldera* y *Electroválvula* usando los eventos de intercambio de mensajes que genera el nodo *Router*.

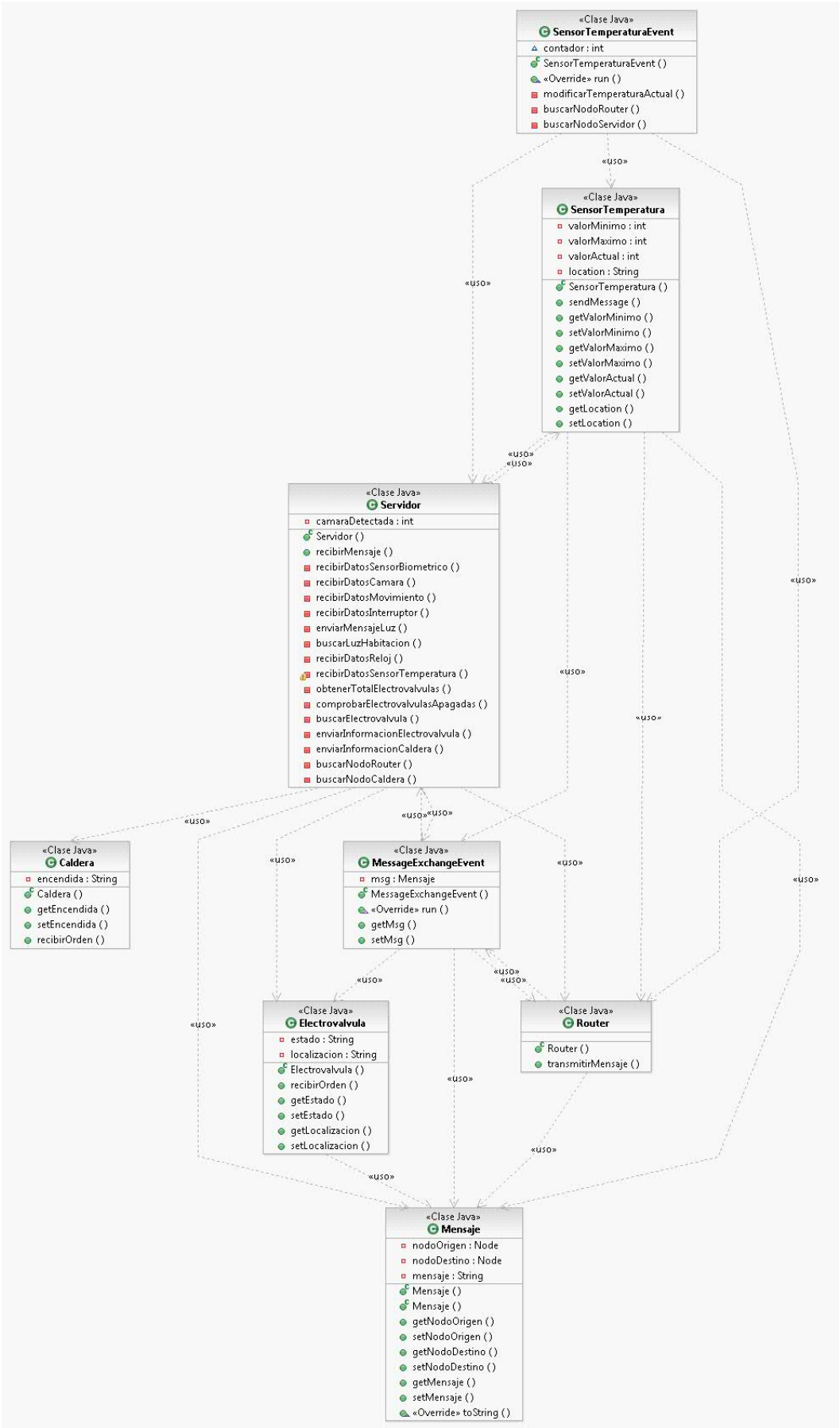


Figura 100: Diagrama de clases de los eventos relacionados con los eventos de temperatura

En la figura 101 muestro el diagrama de clases de los eventos relacionados con la luminosidad. Cuando el evento de envío de información del interruptor o del sensor de movimiento es procesado, le llega al servidor la información y se encarga de comunicarse con el nodo *Luz* usando los eventos de intercambio de mensajes que genera el nodo *Router*.

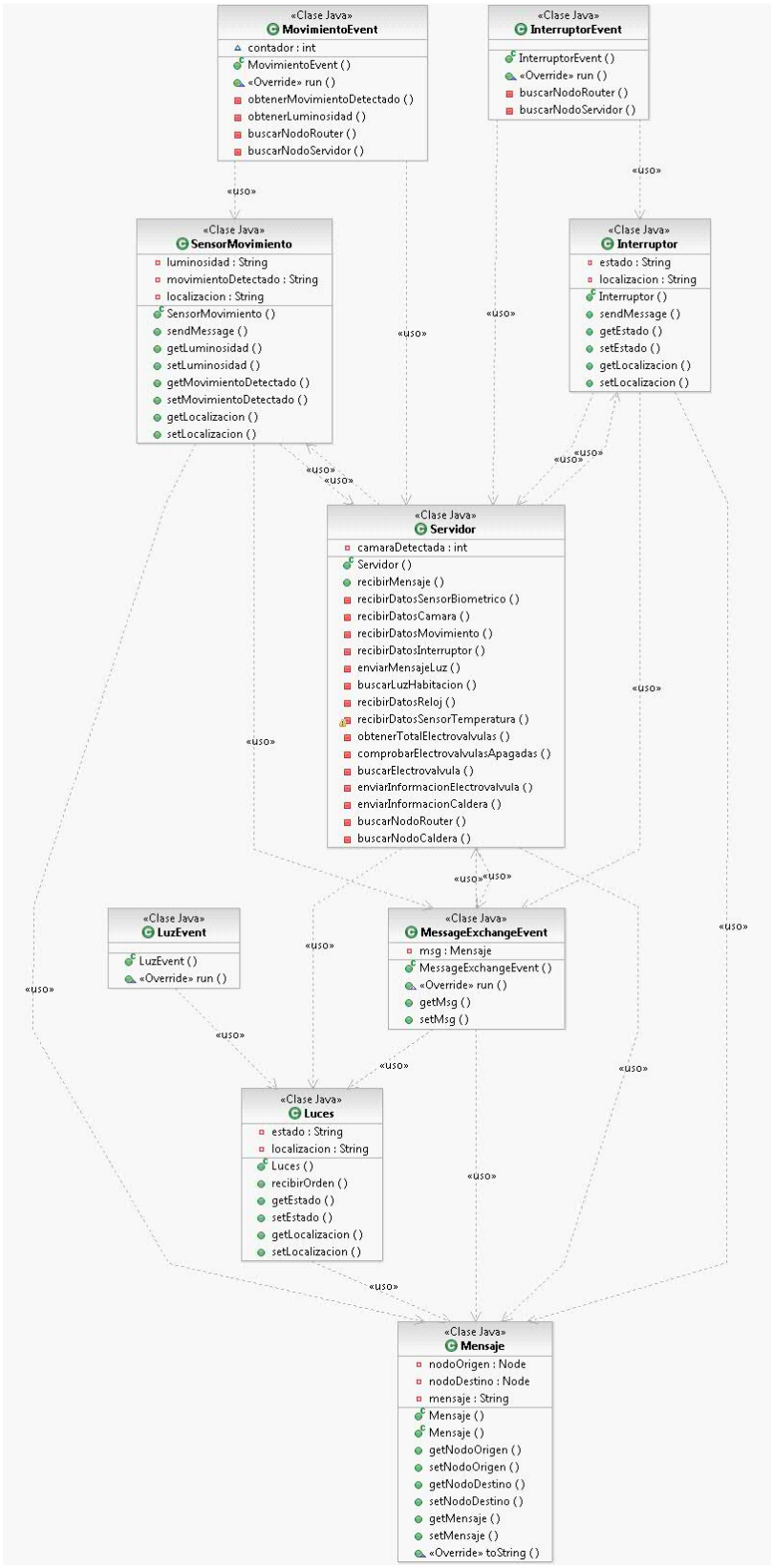


Figura 101: Diagrama de clases de los eventos del sensor de movimiento

En la figura 102 muestro el diagrama de clases de los eventos relacionados con los accesos a la vivienda. Cuando el evento de envío de información desde la cámara es procesado, le llega al servidor la información y se encarga de esperar el evento del lector biométrico usando los eventos de intercambio de mensajes que genera el nodo *Router*.

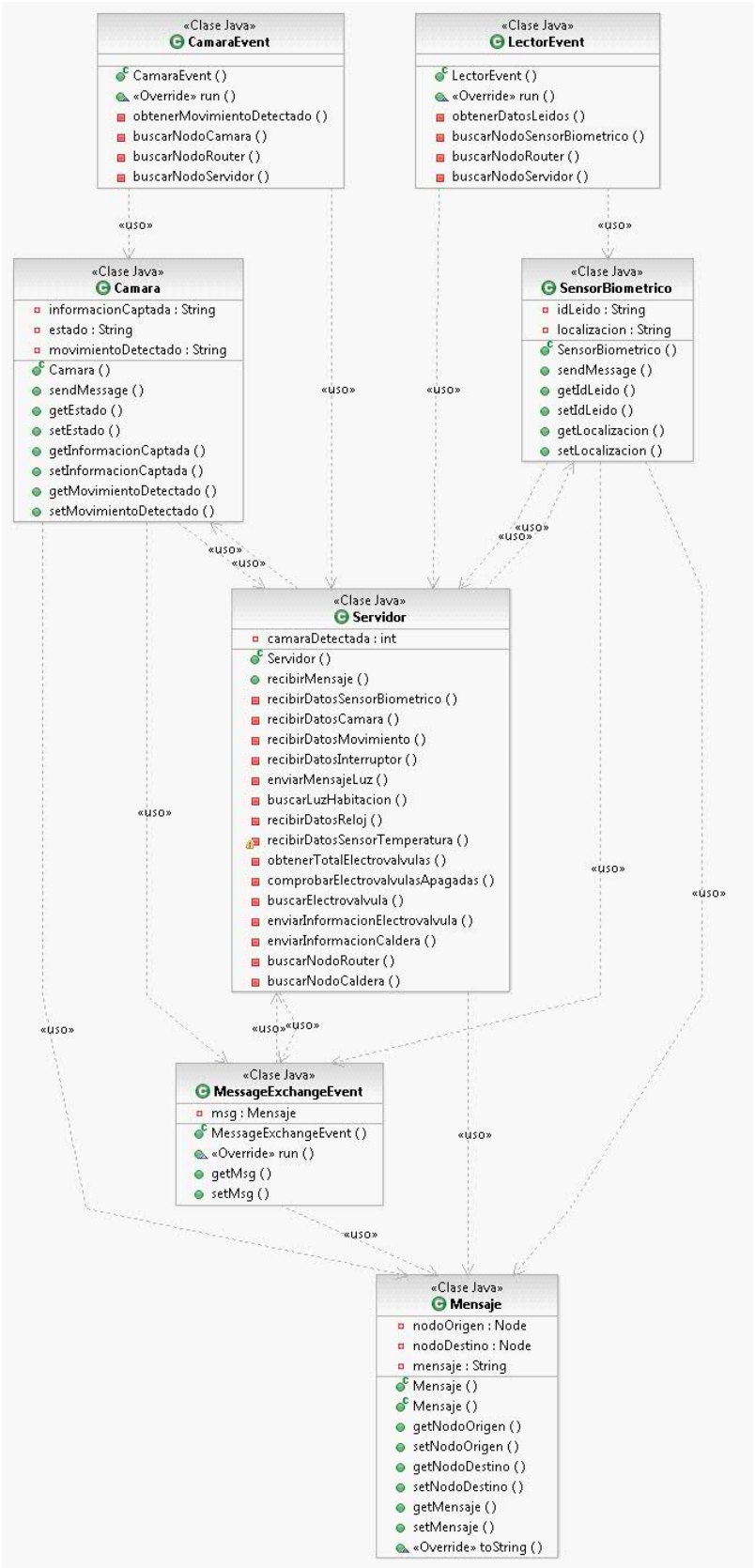


Figura 102: Diagrama de clases de los eventos de la cámara

En la figura 103 muestro el diagrama de clases de los eventos relacionados con el reloj y la salud del usuario. Cuando el evento de envío de información del reloj es procesado, le llega al servidor la información y se encarga de comunicarse con un sistema externo si es necesario usando los eventos de intercambio de mensajes que genera el nodo *Router*.

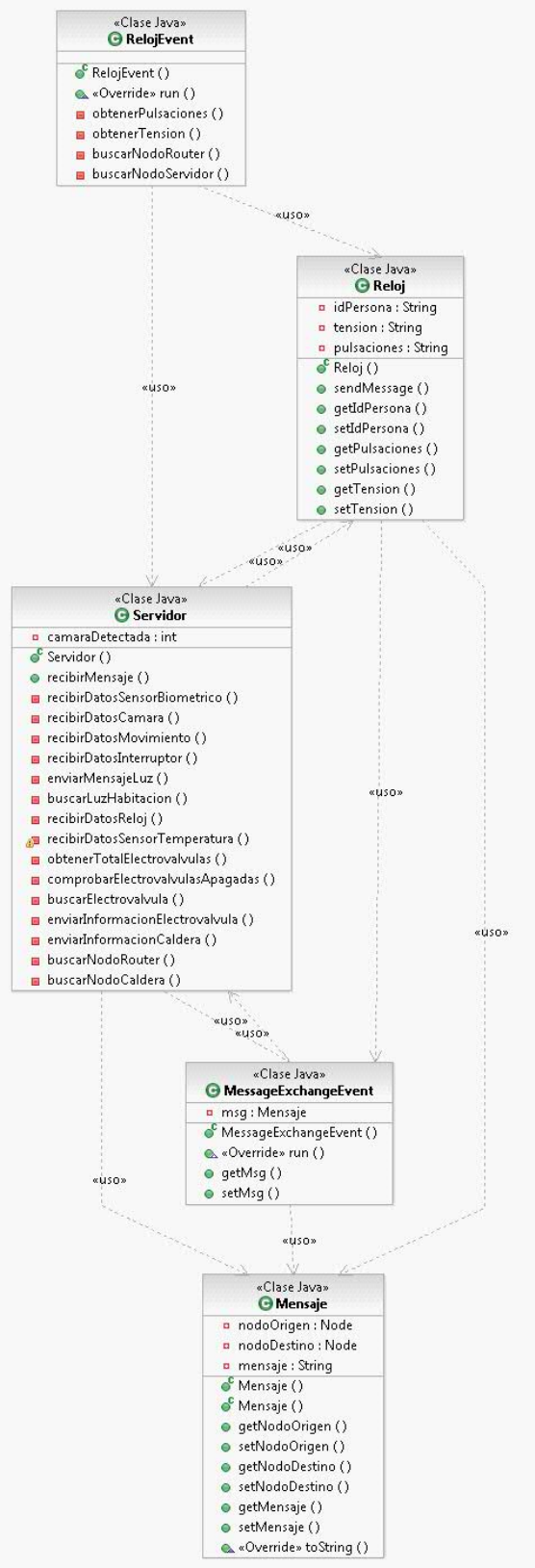
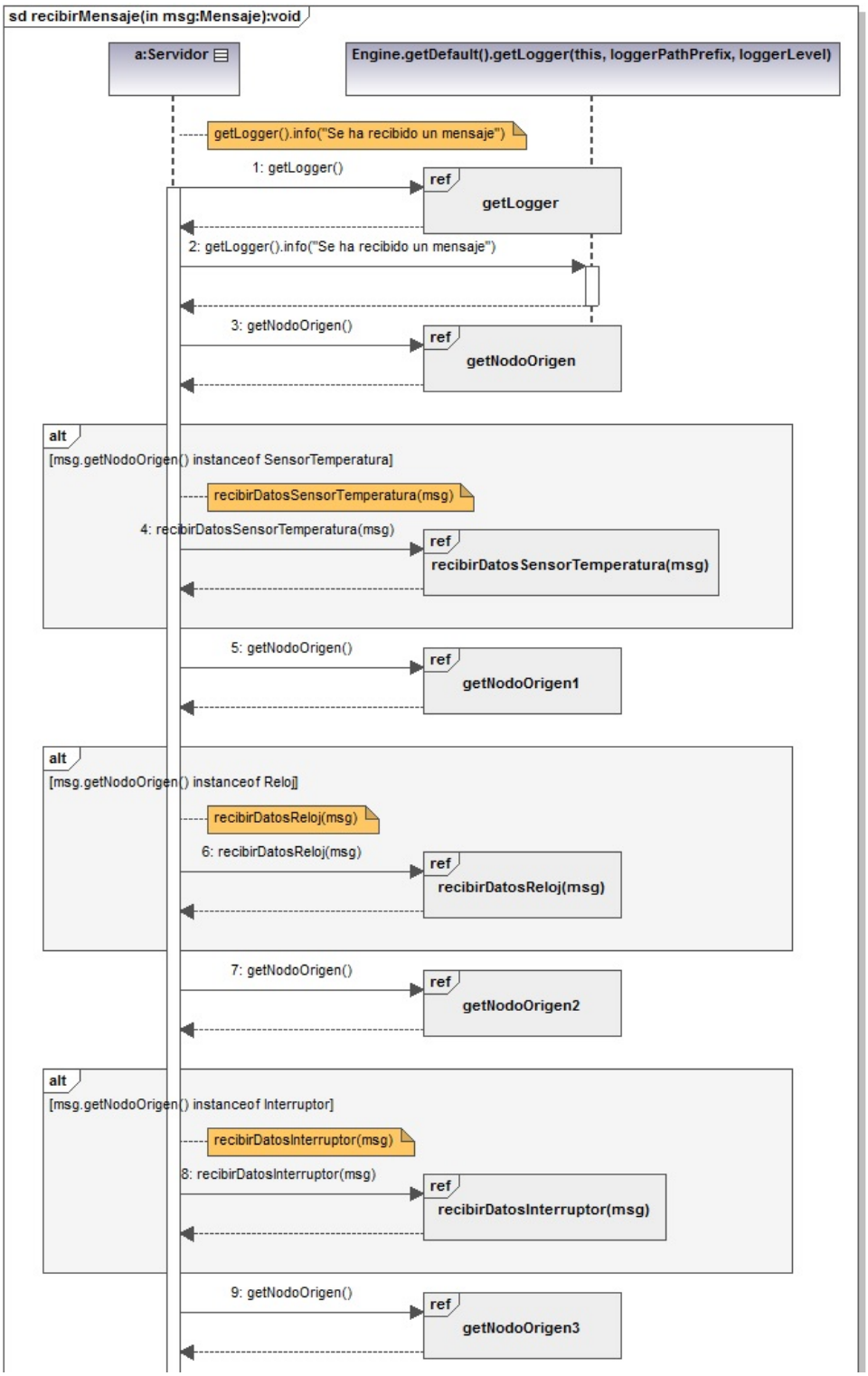


Figura 103: Diagrama de clases de los eventos relacionados con el reloj

5.6. Diagrama de secuencia.

En este apartado voy a mostrar los diagramas de secuencia más importantes de la simulación. Me he centrado en los diagramas que están relacionados con las actividades del servidor, por este motivo mostraré los diagramas de cuando el servidor recibe datos y los envía a los distintos nodos de aplicación. Además añadiré el diagrama de secuencia que muestra las llamadas que pasan por el sistema de mensajería.

En la figura 104 muestro el diagrama de secuencia de la recepción de mensajes del servidor. Cuando el servidor recibe un mensaje, lo primero que hace es comprobar de qué tipo es el nodo que lo ha enviado, y una vez que ha detectado el origen, realiza la lógica asociada a la simulación del evento asociado al nodo.



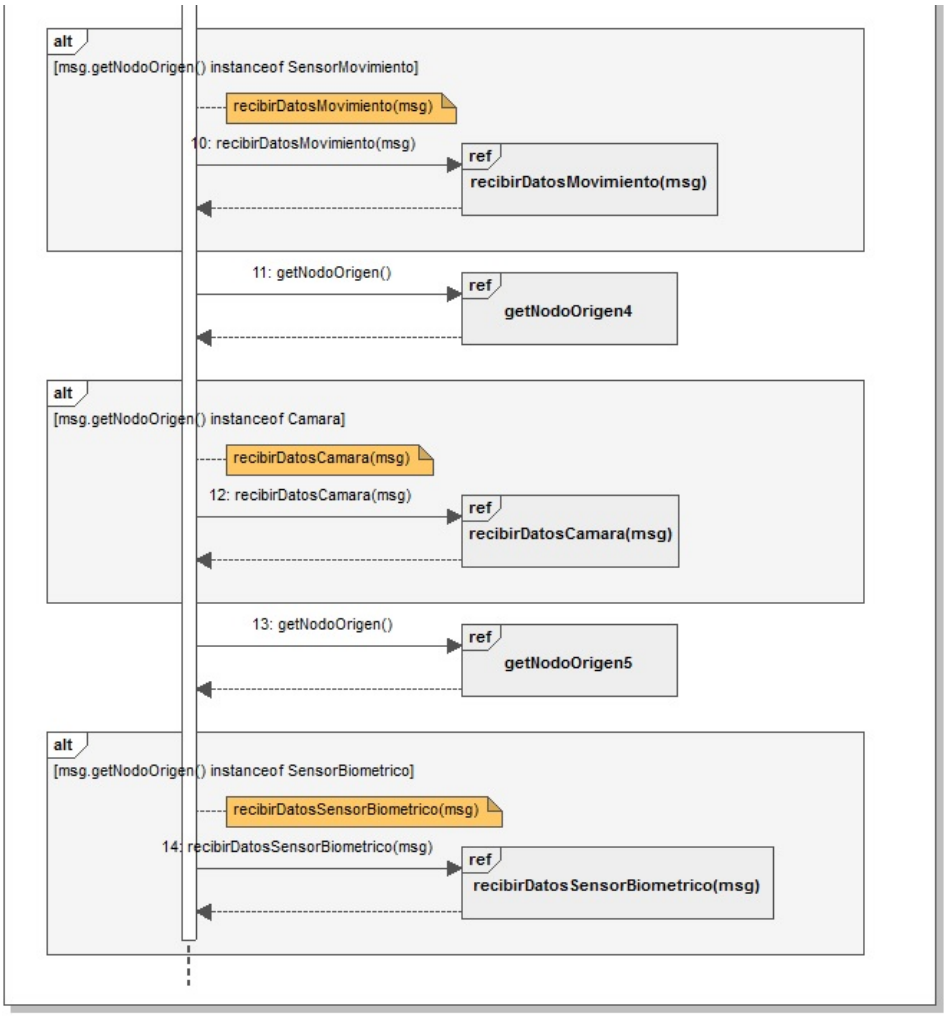


Figura 104: Diagrama de secuencia de recibir mensaje del servidor

En la figura 105 muestro el diagrama de secuencia del envío de mensajes a la electroválvula. Para el envío de la orden de activación o desactivación desde el servidor a la electroválvula se crea un mensaje de un solo envío, que pasará a través del router, con nodo origen el servidor y nodo destino la electroválvula buscada previamente.

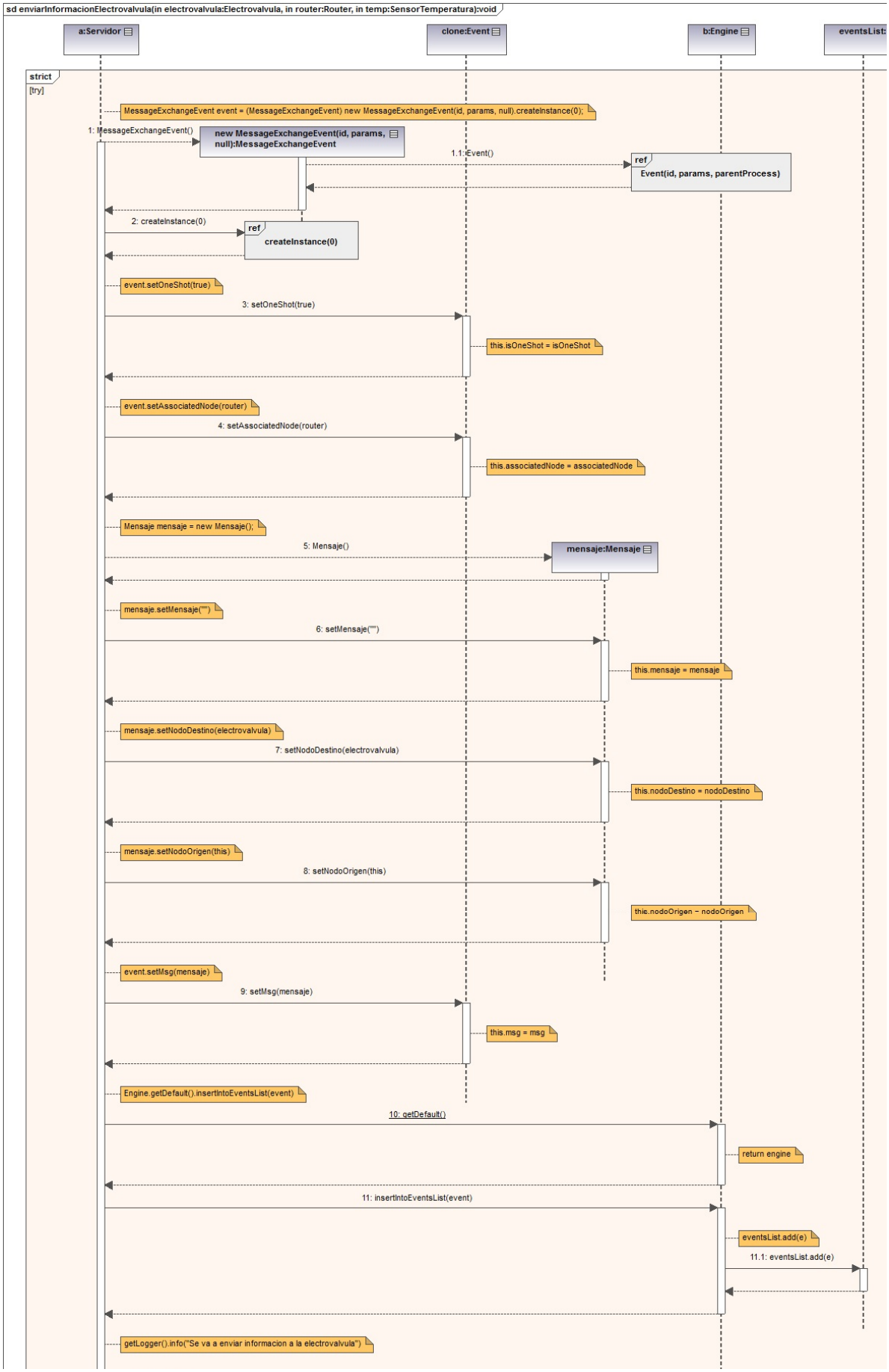


Figura 105: Diagrama de secuencia de envío de mensaje a electroválvula

En la figura 106 muestro el diagrama de secuencia del envío de mensajes a la caldera. Con un comportamiento similar al de la electroválvula realizar el servidor el envío de órdenes a la caldera. Se crea el mensaje si es necesario, indicando como nodo origen el propio servidor y como no de destino la caldera y lo envía a través del router.

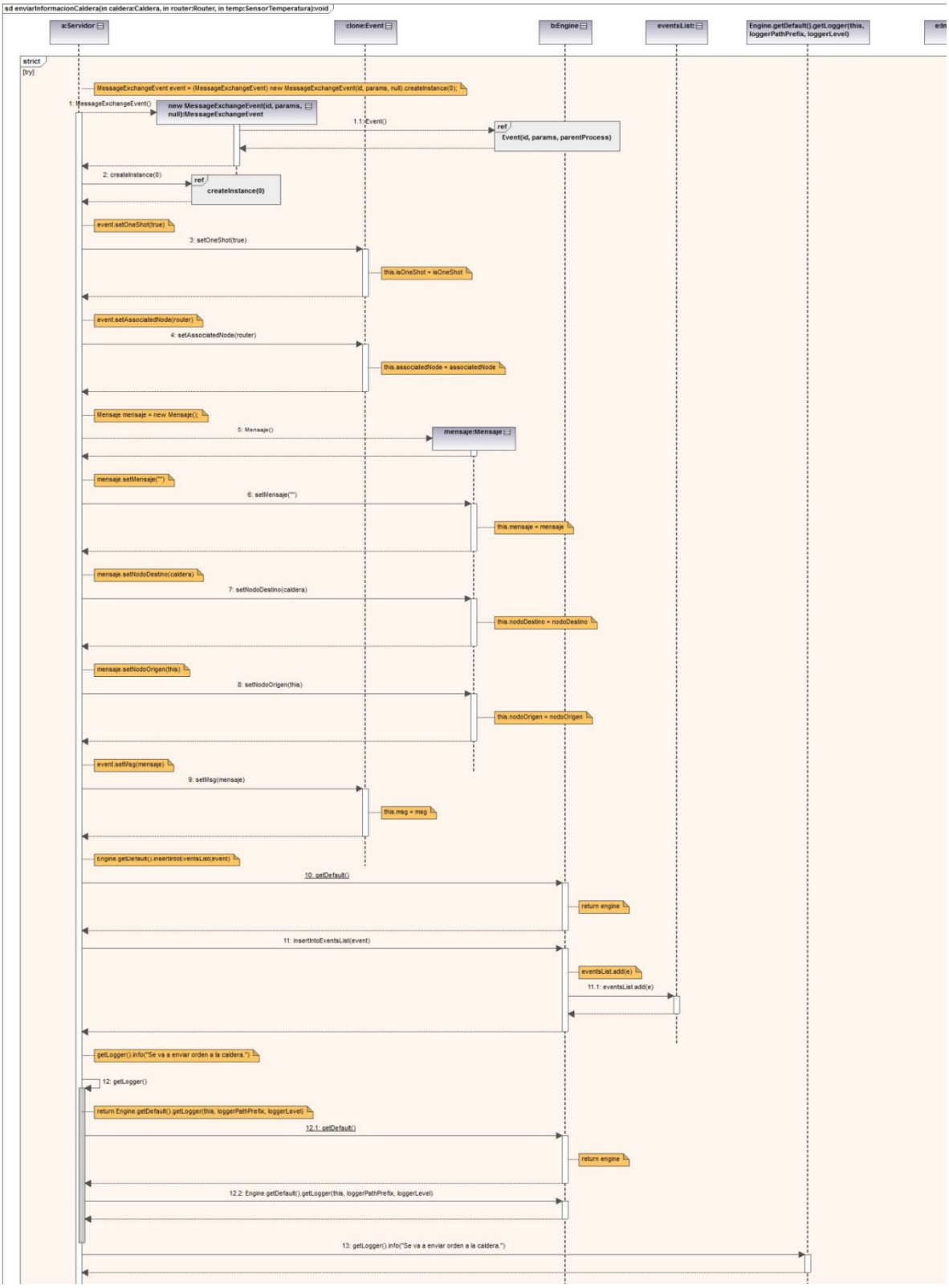


Figura 106: Envío de mensaje al servidor

En la figura 107 muestro el diagrama de secuencia del envío de mensajes a las luces del techo de las habitaciones. Para el enviar a una luz la orden de encendido o apagado, el servidor tiene que detectar el nodo origen, ya que si es el interruptor lo hará sin comprobar nada más, y si es el sensor de movimiento, comprobará la luminosidad y al órdenes dadas a través del interruptor. Una vez decidido si se envía la orden, se crea un mensaje donde el nodo destino es la luz donde están los nodos que han provocado el envío de información y el nodo origen el servidor.

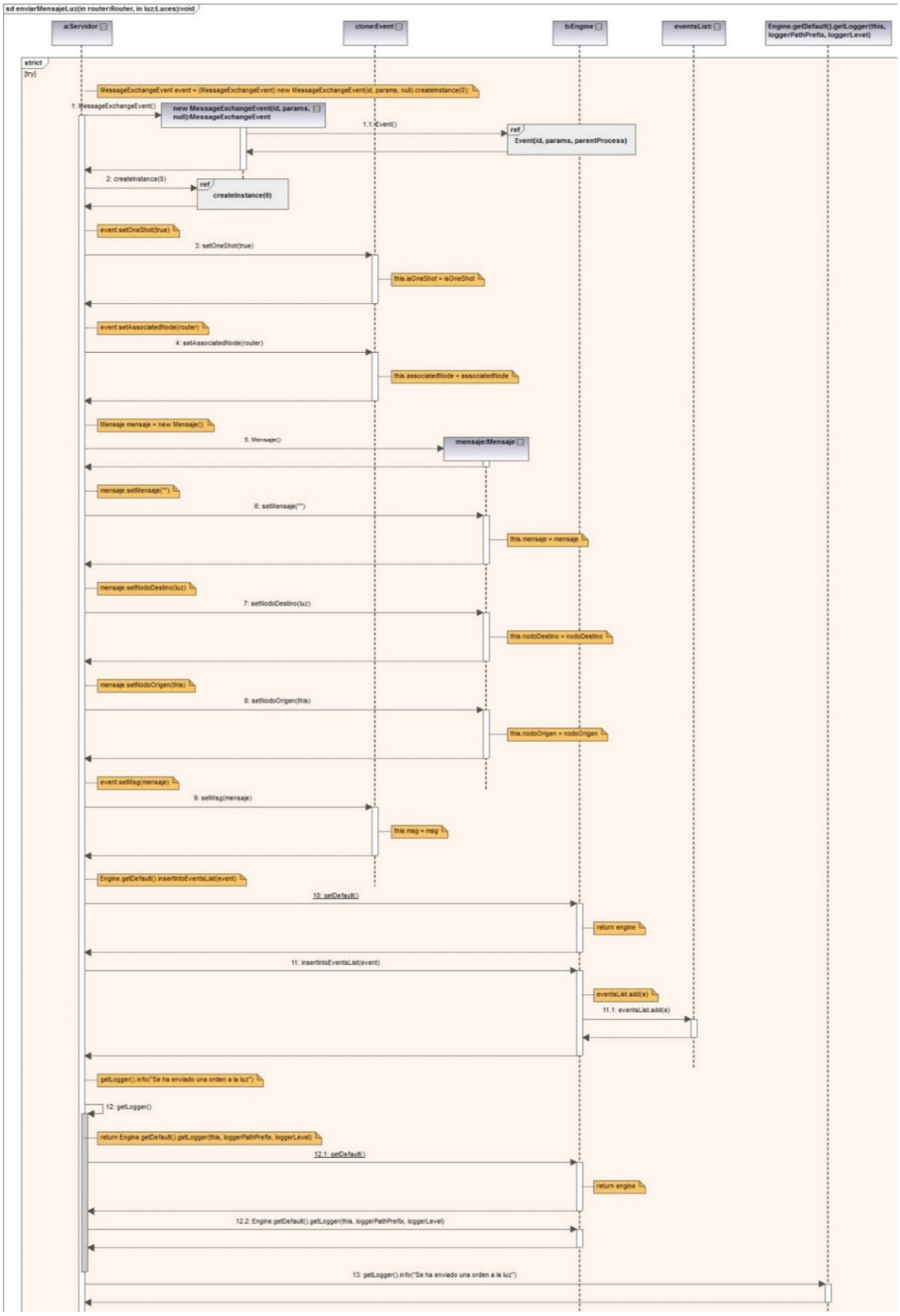


Figura 107: Enviar mensajes a las luces

Debido al tamaño del diagrama de secuencia que representa el comportamiento del envío de mensajes de la simulación, he adjuntado a este documento una imagen con el nombre *diagramaSecuenciaMensajes.jpg* que muestra el diagrama citado.

5.7. Resultados de la simulación

Durante toda la simulación he ido añadiendo logs en el envío de mensajes para poder monitorizar que la aplicación se comporta correctamente. Los archivos de logs se generan por cada tipo de nodo y en cada mensaje se muestra la hora y la acción que se va a llevar a cabo. Durante la simulación he tenido el problema de que al ocurrir demasiados eventos simultáneamente, da la impresión de que ocurren varias cosas a la vez, cuando en realidad son consecuencia de acciones anteriores.

5.7.1. Resultados de la simulación de los eventos de temperatura.

En el caso de la simulación de la temperatura de la habitación, he definido una función ondulatoria que se mueve entre los valores mínimos y máximos indicados en el documento XML de la simulación. Dichos valores son enviados cada minuto, tomando el ciclo completo 1440 minutos (1 día). En concreto, la función que controla la variación de la temperatura es:

$$20 + \left(10 * \left(\cos \frac{t * 2 * 3.14}{1440}\right)\right)$$

Donde *t* es el número de veces que se ha ejecutado el proceso de envío de datos del termómetro. Para obtener la formula he simplificado el comportamiento de la temperatura para suponer que todos los días tienen el mismo ciclo de temperatura, obviamente en la vida real esto no ocurre así, pero para el objetivo de la simulación podemos aceptar dicha simplificación. Cuando la temperatura de la habitación llega al máximo indicado, el servidor cierra la electroválvula y si es la última habitación que ha llegado al máximo, apaga la caldera. El caso contrario, cuando la temperatura es mínima, el servidor manda abrir la electroválvula y encender la caldera. En la Figura 108 se muestra una gráfica de la evolución de la temperatura en una de las habitaciones.

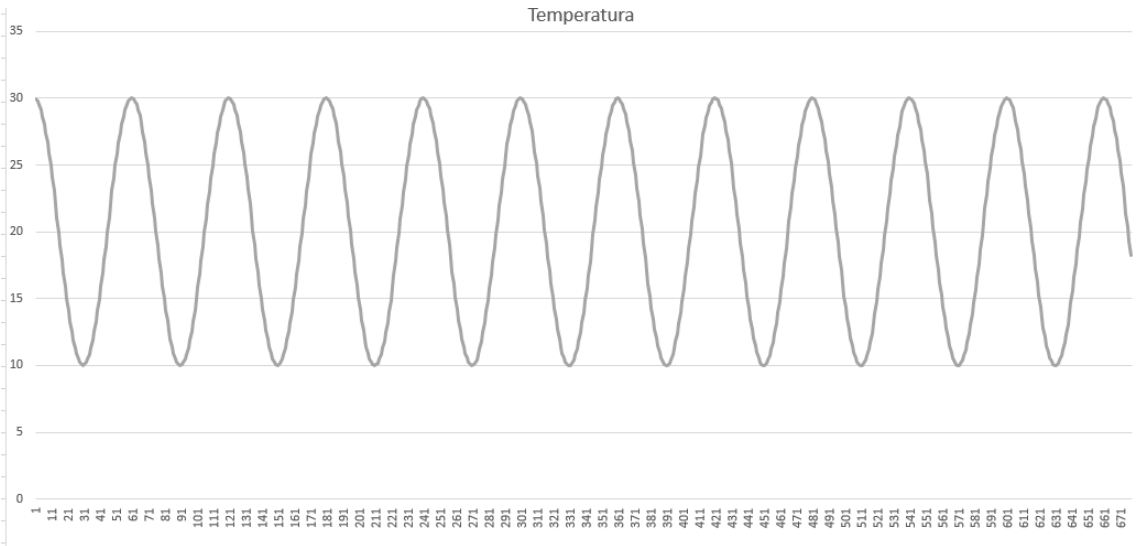


Figura 108: Evolución de la temperatura.

Para seguir el rastro de las trazas es necesario y observar el comportamiento de la simulación, es necesario utilizar el fichero *archivo.log* del nodo *Servidor, SensorTemperatura, Caldera* y *Electrovalvula.* Cuando el nodo sensor de temperatura va a enviar un mensaje se informa como se muestra a continuación:

- *26/05/2015 22:23:31 Se va a enviar la temperatura de la habitacion H4 que es 10*

Este mensaje es recibido por el servidor que muestra los siguientes mensajes:

- *26/05/2015 22:23:31 Se han recibido datos del sensor de temperatura.*
- *26/05/2015 22:23:31 Se ha encontrado la electroválvula buscada en la posición 7*
- *26/05/2015 22:23:31 La temperatura es muy baja*

Cuando el servidor ha detectado que la temperatura es muy baja, se envía un mensaje a la electroválvula que lo muestra así:

- *26/05/2015 22:23:31 Se ha cambiado el estado de la electroválvula de la habitación: H4. El nuevo estado es: 1*

Por último, la caldera mostrará el siguiente mensaje cuando se encienda:

- *26/05/2015 22:23:31 Se ha cambiado el estado de la caldera, el nuevo estado es: 1*

De manera equivalente se muestran los datos cuando se va a apagar la caldera y cerrar las electroválvulas.

5.7.2. Resultado de los eventos del sensor de luminosidad.

Para realizar la simulación de la luminosidad de la habitación, he definido una función ondulatoria que se mueve entre los valores mínimos y máximos indicados en los nodos de la simulación. Para estudiar la luminosidad, la función tiene un periodo de 24 porque se va a aceptar la hipótesis de que durante el tiempo de la simulación, no hay cambios apreciables en la luminosidad de pocos días seguidos. La función que controla la variación de la luminosidad es:

$$50 + 50 * \left(\cos\left(\frac{t * 2 * 3.14}{24}\right)\right)$$

En la función anterior *t* es el número de veces que se ha ejecutado el proceso de envío de datos desde el sensor de movimiento al servidor. Para obtener la formula he simplificado el comportamiento de la luminosidad para suponer que todos los días tienen el mismo ciclo de luz, obviamente en la vida real esto no ocurre así (en verano los días son más largos que en invierno), pero para el objetivo de la simulación podemos aceptar dicha simplificación. El objetivo de tener control de la luminosidad es para que cuando el sensor de movimiento detecta que hay alguien en la habitación mande al servidor que hay movimiento, y si el valor de la luminosidad es menor de 50, enviará la orden de que se encienda la luz. Si la luminosidad es mayor o igual a 50, verificará que la luz este apagada y en caso de que no lo esté, mandara la orden de que se apague. Para la simulación del movimiento no he utilizado ninguna función, y uso simplemente el generador de números aleatorios de Java, si el valor es inferior a 0.5, se interpreta como que no hay movimiento, y si es mayor o igual a 0.5 se interpreta como que hay movimiento. Además, el servidor tiene que tener en cuenta las órdenes recibidas por los nodos *Interruptor* ya que éstos son acciones físicas del usuario y se debe interpretar que tienen prioridad sobre las configuraciones pre-establecidas. Esto implica, que si el *Interruptor* envió la orden de apagar al *Servidor*, aunque el nodo *sersorMovimiento*

detecte movimiento y poca luminosidad no debe encender la luz de la habitación. En la figura 109 se muestra una gráfica de la evolución de la luminosidad en una de las habitaciones.

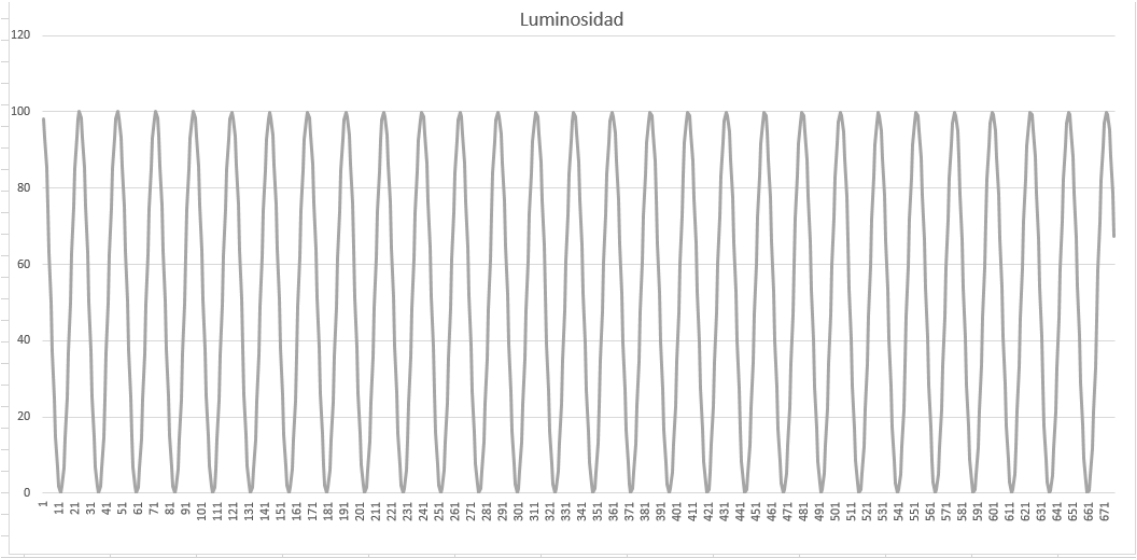


Figura 109: Evolución de la luminosidad

Al igual que con el sensor de temperatura, para seguir el rastro y observar el comportamiento de la simulación, es necesario utilizar el fichero *archivo.log* del nodo *Servidor*, *Luz*, *SersorMovimiento* e *Interruptor*. Cuando el nodo sensor de movimiento va a enviar un mensaje se informa como se muestra a continuación:

- *27/05/2015 23:38:16 Se va a enviar un mensaje del sensor de movimiento de la habitación: H1, con la luminosidad: 50 y movimiento: 1*

Este mensaje es recibido por el servidor que muestra:

- *27/05/2015 23:38:19 Se han recibido datos del sensor de movimiento.*
- *27/05/2015 23:38:19 Se ha encontrado una luz*
- *27/05/2015 23:38:19 Se ha enviado una orden a la luz*

Cuando el servidor ha detectado que la luminosidad es baja y hay movimiento, envían un mensaje a la luz:

- *27/05/2015 23:38:16 Se ha cambiado el estado de las luces para la habitación: H6, el nuevo estado: 1*

El servidor también puede recibir datos del interruptor que los muestra:

- *27/05/2015 23:38:19 Se han recibido datos del interruptor*

El mensaje que se envía a las luces no distingue el nodo origen, ya que no tiene sentido que esos nodos sepan quien dio la orden originalmente.

5.7.3. Resultado de los eventos del sensor biométrico.

Para los eventos relacionados con la cámara y el control de acceso se involucran el nodo *Camara*, *SensorBiometrico* y *Servidor*. Para la simulación de este comportamiento se ha optado por la aleatoriedad de los accesos detectados por la cámara y de la lectura o no por parte del sensor biométrico. Debido a la aleatoriedad del comportamiento, no tengo ninguna forma de predecir que va a ocurrir, que es lo que realmente pasa en la vida real. No obstante, se pueden utilizar los logs de la simulación para analizar el comportamiento de la misma.

Cuando la cámara detecta movimiento, el mensaje que se muestra es:

- *27/05/2015 23:38:16 El movimiento detectado es: 1*
- *27/05/2015 23:38:16 La camara ha captado movimiento.*

Cuando el sensor biométrico detecta que se han leído los datos de una persona, se muestra el siguiente mensaje:

- *27/05/2015 23:38:16 Se va a enviar informacion al servidor: 1*

Por último, cuando el servidor recibe el movimiento de la cámara y el del sensor biométrico muestra los siguientes mensajes:

- *27/05/2015 23:38:19 Se han recibido datos de la camara*
- *27/05/2015 23:38:19 Mandar información a emergencias*

5.7.4. Resultado de los eventos del reloj.

El último comportamiento simulado es el control de las constantes vitales de las personas que hay dentro de la vivienda utilizando un reloj de la pulsera. Para esta simulación he creado una función que elige una valor aleatorio entre los valores permitidos de la tensión y de las pulsaciones, además, de manera aleatoria con una probabilidad del 0.01% (en cada uno de los atributos) se elige un valor al azar que está fuera del rango, de manera que el servidor envía una petición de socorro a los servicios de emergencias.

Para estos eventos de simulación se utilizan los nodos *Reloj* y *Servidor*, cuyos archivos de logs son los que utilizaremos para seguir el comportamiento de la simulación. Cuando el nodo *Reloj* tiene que comunicarse muestra el mensaje que se muestra a continuación:

- *27/05/2015 23:36:01 Se va a enviar un mensaje desde el reloj al server con las siguientes caracteristicas: tension: 120, pulsaciones: 40*

Cuando el servidor recibe los datos del reloj muestra el siguiente mensaje:

- *27/05/2015 23:36:01 Mandar información a emergencias*

6. Conclusiones y trabajos futuros

Es este apartado se va a realizar un pequeño resumen de cada uno de los apartados anteriores, destacando las conclusiones que se pueden obtener de cada uno de ellos, así como una conclusión general de la realización del proyecto. Para finalizar se han presentado unas líneas de mejoras y trabajo futuro, las cuales se detallarán a los largo del capítulo.

6.1. Conclusiones.

A lo largo del desarrollo del proyecto he cumplido los objetivos que se marcaron al comienzo del mismo. Gracias a este IoT de las cosas, centrándome en las comunicaciones entre los distintos nodos, y además, me ha servido para empezar a comprender como funcionan algunas de las herramientas más comunes para simular el IoT.

El primer y segundo apartado sirve para introducirnos en el entorno del proyecto, se hizo un breve resumen de los objetivos que se perseguían, se explicó el ámbito del trabajo, la metodología a seguir y las herramientas a utilizar.

El tercer apartado sirve para que el lector tenga una base teórica de lo que es el IoT, de los distintos protocolos de comunicación que se pueden utilizar. Además, se presentan las herramientas de simulación más comunes y se explican los ámbitos más comunes donde se utilizan dichas herramientas.

El cuarto apartado explica los elementos de la red que se va a simular, para ello se especifican todos los elementos que se van a utilizar, para después explicar las diferentes opciones que se tendrían en la vida real para que se comunicaran entre sí.

El quinto apartado es el más importante de este trabajo, en él se explican cómo se simula cada uno de los nodos, a continuación se explican los eventos y los procesos que utilizan los nodos y los eventos para simular el comportamiento deseado. Este apartado me ha servido para entender cómo funciona la aplicación *DEUS* para simular el IoT, está aplicación aún está en desarrollo, lo que implica que tenga algunos bugs que seguramente serán corregidos en versiones posteriores. Uno de los bugs más importantes que he encontrado es la falta de control que hay para elegir que dos eventos que no están relacionados entre sí, ocurran a la vez. Esto ocasiona que a la hora de ver los resultados de la simulación, da la impresión de que los eventos ocurren de manera secuencial, es decir, que primero se producen todos los cambios posibles de temperatura, con las consecuencias establecidas, y después comenzaban a encenderse y apagarse las luces en función de si se detectaba o no movimiento.

Por último, hay que aclara que el mundo del IoT, al ser un área tan amplia es imposible recoger en un solo trabajo todas las características y posibilidades que ofrece.

6.2. Trabajo futuro.

Una vez que he finalizado el trabajo, y revisado las limitaciones que he tenido debido a la herramienta y al propio tiempo que he tenido para hacerlo, he listado varias mejoras que se podrían aprovechar en futuros desarrollos:

- Mejorar el control de los sensores de temperatura: Durante la simulación se ha detallado cómo se comportan los sensores de temperatura, las electroválvulas y la caldera, con el objetivo de la eficiencia energética, se podrían tener en cuenta también los sensores de movimiento, de manera que si una habitación está vacía no se encienda la calefacción en esa habitación.

- Añadir zonas con accesibilidad restringida a la vivienda: En esta simulación solo hay una zona donde controla el acceso y es la puerta principal, se puede utilizar un funcionamiento similar para impedir que, por ejemplo, los niños no entren a la cocina si no hay nadie allí, de manera que no estén en peligro de encender el fuego de la cocina o utilizar cuchillos o tijeras.
- Añadir otros elementos a la simulación: Por ejemplo, añadir nodos para simular el aire acondicionado y configurarlo para que no se pueda encender a la vez que la calefacción.
- Añadir control vía smartphone: Debido a la limitación de tiempo no he podido desarrollar esta característica como un extra. El objetivo de esta característica es que los dueños de la vivienda o las personas autorizadas puedan controlar en todo momento el comportamiento de los nodos de aplicación en función de la información que obtienen gracias a los nodos finales, además podrían controlar quien está en la casa en todo momento.
- Implementar un *histórico*: Esta línea de trabajo futuro consiste en crear una base de datos a la que se conectaría el servidor y donde se almacenarían todos los eventos que se producen, gracias a esta base de datos, el usuario podría consultar a qué hora se encendió una luz o la calefacción, lo que se serviría para controlar el cuándo se produce el gasto.
- Integrar con sistemas externos del IoT: Esta línea de trabajo consiste en utilizando la información que hay disponible en las *Smart-cities* programar el comportamiento de los nodos de aplicación que hacen que la vivienda esté confortable cuando el usuario llega a casa. Por ejemplo, que utilizando la información del tráfico, sea capaz de encender la calefacción cuando faltan unos 20 minutos para llegar a casa, y que la encienda en las habitaciones que más frecuenta el usuario.

7. Bibliografía

En este apartado listo la bibliografía y los enlaces a los que he ido haciendo referencia a lo largo de todo el documento.

1. Karl Aberer, Manfred Hauswirth, Ali Salehi; 2006; A Middleware For Fast And Flexible Sensor Network Deployment; VLDN Endowment; Páginas 1199-1202.
2. Louie F. Cervantes, Young-Seok Lee, Hyunho Yang, Jaewan Lee; 2007; A Hybrid Middleware for RFID-based Parking Management System using Group Communication in Overlay Networks; '07 Proceedings of the The 2007 International Conference on Intelligent Pervasive Computing; Páginas 521-526
3. Suparna De, Payam Barnaghi, Martin Bauer, Stefan Meissner; 2012; Service Modeling for the Internet of Things; Proceedings of the Federated Conference on Computer Science and Information System; Páginas 949-955
4. Stelios Sotiriadis, Nik Bessis, Eleana Asimakopoulou, Navonil Mustafee; 2014; Towards Simulating the Internet of Things; 28th International Conference on Advanced Information Networking and Applications Workshops; Páginas 444-448.
5. Giacomo Brambilla, Marco Picone, Simone Cirani, Michele Amoretti, Francesco Zanichelli; 2014; A Simulation Platform for Large-Scale Internet of Things Scenarios in Urban Environments; Proceedings of the First International Conference on IoT in Urban Space; Páginas 50-55
6. W3C SSN Incubator Group Report; 2011;
7. N. Gershenfeld, R. Krikorian, and D. Cohen. "The Internet of Things". Scientific Am., vol. 291, no. 4, 2004, Páginas. 46–51.
8. https://www.eclipse.org/downloads/
9. http://deus.googlecode.com/svn/trunk/
10. Andreas Haeberlen, Marcel Dischinger, Krishna P. Gummadi, Stefan Saroiu; 2006; Monarch: A Tool to Emulate Transport Protocol Flows over the Internet at Large; IMC ’06; Páginas 25-27
11. S, Sotiriadis, N. Bessis, N. Antonopoulos, A. Anjum; 2013; SimIC: Designing a New Inter-cloud Simulation Platform for Integrating Large-Scale Resource Management; Advanced Information Networking and Applications; Páginas 90-97.
12. Jayavardhana Gubbi, Rujkumar Buyya, Slaven Marusic, Marimuthu Palaniswami:
13. M. Picone; 2015; Deus: A simple tool for complex simulation; Advanced Technologies for Intelligent Transportation Systems, 139;
14. Michelle Amoretti, Matteo Agosti, Francesco Zanichelli; 2009; DEUS: a Discrete Event Universal Simulator; SIMUTools 2009 March; Páginas 2-6
15. Marco Picone, Michele Amoretti, Francesco Zanichelli; 2012; Simulating Smart Cities with DEUS; Simutools 2012, March; Páginas 19-23
16. Michele Amoretti, Marco Picone; Francesco Zanichelli, Gianluigi Ferrari; 2013; Simulating Mobile and Distribuided System with DEUS and ns-3; 978-1-4799-0838-7
17. Edgar Perk, Nikola Bogunović; 2004; Formal Verification of Logical Link Control and Adaption Protocol; IEEE MELECON 2004, May; Páginas 12-15

18. 1999; Logical Link Control and Adaptation Protocol Specification; Bluetooth Specification Version 1.0 A; Páginas 247-320
19. Giacomo Brambilla, Marco Picone, Simone Cirani, Michele Amoretti, Francesco Zanichelli; 2014; A simulation Platform for Large-Scale Internet of things Scenarios in Urban Environments; Urb-IoT 2014, October; Páginas 27-28.
20. Jorge Malla; 2013; Servicios Web; www.researchgate.net; http://www.researchgate.net/profile/Jorge_Luis_Malla_Sanchez/publication/262688345_Servicios_Web/links/02e7e53878d7fe0ac7000000.pdf
21. Pablo Valledor Pellicer; 2006; Servicios Web Semanticos; Universidad de Oviedo; http://di002.edv.uniovi.es/~cueva/asignaturas/doctorado/2006/trabajos/sws.pdf

ANEXOS

ANEXO I: Ejecución de la simulación

El objetivo de este anexo es explicar cómo ejecutar una simulación utilizando DEUS. Para ello disponemos de dos alternativas:

Utilizando eclipse: Esta alternativa consiste en importar el código fuente a un *Workspace* de eclipse y ejecutar el código desde la clase *DeusAutomatorFrame*. Para hacerlo se crea una configuración de ejecución para la clase indicada. En la figura 110 se muestra la pestaña de configuración principal.

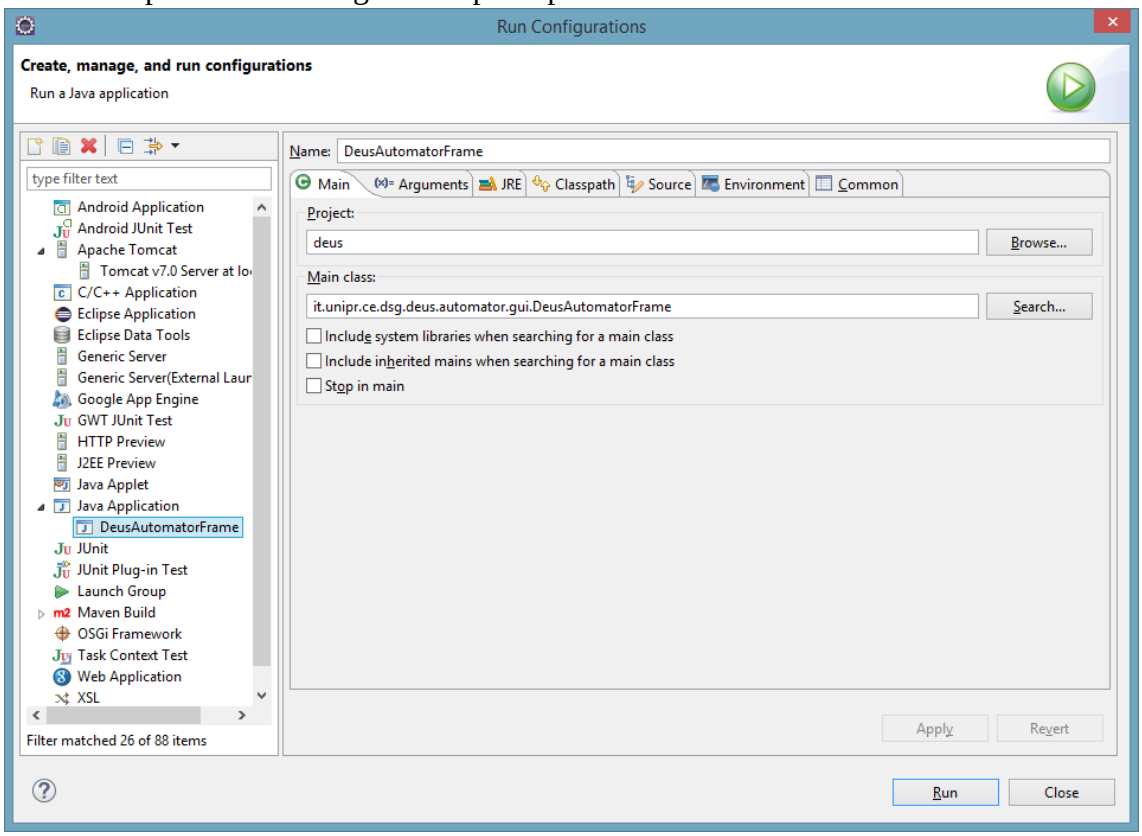


Figura 110: Configuración principal

Además, es muy importante indicar por parámetros los archivos de configuración necesarios para la ejecución de la simulación, ya que sin ellos, el simulador no tendría la definición de los nodos, eventos y procesos a simular. En la figura 111 se muestra la pestaña de la configuración donde se indican los archivos necesarios. El segundo archivo que se informa como parámetro no es usado en la simulación, no obstante hay que añadirlo para cumplir con los requisitos de *DEUS*.

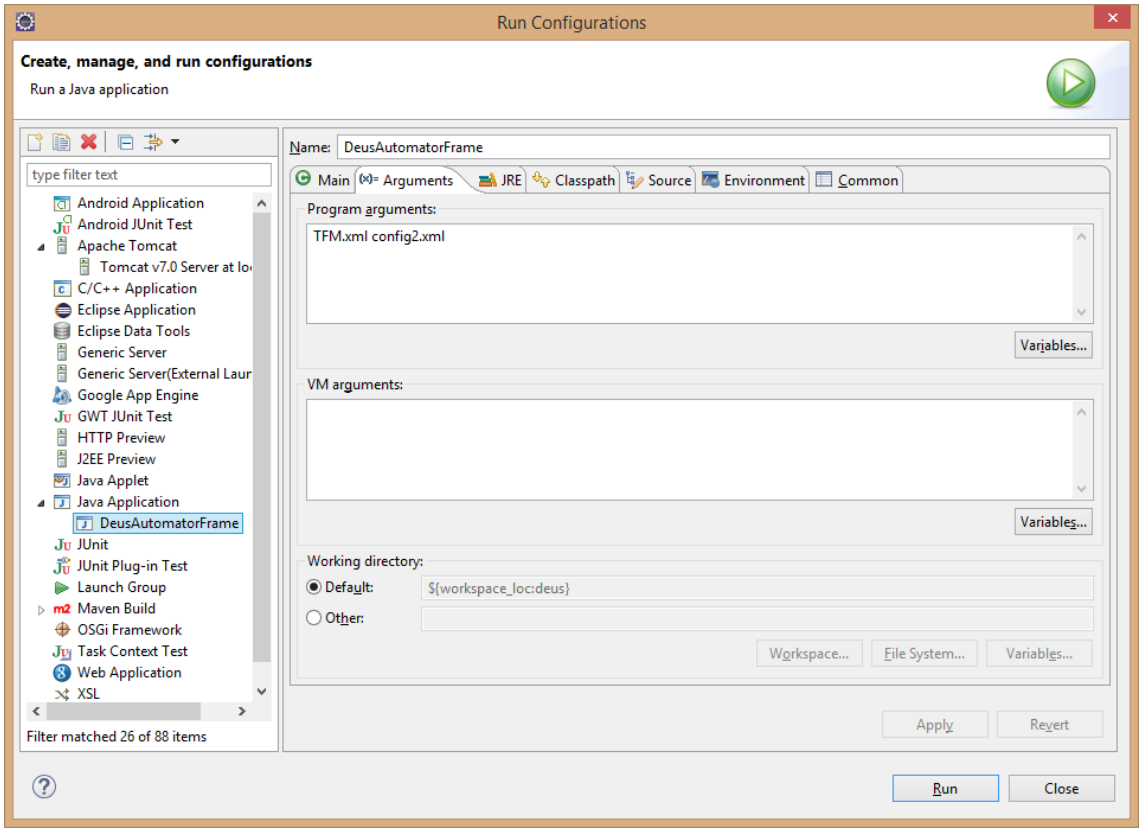


Figura 111: Pestaña de ficheros de configuración

Una vez que está la configuración creada, se ejecutará para ver la ventana de inicio. Esta ventana nos da la posibilidad de definir parámetros de los nodos, procesos, etc. Como estamos usando un archivo XML con toda la información, solo es necesario pulsar el botón etiquetado con *run*. En la figura 112 se muestra la ventana inicial y se remarca el botón *run*.

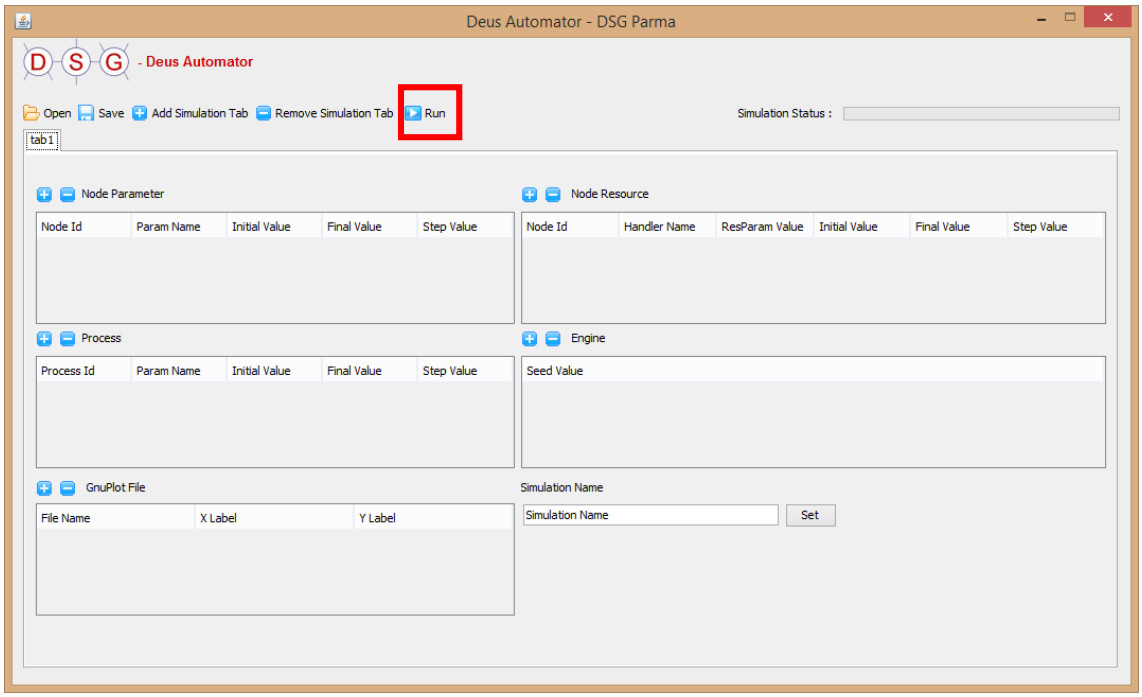


Figura 112: Ventana principal DEUS

Después de pulsar el botón *run* aparecerá otra ventana destinada a mostrar elementos gráficos. En este trabajo no se ha desarrollado esa posibilidad, por lo que no aportará ninguna información, solo hay que pulsar el botón *Start*. En la figura 113 se muestra esta ventana y se ha remarcado el botón *Start*.

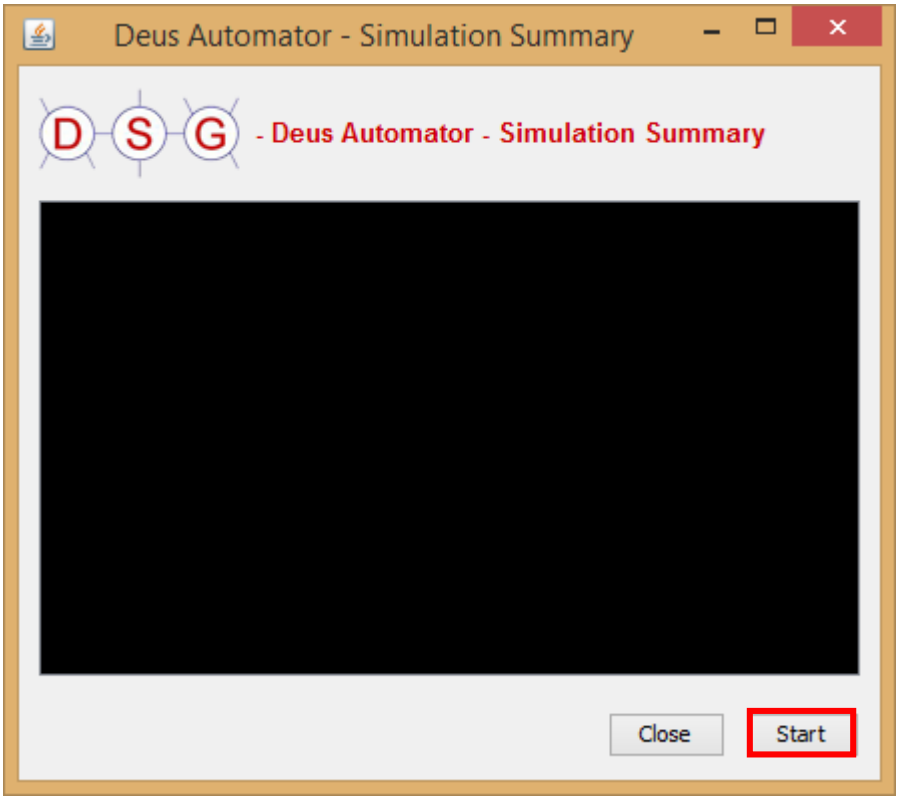


Figura 113: Ventana de elementos gráficos de la simulación

Después de que la simulación haya terminado, la ventana principal mostrará una barra verde si todo ha ido bien y una roja se ha habido algún error. En la figura 114 se muestra el resultado correcto de la simulación.

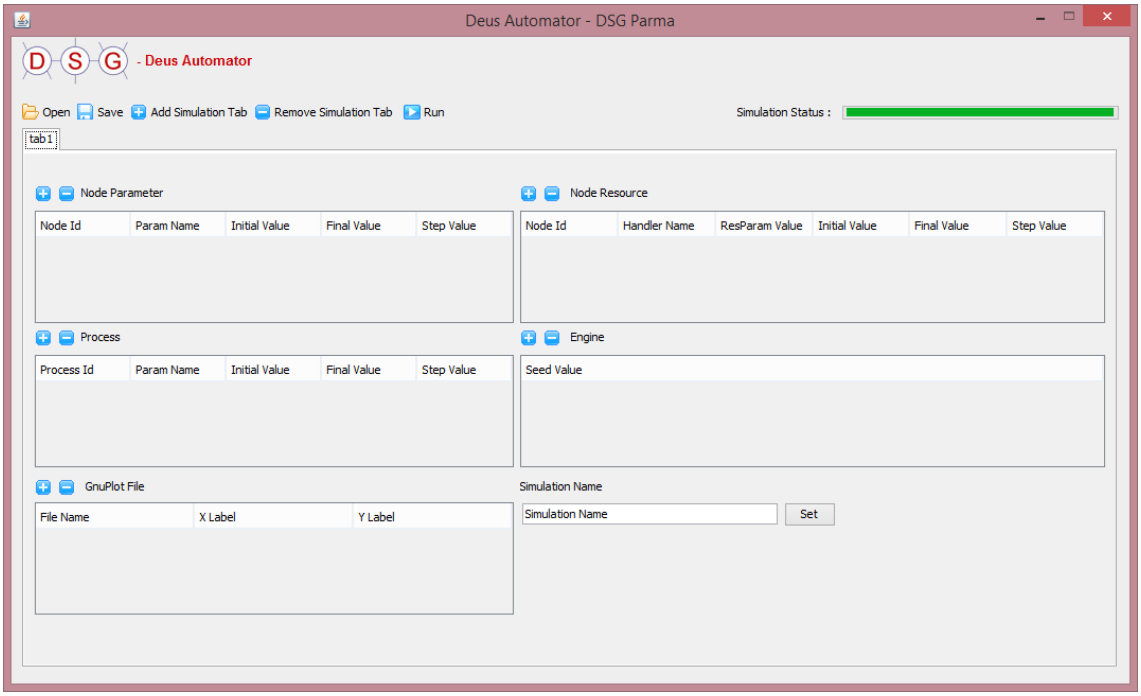


Figura 114: Resultado simulación

Esta forma de utilizar *DEUS* proporciona en principio poca información, por este motivo se han generado una serie de ficheros de logs con los resultados. Un ejemplo de estos ficheros va adjunto a este documento y se llama “resultados.zip”.

Desde ejecutable: Realizar la simulación desde un ejecutable solo cambia el comienzo de la simulación, una vez que se ha lanzado la ventana de *DEUS* el resto del proceso es igual a lo descrito anteriormente en esta sección.

Para ejecutar *DEUS* desde el ejecutable, es necesario la consola de comandos de *Linux* o *Windows* y navegar por los directorios hasta la ruta donde se encuentran los tres ficheros (el ejecutable, la definición de la simulación y el archivo de localización de ficheros temporales). Una vez que se ha alcanzado el directorio del ejecutable, se ejecuta el comando que aparece en la figura 115.

Además de los archivos indicados, es necesario que las carpetas de recursos estén en el mismo directorio. Dichas carpetas son: *icon, lib, log, res, results* y *schema*.

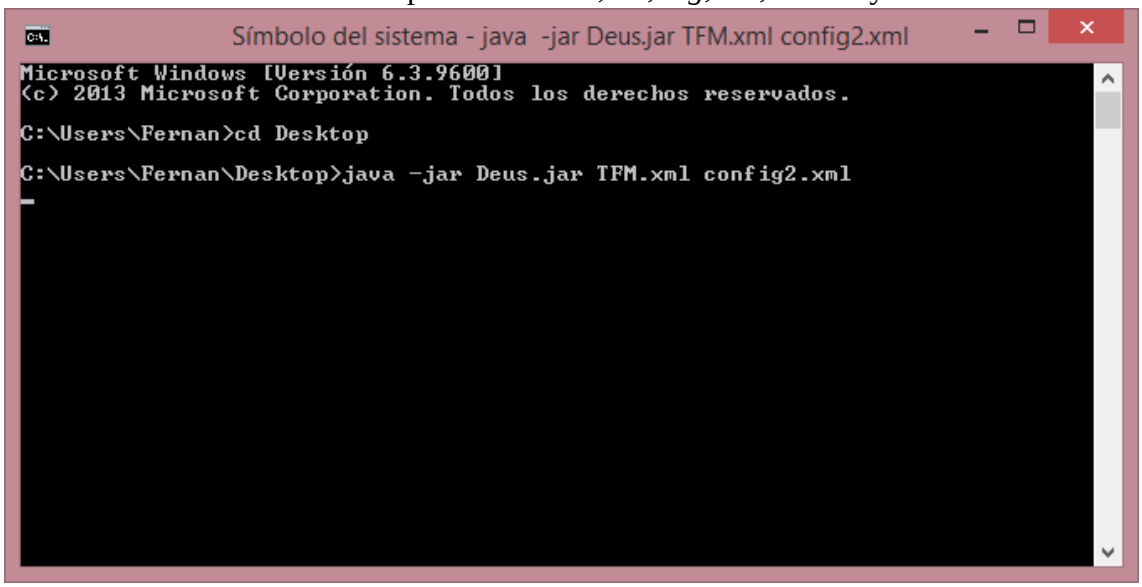


Figura 115: Ejecutar desde la consola de comandos

ANEXO II: Valores alternativos para realizar la simulación

El objetivo de este anexo es explicar donde habría que modificar el fichero *XML* de la simulación para modificar el comportamiento de la simulación. Debido a la naturaleza de la misma, podemos modificar dos tipos de valores, por un lado el número de repetición de los eventos, y por otro, los atributos de los nodos.

Repetición de eventos: En cada uno de los procesos que se definen en la simulación, hay un elemento dentro del elemento *aut:process* que define el valor inicial donde empezará a contar el iterador de la simulación, el valor final, y el incremento, se definen los mismos tipos de valores que un bucle *for* en Java. Lógicamente, si se incrementa el valor final y no se incrementa lo que aumenta el iterador, la simulación va a durar más tiempo. En la figura 116 se muestra un ejemplo de dos procesos, uno se ejecuta una vez y el otro 100, ambos están resaltados en la imagen.

```
<aut:process handler="it.unipr.ce.dsg.deus.impl.process.RectangularPulsePeriodicProcess"
    id="crearEventosSensorTemperatura">
    <aut:params>
        <aut:param name="period" value="1" />
        <aut:param name="startVtThreshold" value="0" />
        <aut:param name="stopVtThreshold" value="10000" />
    </aut:params>
    <aut:nodes>
        <aut:reference id="sensorTemperaturaH1" />
        <aut:reference id="sensorTemperaturaH2" />
        <aut:reference id="sensorTemperaturaH3" />
        <aut:reference id="sensorTemperaturaH4" />
        <aut:reference id="sensorTemperaturaH5" />
        <aut:reference id="sensorTemperaturaH6" />
    </aut:nodes>
    <aut:events>
        <aut:reference id="enviarTempServer" />
    </aut:events>
</aut:process>

<aut:process handler="it.unipr.ce.dsg.deus.impl.process.RectangularPulsePeriodicProcess"
    id="crearServer">
    <aut:params>
        <aut:param name="period" value="1" />
        <aut:param name="startVtThreshold" value="0" />
        <aut:param name="stopVtThreshold" value="1" />
    </aut:params>
    <aut:nodes>
        <aut:reference id="servidor" />
    </aut:nodes>
    <aut:events>
        <aut:reference id="birth2" />
    </aut:events>
</aut:process>
```

Figura 116: Parámetros del proceso

Cambiar atributos de los nodos: La otra alternativa para modificar la simulación, consiste en modificar los valores de los atributos de los nodos, esto puede ocasionar que dos nodos que representan el mismo objeto se comporten de manera diferente. Por ejemplo, se podría modificar los valores del sensor de temperatura de una habitación para que el límite inferior sea diferente, lo que ocasionaría que la caldera se activara con una temperatura que en otra habitación no habría hecho que se activase. En la figura 117 se muestra un ejemplo de que el sensor de temperatura de la habitación H1 tiene como valores límite 20º y 40º, mientras que el sensor de temperatura de la habitación H2 tiene como valores límite entre 10º y 30º.

```
<!-- SENSOR DE TEMPERATURA DE LA HABITACION 1 -->
<aut:node handler="it.unipr.ce.dsg.deus.nodos.temperatura.SensorTemperatura"
    id="sensorTemperaturaH1">
    <aut:params>
        <aut:param name="valorMinimo" value="20" />
        <aut:param name="valorMaximo" value="40" />
        <aut:param name="valorActual" value="30"/>
        <aut:param name="localizacion" value="H1"/>
    </aut:params>
</aut:node>
<!-- SENSOR DE TEMPERATURA DE LA HABITACION 2 -->
<aut:node handler="it.unipr.ce.dsg.deus.nodos.temperatura.SensorTemperatura"
    id="sensorTemperaturaH2">
    <aut:params>
        <aut:param name="valorMinimo" value="10" />
        <aut:param name="valorMaximo" value="30" />
        <aut:param name="valorActual" value="20"/>
        <aut:param name="localizacion" value="H2"/>
    </aut:params>
</aut:node>
```

Figura 117: Modificación de atributos de los nodos