UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

Titulo: Una arquitectura PUB-HUB para la Internet de las cosas.
Autores: Agustin Iannicelli
Director: Silvia Gordillo
Codirector:
Asesor profesional: Fernando Lyardet
Carrera: Licenciatura en Sistemas

Resumen

En el contexto de Internet de las Cosas, donde los objetos contarán incrementalmente con la capacidad de conectarse y compartir información, es necesario una arquitectura publish-to-hub que tiene como objetivo extender los beneficios de la comunicación publish-subscribe a dispositivos físicos que cuenten con recursos de computación y comunicación limitados tal cual sucede con los objetos que integran Internet de las Cosas.

El esquema donde uno de los componentes (el “Hub”) recibe todas las peticiones de los clientes que deseen publicar alguna información. Las entidades (dispositivos y aplicaciones de software) de la red local que estén interesadas en alguno de los temas pueden suscribirse al tema específico para recibir actualizaciones de información.

El trabajo aporta la extensión de una arquitectura pub-hub para permitir la publicación y suscripción de canales descriptos a partir de combinaciones de tags arbitrarios basados en lenguaje natural, el descubrimiento del hub en la red y la comunicación server push a objetos de la red.

Palabras Claves

Internet de las cosas, IoT, publish-subscribe, html5, webSockets, objetos inteligentes, ciudades inteligentes, canals de comunicacion, wordNet, descubrimiento de objetos en la Lan, Easy IoT, UDP, Comet, Server Push.

Trabajos Realizados

Se realizó un trabajo para poder demostrar fehacientemente la posibilidad de conectar objetos de una Lan interesados en un mismo tópico. Para ello se desarrollo una aplicación cliente (Android) y un servidor (Java), y con tecnologías tales como UDP, webSocket y wordNet se logro resolver con éxito el descubrimiento y la comunicación entre los objetos pertenecientes a Internet de las cosas.

Conclusiones

Se demostró que una variación del esquema publish-subscribe, cumplía con lo necesario para llevar a cabo la comunicación deseada entre objetos pertenecientes a Internet de las cosas. La variación corresponde a un esquema publish-to-hub que ayuda a dispositivos con capacidades limitadas a llevar a cabo la comunicación entre ellos. Este trabajo está íntimamente ligado a la importancia que se le está dando hoy en día a Internet de las cosas.

Trabajos Futuros

En un futuro se podría afrontar el problema de trabajar con aplicaciones web y cómo estas podrían encontrar el Hub que atiende las solicitudes de la Lan en la que se encuentra.
Otro punto a tener en cuenta es reducir las coincidencias de sinónimos no deseados de palabras utilizadas para asignar los tags en la creación de canales de comunicación.

Fecha de la presentación: 5 / 2015

Índice general

Objetivo y motivación de la tesis

1. Objetivo 7

2. Motivación 7

Capítulo 1: Introducción 9

1. Introducción: Internet de las cosas 11

1.1. Internet de las cosas (IoT) 12

1.2. Comienzos y perspectiva 14

1.3. Beneficios de IoT 15

1.4. Comunicación 16

1.5. Ciudades inteligentes? 17

Capítulo 2: Trabajos relacionados 19

1. Tecnologías y trabajos relacionados 19

1.1. Comunicación Máquina a Máquina (M2M) 19

1.2. IoT - M2M 21

1.2.1. Acceso a dispositivos remotos 22

1.2.2. Soluciones IoT benefician la gestión de servicios 22

1.2.3. Las diferencias en el proveedor de aplicaciones 23

1.2.4. Especificar la solución correcta según IoT o M2M 23

2. JWebSocket 24

2.1 Componentes de JWebSocket 24

2.2. Listeners, plugins y filters 25

2.2.1. Listeners 26

2.2.2. Plugins 28

2.2.3. Filters 30

2.3. Conectores, motores y servidores. 32

2.4. Token 32

2.4.1 Tokens desde el servidor al cliente 34

2.4.2 Tokens desde el cliente a servidor 34

2.5 Canales 35

2.5.1. Configuración de Canales 36

2.6. Cliente Javascript para un navegador web. 37

2.6.1. Hola mundo! 38

Capítulo 3: Tecnologías asociadas 42

3.1. Paquetes UDP 42

3.2. WebSockets 43

3.3. Publicación / Suscripción 45

3.4. Canales de comunicación 47

3.5. WordNet 47

Capítulo 4: Una nueva solución 51

4.1. El modelo de solución 51

4.2. Encontrar el hub en la red 51

4.2.1 SSDP (Protocolo Simple de Descubrimiento de Servicios) 52

4.2.2. UPnP 52

4.2.3. Multicast DNS (mDNS) 54

4.2.4. DIAL (DIscovery And Launch) 54

4.2.5. Protocolo de descubrimiento para WebSockets 55

4.3. El canal de comunicación 56

4.3.1. Canales definidos por palabras claves / tópicos 57

4.4. WebSockets para la comunicación a través de canales 57

Capítulo 5: Implementación EIot 61

5.1. Buscar el hub en la red. 61

5.2. El hub. 63

5.3. Descubrimiento de servicios, el hub. 65

5.4. Creación de canales, solicitud al hub. 66

5.5. Creación de canales, el hub. 67

5.5.1. Creación de canales, según palabras claves. 67

5.6. Envío de información al hub. 68

5.7. Comunicación a través de canales. 69

5.8. Recepción de información a través de canales. 70

5.9. Consola de control 70

5.10. Arquitectura 71

5.10.1. Servidor 71

5.10.2. Cliente 72

5.11. Comparación con JWebSockets 74

5.12. Casos de uso 75

5.12.1 Caso de éxito 75

5.12.2. Caso de fallo 79

Capítulo 6: Trabajos futuros 82

Conclusión 82

Bibliografía 83

Índice de figuras

Capítulo 1

Figura 1.1: Teléfonos inteligentes 9

Figura 1.2: Relojes inteligentes 10

Figura 1.3: Anteojos inteligentes 10

Figura 1.4: Internet de las cosas en las distintas áreas 14

Figura 1.5: Expectativas de crecimiento de objetos conectados 15

Figura 1.6: Santander, ciudades inteligente 18

Capítulo 2

Figura 2.1: Integración máquina máquina (M2M) 20

Figura 2.2: IoT vs M2M 21

Figura 2.3: Comparativa entre IoT y M2M 23

Figura 2.4: Estructura de la carpeta jWebSockets 25

Figura 2.5: Ejemplo de clase webSocketTokenListener. 27

Figura 2.6: Ejemplo para inclusión de un listener en el servidor de jWebSocket 28

Figura 2.7: Infraestructura de la cadena de listeners 28

Figura 2.8: Infraestructura de la cadena de plugins 29

Figura 2.9: Configuración para incluir un plugin 30

Figura 2.10: Configuración para incluir un filtro 31

Figura 2.11: Infraestructura completa de la cadena del servidor 31

Figura 2.12: Conectores, motores y servidores 32

Figura 2.13: Configuración para canales 37

Figura 2.14: Infraestructura del cliente jWebSocket 38

Figura 2.15: Instanciación del cliente de jWebSocket 39

Figura 2.16: Autenticación contra el servidor jWebSocket 39

Figura 2.17: Mensaje de broadcast 40

Figura 2.18: Procesamiento de mensajes entrantes 40

Figura 2.19: Cierre de la conexión en el cliente 40

Capítulo 3

Figura 3.1: Comunicación con webSockets 45

Figura 3.2: Esquema de publicación/suscripción simple 46

Capítulo 4

Figura 4.1.: Protocolo DIAL 55

Figura 4.2.: Protocolo de Descubrimiento 56

Figura 4.3.: Comunicación servidor-cliente por ajax 58

Figura 4.4.: Comunicación servidor-cliente por long polling 59

Figura 4.5.: Comunicación servidor-cliente por webSockets 60

Capítulo 5

Figura 5.1: Diagrama de componentes de la aplicación cliente 62

Figura 5.2: Diagrama de interacción de la aplicación cliente 63

Figura 5.3: Diagrama de interacción del hub 65

Figura 5.4: Diagrama de interacción del cliente: respuesta del hub 66

Figura 5.5: Diagrama de interacción del cliente 67

Figura 5.6: Creación de canales según palabras claves en el servidor 68

Figura 5.7: Diagrama de interacción: Envío de información por parte del cliente 69

Figura 5.8: Recepción y reenvío de información a través de un canal 70

Figura 5.9: Arquitectura de la aplicación servidor 72

Figura 5.10: Arquitectura de la aplicación cliente 73

Figura 5.11: Tabla comparativa entre JWebSockets y EIoT 75

Figura 5.12: Aplicación cliente desconectada. 76

Figura 5.13: Aplicación cliente conectada y enviando datos. 77

Figura 5.14: Aplicación dashboard conectada. 78

Figura 5.15: Aplicación cliente conectada. 78

Figura 5.16: Aplicación cliente conectada con el tag "core" 80

Figura 5.17: Aplicación cliente conectada con el tag “heart" 81

Figura 5.18: Aplicación dashboard inspeccionando los canales creados. 81

Objetivo y motivación

1. Objetivo

El objetivo de este trabajo de tesis es el diseño e implementación de un mecanismo para la comunicación flexible entre dispositivos físicos, aplicaciones y servicios de software. Para alcanzar dicho objetivo se presenta un esquema basado en una variación del esquema *publish-subscribe (*publicación-suscripción*)*. La variación propuesta se basa en un esquema que se denomina *publish-to-hub* o *pub-hub* (publicación al hub), y tiene como objetivo extender los beneficios de la comunicación publish-subscribe a dispositivos físicos que cuentan con recursos de computación y comunicación limitados. Esta característica es especialmente relevante en el contexto de Internet de las Cosas, donde los objetos y electrodomésticos que nos rodean a diario contarán incrementalmente con la capacidad de conectarse y compartir información, y por tanto es indispensable contar con una infraestructura que posibilite la interconectividad y compartición de datos.

2. Motivación

La intercomunicación entre dispositivos se ha basado tradicionalmente en enfoques a medida y verticales. Esto hace que la integración de dispositivos, sensores y servicios de software heterogéneos sea difícil de lograr por diversos motivos: primero, la variedad de mecanismos de comunicación es cada vez mayor, y los protocolos específicos hacen que sea más difícil compartir la información entre ellos, obligando el desarrollo de mecanismos de intercomunicación particulares para cada aplicación. Segundo, las capacidades de los dispositivos es muy variada en su capacidades de cómputo y comunicaciones. Un gran avance son los esquemas de comunicación *publish-subscribe* o “publicación-suscripción”, que posibilitan la comunicación entre diferentes entidades con bajo nivel de acoplamiento. Sin embargo, son poco aplicables a pequeños dispositivos como los que pueden esperarse en la internet de las cosas, ya que dichos mecanismos de comunicación, aunque flexibles, requieren en su amplia mayoría una capacidad de cómputo superior al disponible en estos casos.

Las tecnologías de comunicación de bajo acoplamiento como las que sustentan la web han demostrado con éxito que pueden ser escalables y flexibles para soportar a un gran número de entidades heterogéneas. Una variación son las llamadas arquitecturas de comunicación tipo *publish-to-hub* (o “*pub-hub*”), que representan un tipo particular de arquitecturas *publish-subscribe* [Aiten07]. Se trata de un esquema donde uno de los componentes (el denominado “Hub”) recibe todas las peticiones de los clientes que deseen publicar alguna información o "tema". Las entidades (dispositivos y aplicaciones de software) de la red local que estén interesadas en alguno de los temas pueden registrar su interés o "suscribirse" al tema específico para recibir actualizaciones de información. A su vez, el Hub notifica a todas las partes interesadas cuando nueva información ha sido publicada en algún tema. En este esquema, los dispositivos con baja capacidad de cómputo sólo deben ser capaces de suscribirse y recibir notificaciones, lo cual facilita su accesibilidad y implementación.

Este trabajo aporta la extensión de una arquitectura pub-hub para permitir la publicación y suscripción de canales descriptos a partir de combinaciones de tags arbitrarios basados en lenguaje natural. Adicionalmente, se introduce la utilización del protocolo HTTP [RFC2616 1999] para las suscripciones en combinación con el protocolo Websockets [Fette 2011] para el broadcast de eventos. La utilización de este protocolo permite la suscripción y notificación a aplicaciones web en una forma sencilla, transparente y uniforme, permitiendo una integración dinámica más sencilla. Esta configuración permite además que los dispositivos con recursos muy limitados (por ejemplo, Arduino [Arduino]) puedan compartir su información con todo el mundo, algo que en un esquema pub-sub distribuido tradicional no sería posible.

Combinando esta arquitectura pub-hub con las tecnologías emergentes de HTML5, podremos lograr la integración de dispositivos, servicios de software y combinarlas (mashups) en una típica aplicación web 2.0.

Capítulo 1: Introducción

Las similitudes entre objetos de uso cotidiano como un teléfono, una heladera, un par de anteojos y un reloj no parecen ser demasiadas...sin embargo estos objetos que se suelen usar diariamente, han pasado a ser parte de Internet de las cosas (IoT) Estos dispositivos ya no solo cumplen con su función original, sino que gracias a que poseen conexión a internet pasan a ser objetos pertenecientes a la red de objetos interconectados, estos objetos pueden enviar o recibir información de la red, de otros objetos o bien enviar información al resto de los objetos conectados.

Estos dispositivos tienen la habilidad de conectarse y desempeñar tareas más complejas. Hoy un celular es también un gps que puede guiar el recorrido de un vehículo, una heladera puede avisarnos al celular que un alimento está por vencer, un par de anteojos [Glass 2014] es también una cámara de fotos que puede ser enviada a una red social.

Figura 1.1: Teléfonos inteligentes

Los smart phones ya no solo permiten llamar y enviar sms ahora se puede enviar mails, usarlo como gps y programar nuestro calendario.

Figura 1.2: Relojes inteligentes

Los relojes inteligentes que no solo dan la hora, se comunican con el celular y permiten realizar múltiples tareas sin acceder al teléfono, también o funcionan como un gps.

Figura 1.3: Anteojos inteligentes

Los anteojos inteligentes creados por google nos permiten acceso al mundo digital constantemente.

Esta posibilidad de conectar objetos con objetos sin la necesidad de la presencia de humanos tiene un sin fin de utilidades, pensemos por ejemplo en los autos inteligentes este podría sugerirnos la mejor ruta, accediendo no sólo a los mapas del GPS, sino también a la información del tráfico. Además, mientras conducimos podremos ir descargando películas para así aprovechar los aburridos trayectos del trabajo a casa y de casa al trabajo.

En Santander (España), se ha instalado una red de 25.000 sensores que monitorizan el tráfico, el transporte, la iluminación, el ruido, la contaminación y la calidad del agua. La ciudad ha reducido un 80 por ciento la congestión del tráfico, además de desplegar aplicaciones abiertas que mejoren los servicios y facilitan la participación ciudadana [Santander ciudad inteligente 2013].

1. Introducción: Internet de las cosas

Teléfonos, anteojos, relojes y más, todos estos objetos de la vida cotidiana se han vuelto “inteligentes” todos conectados a internet, capaces de captar información del medio y enviarla y/o recibirla por la red….si la tendencia continúa así, la red estará plagada de objetos conectados a la red. Es la compañía investigadora del mercado de IT, Gartner, quien , analizó la potencial evolución de lo que se ha dado en llamar “*Internet de las Cosas*” [Introduction IoT 2013] . Según sus más recientes proyecciones, en pocos años será de miles de millones la cantidad de conexiones de sensores y dispositivos que se conectan a Internet para compartir información o nutrirse de ella [Gartner 2013].

Estos objetos son objetos cotidianos que se usan día a día y ya que no son grandes computadoras con grandes procesadores y mucho poder de cálculo y almacenamiento, es necesario minimizar todas las operaciones que se puedan, tanto de comunicación como de conocimiento con el resto de los objetos conectados a la misma red.

Para lograr la comunicación efectiva entre los objetos de la red se necesita un hub que coordine los objetos [PubSubHubHub 2014], los mensajes y los canales, ya que la poca capacidad de cómputo de los objetos hace imposible que estos puedan administrar los objetos a los que interesa conectarse y/o mandar información, quien puede ayudar a resolver esto es un hub que si es una computadora u objetos con mayor poder de cómputo y de almacenamiento que intercede para coordinar la red. El problema que tienen los objetos es a la hora de encontrar el hub, ya que la direccion no esta fija y son objetos que van cambiando su posición constantemente y por ende cambiando la red a la cual están conectados. Para poder encontrar el hub, se necesita un mecanismo que nos permita llegar a todos los dispositivos de la red con un mismo mensaje, un mensaje de pedido de dirección por parte del hub, la forma de hacer esto es con un paquete UDP [IETF RFC768 2013].

Para que los objetos puedan recibir y enviar información se necesita de alguna tecnología que permita comunicaciones bidireccionales y que cualquiera de los objetos conectados puede iniciar la comunicación, es necesario informar la dirección ip de cada objeto y poder recibir información sin estar haciendo operaciones que sobrecarguen la capacidad de los pequeños dispositivos que tendrán los objetos, una tecnología como Websocket [W3 Websockets 2014] soluciona esto ya que implementa una comunicación bidireccional donde cualquiera de los objetos conectados puede iniciar una comunicación con el otro sin estar utilizando técnicas como polling o long polling.

Por el bajo poder de cómputo de los objetos no es posible que lleven control del resto de los objetos de la red con los cuales tiene interés en conectarse para recibir o enviar información, para solucionar este problema cada red cuenta con la ayuda de un hub que será responsable de administrar los objetos y mensajes de la red, los objetos no se comunicaran directamente sino que lo

harán a través del hub, quien llevará registro de los objetos, sus tópicos de interés y los canales de comunicación que se usarán para comunicar los mensajes entre los objetos. Esta metodología de conectar objetos a través de un hub intermedio es una variación de Publicación/Suscripción [Publish-subscribe 2014].

En la red los objetos estarán enviando miles de mensajes todo el tiempo y estos mensajes no son de interés para todos los objetos, es decir no todos los objetos están interesados en todos los mensajes, por el contrario cada objeto tendrá un tópico en el cual estará interesado y en el resto no tendrá interés. Para evitar que todos los mensajes lleguen a todos los objetos y sobrecargar la red con mensajes a destinatarios a los que no les interesa el mensaje, el envío de mensajes se organiza a través de canales de comunicación, estos canales agrupan a los objetos que requieren cierto tipo de información sirven como tipificadores de mensajes. Si por ejemplo un objeto va a enviar un mensaje con un dato censado lo enviará al hub para que lo envíe por el canal correspondiente y no a todos los objetos de la red.

Para poder tipificar los canales de comunicación y hacerlo de la forma más intuitiva posible se utilizan palabras en inglés del lenguaje natural, esto le da un significado semántico al canal, pero como el lenguaje natural tiene varios sinónimos para hacer referencia a la misma cosa, es que se utiliza la base de datos de WordNet [Wordnet 2014], esta base de datos posee las palabras del idioma inglés asociados a sus sinónimos. Con esto se logra que canales que deseen enviar o recibir información correspondiente a un tópico en particular no necesariamente tengan que usar el mismo tópico para encontrar el canal, sino que bastará con que usen un sinónimo de dicho tópico, ya que wordNet se encarga de interpretar los sinónimos para utilizar el mismo canal, ya que para palabras sinónimas el significado semántico del canal es el mismo y debería utilizarse el mismo canal.

1.1. Internet de las cosas (IoT)

El término Internet de las cosas (Internet of Things - IoT) fue propuesto por Kevin Ashton en 1999 en una presentación que defendía la idea de que las etiquetas RFID [RFID 2014] asociadas a objetos físicos les conferían una identidad bajo la cual podían generar datos sobre ellos mismos o sobre lo que perciben y publicarlos en internet [Introduction to IoT november 2013] . La principal novedad en esta visión es que hasta entonces han sido fundamentalmente las personas quienes generan la información accesible en la red (textos de noticias, artículos o comentarios) o sistemas software automáticos (información de vuelos o de la bolsa), pero nunca lo habían sido objetos físicos reales.

En esencia el Internet de las Cosas se basa en sensores, en redes de comunicaciones y en una inteligencia que maneja todo el proceso y los datos que se generan. Los sensores son los sentidos del sistema y, para que puedan ser empleados de forma masiva, deben tener bajo consumo y costo, un tamaño reducido y una gran flexibilidad para su uso en todo tipo de circunstancias. La evolución de Internet también precisa de potentes y seguras redes de comunicación inalámbrica M2M (máquina a máquina) [Machine-To-Machine 2010], que hagan posible la incorporación a las redes y a los sistemas de objetos totalmente fuera de ellos hasta hace poco. Finalmente es necesario aplicar inteligencia (“smart”) a los sistemas y a los objetos, aprovechando los datos recogidos por los sensores, para procesarlos y convertirlos en información útil y en acciones [Atzori 2010].

Bajo el concepto de «Internet de las cosas» se encuentra la idea primordial de que los

objetos que nos rodean, sean electrodomésticos, vehículos, ropa, latas de gaseosa o el propio banco de la calle se convierten en ciudadanos de primera clase en internet, productores y consumidores de información, generada por ellos mismos, por las personas o por otros sistemas.

Las computadoras y, por tanto, Internet dependen casi exclusivamente de los seres humanos para obtener información. Casi la totalidad de los aproximadamente 50 petabytes (un petabyte es 1024 terabytes) de datos disponibles en Internet fueron capturados y creado por los seres humanos. El problema es que la gente tiene tiempo, atención y precisión limitada, esto significa que no son muy buenos en la captura de datos acerca de las cosas del mundo real. Y esto es una gran problema. La IoT tiene el potencial de cambiar el mundo, tal como lo hizo Internet. Tal vez aún más [Ashton 2009].

IoT es la red de objetos físicos cotidianos que están conectados a internet y son capaces de comunicarse y generar información para otros objetos. Éstos contienen tecnología embebida para interactuar con estados internos o con el medio que los rodea. Cuando estos objetos sensan y comunican información cambian como y cuando las decisiones son tomadas.

Es importante que un objeto puede representarse digitalmente, convirtiéndose en algo más grande que el objeto en sí mismo. Ya que no solo se relaciona consigo mismo sino que ahora se conecta con objetos cercanos. Cuando varios objetos actúan juntos, esto es conocido como ambiente inteligente.

IoT no es algo extraño, por el contrario, el fenómeno es el natural desarrollo de la Internet existente. Hoy en día varios objetos están comunicados unos con otros, podemos encontrar heladeras conectadas a internet, relojes conectados al celular, anteojos google glass y cada vez veremos más y más objetos interconectados.

Según Patrick Fältström lo que estamos presenciando ahora es el retorno al diseño original de Internet, conectar objetos con objetos, donde la interacción humana ya no es indispensable para realizar esta coneccion y donde los objetos se nutren de otros para el propio funcionamiento [Raunio 2009].

Aunque el concepto de “objeto inteligente” ha estado siempre presente desde la antigüedad, no es casualidad que sea ahora cuando realmente hemos empezado a ver su materialización, y ello se debe fundamentalmente a cuatro factores: decrementos de tamaño y precio en electrónica, cobertura de comunicación global y estilo de vida digital de la población [Smart Objects 2014].

Los decrementos de tamaño y precio en los componentes electrónicos necesarios para conectar cualquier tipo de producto a internet y dotarlo de una nueva proposición de valor permiten que el sobrecosto de dicho proceso no sea excesivo y que está electrónica pueda camuflarse dentro del propio objeto sin que el usuario lo perciba como más voluminoso. Populares marcas de ropa comercializan artículos que permiten monitorizar el rendimiento al correr mediante un pequeño dispositivo electrónico situado bajo la suela del calzado deportivo del fabricante para luego visualizar las medidas en un terminal móvil. Sin los avances en miniaturización y reducción de costos el producto no podría haber sido tan popular [Gartner 12-2013].

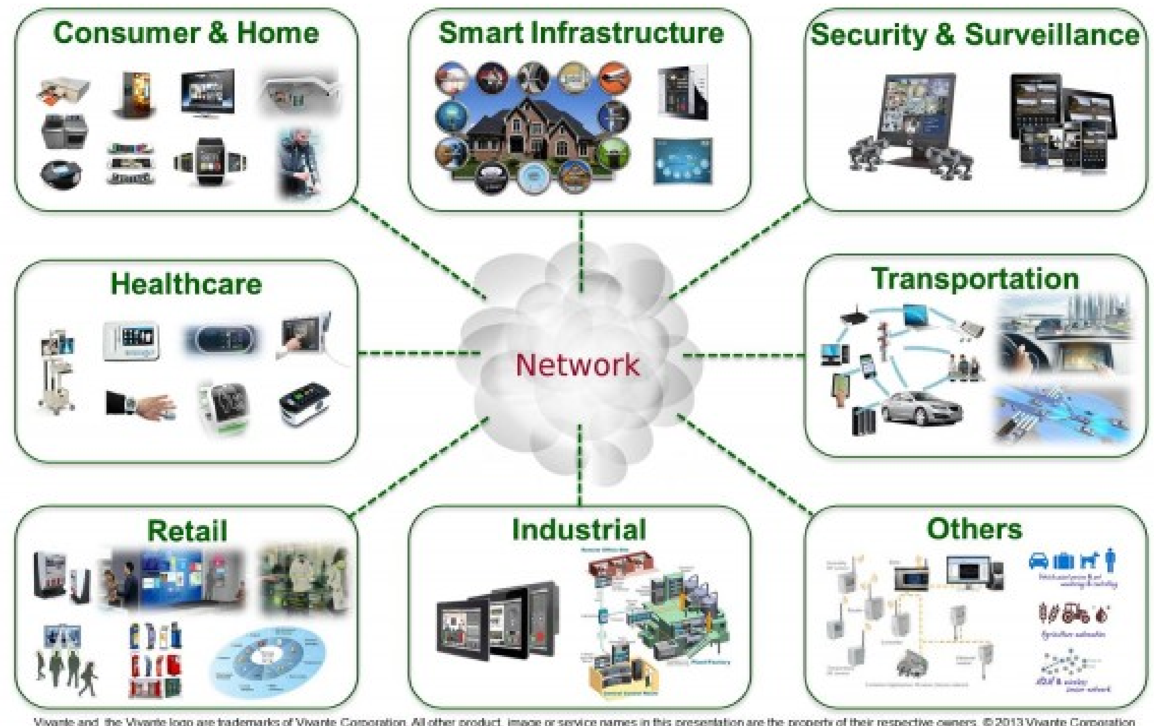

Figura 1.4: Internet de las cosas en las distintas en distintas áreas.
Internet de las cosas se introducirá en todos los ámbitos de la vida cotidiana.

1.2. Comienzos y perspectiva

Los primeros pasos fueron dados a través del experimento de un grupo de científicos realizado en 1991 con una cafetera convencional y su deseo de poder saber a todo momento cuánto café quedaba en ella. Lo que realizaron fue la primer implementación de la webcam en el año 1991. Dicha cámara fue conectada por el grupo apuntando a la cafetera, que desde ese momento pasó a tener su propia identidad en Internet y se convirtió en un objeto interconectado, ya que a través de la cámara y desde cualquier otro dispositivo con conectividad a internet se podía acceder a ella.

Adam Dunkels es un científico de SICS que ha trabajado con la tecnologia detras de “Internet de las cosas” cerca de 10 años. En su seminario cuenta que Ericsson prevé que más de 50 billones de “cosas” estarán interconectadas vía Internet en el año 2020 y que Cisco prevé que Internet puede crecer hasta mil veces su tamaño actual y éste incremento se verá reflejado en objetos cotidianos que utilizando pequeños chips podrán comunicarse unos con otros.

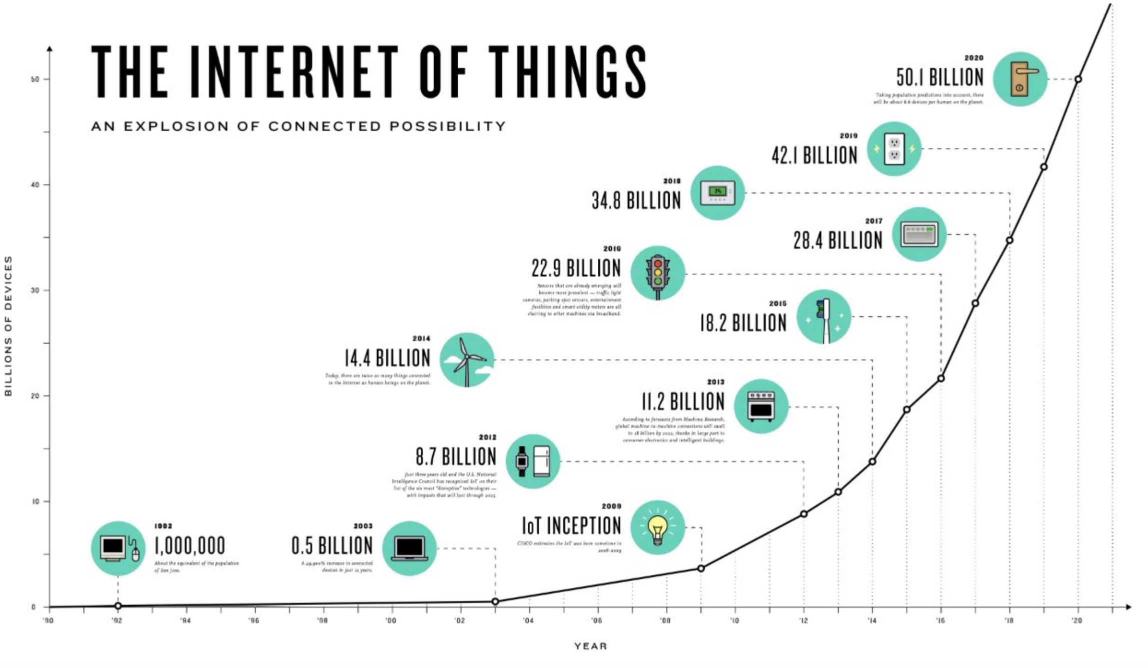

Figura 1.5: Expectativas de crecimiento de objetos conectados

Gráfico que muestra la expectativa de crecimiento de la cantidad de objetos conectados a la red dando lugar a la internet de las cosas.

1.3. Beneficios de IoT

La IoT está compuesta por objetos inteligentes que no dependen de los seres humanos para enviar y/o recibir información, estos objetos son capaces de transmitir información sobre su estado interno o sobre el medio en el que se encuentran. Que un objeto pueda enviar este tipo de información a Internet proporciona numerosas ventajas.

- *Un aire acondicionado podría enviar información sobre si necesita un cambio de filtro porque está ya sucio*
- *Un producto en stock podría enviar información sobre la temperatura y grado de humedad en la que está, y así ver si se está conservando de forma adecuada*
- *Un producto podría enviar información sobre su ubicación, de manera que en todo momento podríamos saber dónde está.*
- *Un fabricante podría recibir información acerca de la frecuencia de uso de sus productos (si el usuario lo permite, claro) y así poder optimizar el diseño de los mismos*

Los objetos inteligentes de IoT pueden recibir información vía Internet de otros objetos conectados y utilizar esa información para tomar decisiones.

- *Activación a distancia, por ejemplo de la luz de una habitación, en cuanto nos estemos acercando y apagarla en cuanto no sea necesaria, o Conectar el alumbrado público en*

cuanto haya coches o presencia humana

- *Activación de un Producto cuando el Producto complementario esté ya preparado.*
- *Entrega de productos (por ejemplo, en un drive in) de forma automática y Just in Time cuando el coche esté delante del mostrador*
- *Identificarnos (y por tanto ofrecernos, en un aparador de cristal digital, lo que sea más adecuado para nosotros*

En cualquier sistema de regulación automática hay un input, una gestión, y un output. Los objetos dejan de ser entidades inertes para convertirse en entidades que pueden captar cambios en el entorno (químicos, físicos, señales electromagnéticas, etc) , computarlos y reaccionar a ellos. Se convierten en **'smart products'** , más o menos complejos en función de lo que esperamos de ellos.

1.4. Comunicación

IP para objetos inteligentes

Para dar soporte a la gran cantidad de nuevas aplicaciones para los objetos inteligentes, la tecnología de red subyacente debe ser inherentemente escalable, interoperable, y tienen una base sólida de normalización para soportar futuras innovaciones.

IP ha demostrado ser una comunicación de larga vida, estable y altamente escalable tecnología que soporta tanto una amplia gama de aplicaciones, de dispositivos y de tecnologías de comunicación subyacentes. La arquitectura en capas de IP proporciona un alto nivel de flexibilidad e innovación. IP ya soporta una gran cantidad de aplicaciones, tales como el correo electrónico, la World Wide Web, la telefonía por Internet, streaming de vídeo y herramientas de colaboración. En los últimos 20 años, la propiedad intelectual ha evolucionado para apoyar nuevos mecanismos de alta disponibilidad, mejorada seguridad, soporte de calidad de servicio (QoS), transporte en tiempo real y virtual Redes privadas (VPN).

IP ofrece estandarización, ligereza e independencia de la plataforma para el acceso a los objetos inteligentes y otros dispositivos integrados de la red. El uso de IP hace a los dispositivos accesibles desde cualquier lugar y de cualquier forma; PCs, teléfonos celulares, PDAs, así como servidores de bases de datos y otros equipos, como un sensor de temperatura o una bombilla. IP se ejecuta virtualmente sobre cualquier tecnología de comunicación subyacente, que van de alta velocidad de los enlaces Ethernet cableadas a radios 802.15.4 de baja potencia y equipos 802.11(WiFi). Para la comunicación de larga distancia, datos IP se transporta fácilmente a través de canales cifrados sobre Internet global [Dunkel 2008].

Aún hoy hay muy pocos productos conectados a Internet en comparación a las proyecciones próximas. La innovación será en los próximos años cuando se produzca un crecimiento masivo de los productos conectados a internet y entre sí. Y estos productos no tendrán que ser sofisticados, sino que en la mayoría de los casos serán productos cotidianos que usamos día a día, como puede ser una heladera, un aire acondicionado o un chip. Cuando estos objetos se quieren conectar deben superar los inconvenientes a los que se enfrenta cualquier sistema que desea transmitir información. Estos objetos son objetos cotidianos con muy poco poder de cómputo y la comunicación no debe consumir más de lo indispensable y deben poder satisfacer las siguientes operaciones:

- introducir un nuevo dispositivo a la red de objetos conectados.
- comunicación de objetos entre sí con grado de procesamiento mínimo en cada uno.
- recibir en cada objeto la información necesaria.

1.5. Ciudades inteligentes?

Una ciudad inteligente [Smart Cities 2015] sería un repositorio de conocimiento sobre la urbe, donde los objetos físicos que la habitan, como mobiliario urbano, sensores de polución, semáforos, camiones de recolección de residuos o sistemas de riego de jardines podrían crear o actualizar los contenidos para reflejar los cambios que perciben a lo largo del tiempo. Por ejemplo, la página relativa al nivel de polución diaria sería actualizada constantemente por los sensores de polución o de partículas en suspensión, según la hora del día y zona; la página de información meteorológica de la ciudad sería actualizada constantemente por los sensores de temperatura, viento, luz y lluvia desplegados por los diversos parques y jardines. Ambas páginas podrían a su vez ser consultadas por los sistemas de control de tráfico para determinar la correlación entre un aumento de los niveles de polución de una zona, el tráfico de la misma recogido por los sensores situados en el asfalto y la información meteorológica relevante, y en base a ello tomar decisiones de planificación de tráfico que mejoren la calidad de vida de los ciudadanos.

El concepto de la ciudad inteligente no es muy distinto al de la Wikipedia. La única diferencia radica en los productores y consumidores de la información: ahora son los objetos físicos conectados a internet quienes crean un repositorio de conocimiento sobre un determinado entorno para enriquecerse unos a otros y volverse más inteligentes desde un punto de vista colectivo.

Ya existen ejemplos de metrópolis de este tipo, donde las redes de sensores inteligentes se han desplegado de manera experimental para crear ciudades conscientes de sí mismas, que sienten y se adaptan, en una tendencia denominada smart cities. Algunos de los casos más paradigmáticos son Smart Santander (España), Amsterdam Smart City (Países Bajos) y Songdo IBD (Corea del Sur). Detrás de muchas de estas iniciativas se encuentran grandes corporaciones de software y equipamiento que apuestan estratégicamente por los servicios de valor añadido que una ciudad conectada puede prestar a sus ciudadanos.

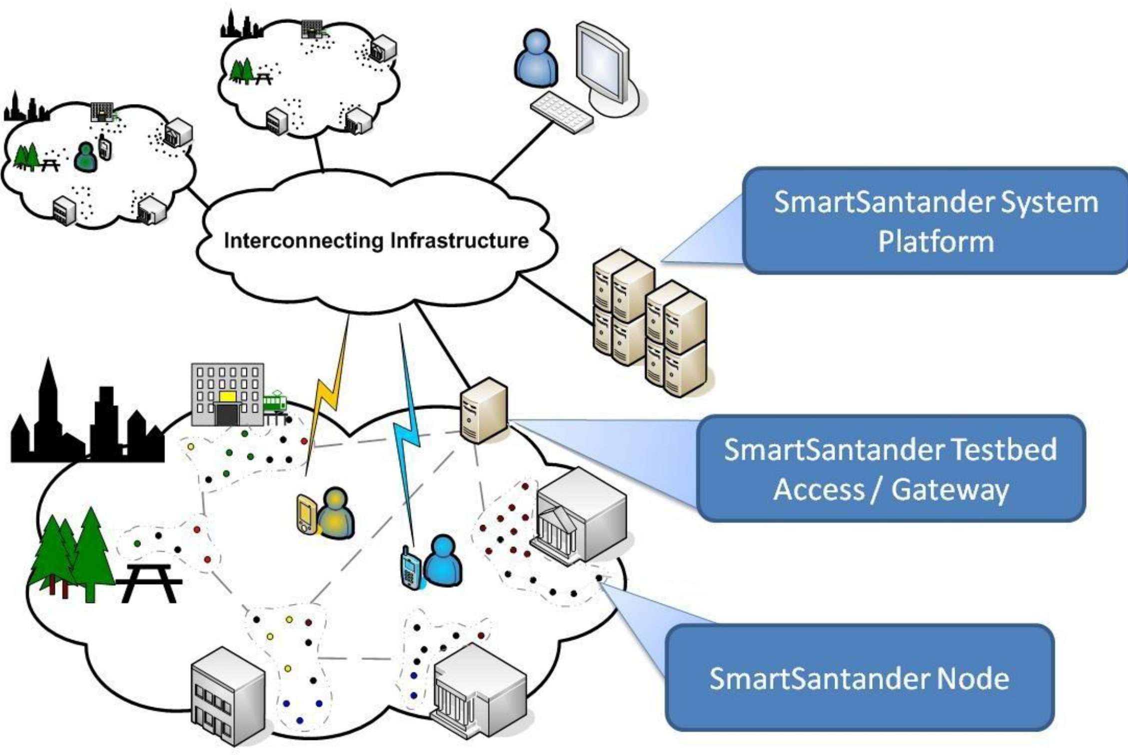

Figura 1.6: Santander, ciudades inteligente

Las ciudades inteligentes sacan todo el provecho necesario a internet de las cosas y el flujo de comunicación continuo entre los objetos.

Capítulo 2: Trabajos relacionados

En 1885 fue un termostato el primer objeto en la tecnología de comunicar máquinas con máquinas; ya en ese entonces se empezó a ver la necesidad de que las máquinas se comuniquen unas con otras, enviando información y accionando acorde al dato recibido.

Pero comunicar una máquina con otra no es suficiente, la necesidad pasó a ser más compleja ya no basta conectar una máquina con otra sino que se necesita conectar todos los objetos de una red. Los datos generados por un objeto pueden ser requisito para muchos de los objetos conectados, es por eso que una conexión punto a punto, máquina a máquina no es suficiente.

También es necesario que los objetos tengan la capacidad de adaptarse y puedan cambiar de red y seguir ofreciendo sus servicios. Hoy muchos de los objetos conectados son objetos pequeños y móviles que cambian de subred constantemente.

1. Tecnologías y trabajos relacionados

Con la escalada de dispositivos conectados a la red pronosticada en más de 50 billones para el año 2020 nace la necesidad de sistemas que aprovechen la información que estos objetos pueden generar.

La necesidad de compartir información siempre está presente sobre todo en la era de la informática donde todo está dependiendo de la información que se recupera, minar la red con miles de millones de objetos genera la posibilidad de obtener y procesar infinita cantidad de información, pero para esto se necesita poder organizar estos dispositivos de manera que puedan comunicarse con el resto de los objetos de la red de manera de minimizar la administración de estos objetos.

Todo esto no es nuevo pero cada vez se estan agregando mas y mas objetos a la red, objetos de la vida cotidiana pasan a ser objetos conectados, objetos productores y consumidores de información, celulares, anteojos, impresoras, etc. manipular tal cantidad de datos y objetos necesita otro enfoque desde el punto de vista de la comunicación de estos dispositivos, enfoques que ya fueron iniciados por la comunicación M2M (Máquina a máquina), que siguió con la Internet de las Cosas y que intentamos ampliar ahora con un modelo que se adapte a las nuevos desafíos tales como la comunicación bidireccional, la ausencia del hombre para interceder en las comunicaciones y conexiones y el bajo procesamiento de los dispositivos cotidianos.

1.1. Comunicación Máquina a Máquina (M2M)

La comunicación máquina a máquina (M2M) se refiere a las tecnologías que permiten tanto a los sistemas inalámbricos y con cable comunicarse con otros dispositivos del mismo tipo. M2M [M2M 2014] es un término muy amplio, ya que no señala la creación de redes específicas, la información inalámbrica o por cable y tecnología de las comunicaciones. Este amplio término se utiliza sobre todo por los ejecutivos de negocios. M2M se considera una parte integral de la Internet de las Cosas (IoT) y trae varios beneficios a la industria y los negocios, en general, ya que tiene una amplia gama de aplicaciones tales como la automatización industrial, la logística, *Smart Grid, Smart*

Cities, salud, defensa, etc. principalmente para el seguimiento, con fines de control.

Con el fin de apoyar los nuevos desarrollos y la adopción mundial de Internet de las cosas, así como el crecimiento continuo de la tecnología M2M y sus aplicaciones a gran escala en el futuro, una adopción global y el despliegue del Protocolo de Internet versión 6 (IPv6) son necesarios porque todos los sensores e identificadores de lectura mecánica necesarios para hacer de Internet de las Cosas en realidad tendrá que usar IPv6 para acomodar el espacio de direcciones extremadamente grande requerida. Incluso si la oferta actual de direcciones IPv4 no fuera a agotarse pronto, el tamaño de la misma IPv4 no es lo suficientemente grande como para soportar el futuro requerimiento de la IoT.

M2M puede incluir el caso de instrumentación industrial - que comprende un dispositivo (tal como un sensor o medidor) para capturar un evento (tales como la temperatura, el nivel de inventario, etc.) que se transmite a través de una red a un aplicación que traduce el evento capturado en información significativa (por ejemplo, que artículos necesitan ser abastecidos).

Sin embargo, la comunicación M2M moderna se ha expandido más allá de una conexión de uno a uno y a cambiado a un sistema de redes que transmite los datos a los aparatos personales. La expansión de las redes IP en todo el mundo ha hecho que sea mucho más fácil para que la comunicación M2M se lleve a cabo y se ha disminuido la cantidad de energía y tiempo necesario para que la información se transmita entre las máquinas.

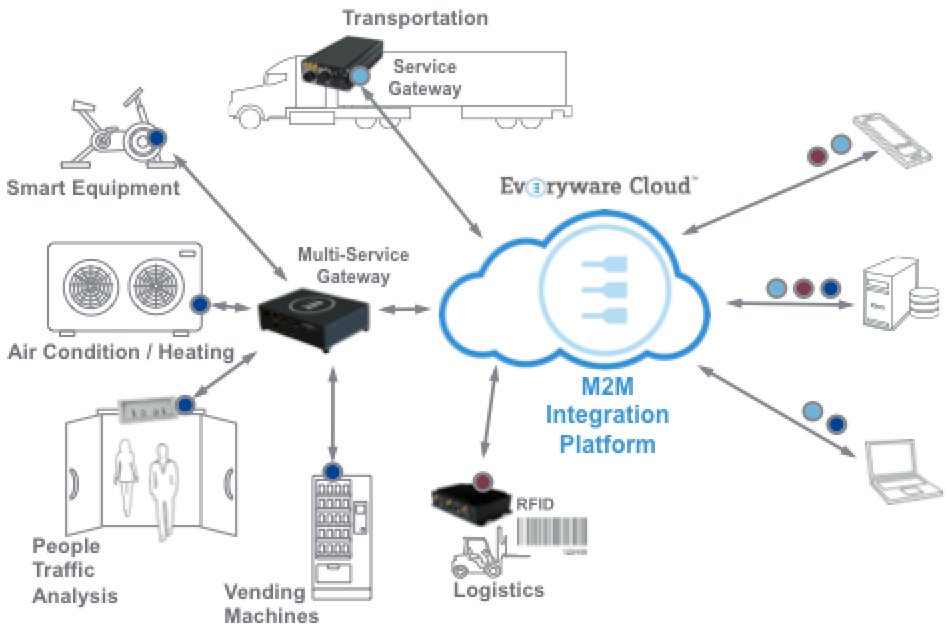

Figura 2.1: Integración máquina máquina (M2M)

Representación de integración máquina máquina (M2M) según el *EverywareCloud* de Eurotech que simplifica dispositivos y gestión de datos mediante la conexión de dispositivos distribuidos en servicios en la nube seguras y fiables

1.2. IoT - M2M

Las soluciones M2M tradicionales normalmente se basan en comunicaciones de punto a punto utilizando módulos de hardware embebido y, o bien las redes celulares o de línea fija. Por el contrario, las soluciones de IoT se basan en las redes basadas en IP para interconectar los datos del dispositivo a una nube o plataforma middleware.

El mercado M2M ha sufrido incapacidad para darse cuenta de su potencial de crecimiento previsto - y las razones de ese fracaso - proporcionan diciendo indicadores de las verdaderas diferencias entre el IoT vs M2M [M2M-IoT 2014]. Mientras que las soluciones M2M ofrecen acceso remoto a los datos de la máquina, estos datos son objeto tradicionalmente en soluciones puntuales en las aplicaciones de gestión de servicios. Rara vez, o nunca, son los datos integrados con las aplicaciones empresariales para ayudar a mejorar el rendimiento general del negocio. Integración de dispositivos y sensores de datos con grandes volúmenes de datos, análisis y otras aplicaciones de la empresa es un concepto central detrás de la emergente IoT. Esta integración es clave para lograr numerosos beneficios en toda la empresa de fabricación y, en última instancia, el crecimiento en el mercado.

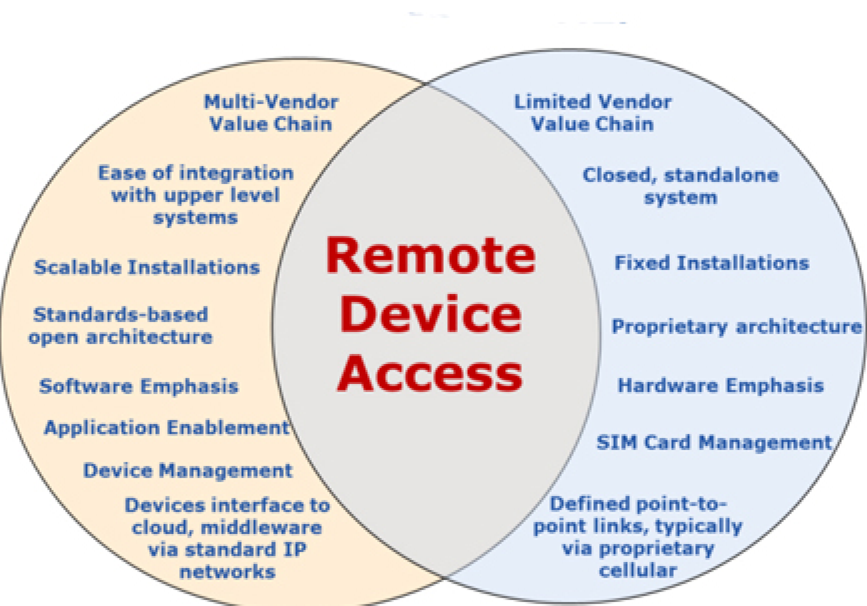

Figura 2.2: IoT vs M2M

Si bien las dos tecnologías proveen acceso remoto a dispositivos, el gráfico muestra las diferencias entre las dos tecnologías.

1.2.1. Acceso a dispositivos remotos

El acceso a dispositivos remotos, máquinas, objetos y otras entidades ofrece una propuesta de valor primordial tanto para las soluciones M2M e IoT. Las aplicaciones M2M se componen típicamente de módulos de hardware incorporados en una máquina del lado del cliente que se comunican a través de redes celulares o de telefonía fija a una aplicación de software dedicada, a menudo en la operación de servicio del proveedor. Esta capacidad permite que el proveedor del dispositivo reduzca sus costos de gestión de servicios a través de diagnósticos a distancia, resolución de problemas, actualizaciones y otras capacidades remotas que disminuyen la necesidad de desplegar el personal de servicio de campo.

En las soluciones industriales de IoT, el "qué, cómo y por qué" de acceso al dispositivo remoto implica pinceladas mucho más amplias. IoT acomoda no sólo los mismos dispositivos como las aplicaciones M2M, sino también sensores de baja potencia y pasivos, así como dispositivos de bajo costo que pueden no ser capaces de justificar un módulo de hardware M2M dedicado. Dispositivos IoT se comunican a través de redes IP basadas en estándares y sus datos se incorporan a las aplicaciones de la empresa para permitir no sólo la mejora del servicio, sino también la mejora operativa y nuevos modelos de negocio como "producto como servicio".

La capacidad de las aplicaciones en toda la empresa para acceder a los datos del dispositivo para permitir mejoras en el rendimiento, la innovación empresarial u otras posibilidades distingue claramente el potencial de la IoT en comparación con la de M2M. Esta entrega de datos basada en la IoT es por lo general a una nube, lo que permite el acceso de cualquier aplicación empresarial. Por el contrario, las soluciones M2M normalmente emplean la comunicación directa de punto a punto. La arquitectura basada en la nube también hace que IoT sea inherentemente más escalable, lo que elimina la necesidad de conexiones cableadas adicionales y las instalaciones de la tarjeta SIM. Esta es una razón por M2M se refiere a menudo como "la plomería", mientras que IoT es vista como un habilitador universal.

1.2.2. Soluciones IoT benefician la gestión de servicios

Los clientes de las aplicaciones M2M y los de IoT ambos por igual tienen como objetivo reducir el tiempo de inactividad no planeado, y ambos tipos de soluciones potencialmente pueden mejorar la gestión de servicios. IoT sobresale aquí también, proporcionando la capacidad de evaluar estas cuestiones desde un nivel del sistema, así como en el dispositivo o el nivel de la máquina y la aplicación de análisis y el procesamiento de grandes volúmenes de datos para ajustar a los beneficios incrementales.

La confianza en el software frente a aspectos de hardware de la arquitectura hace que las soluciones de IoT sean más accesibles a una variedad más amplia de clientes internos y externos. Capacidades de visualización universales permiten que los datos se presenten en cualquier lugar, incluso en los dispositivos móviles, a aquellos usuarios sancionados. La combinación de estos atributos eleva aún más la visibilidad de las soluciones de IoT y genera la atención a nivel

corporativo, y no sólo a nivel departamental.

Supplier Characteristic	IoT	M2M
Value Chain/Number of Vendors Involved	Several to Many	One
Supplier Competencies	Device connectivity, networking, system integration, enterprise integration	Device connectivity, embedded modules, SIM card management and telecommunications
Competitive Nature	Coopetition	Competition
Solutions Emphasis	Device connectivity, data analysis, visualization	Embedded point solutions
Pricing Model	Varies: subscription fee per device, # of users, traffic, etc.	Monthly/annual subscription fee per device

Supplier Core Competencies and Value Chains Differ for IoT vs. M2M

Figura 2.3: Comparativa entre IoT y M2M

Tabla comparativa sobre los puntos más importantes entre internet de las cosas y máquina a máquina

1.2.3. Las diferencias en el proveedor de aplicaciones

Los proveedores de aplicaciones M2M y IoT suelen tener diferentes competencias. Esto afecta directamente a la capacidad de los usuarios para generar los beneficios deseados de sus soluciones de acceso al dispositivo remoto.

Las capacidades del proveedor M2M tienden a centrarse en los aspectos de "plomería" mencionados anteriormente, el hardware en particular incrustado y redes de telecomunicaciones celulares. Muchos están empezando a añadir capacidad de la nube mediante el desarrollo interno, la adquisición o asociación. Pero para la mayoría de los proveedores de M2M, esto representa un nuevo terreno. Proveedores de soluciones de IoT, por su parte, tienden a enfatizar las capacidades del software y la integración en particular de la empresa. Estas son importantes distinciones.

1.2.4. Especificar la solución correcta según IoT o M2M

Los términos M2M e IoT se han convertido en sinónimo en muchos sectores, pero es

importante asegurarse de que se especifique una solución que se adapte a las necesidades actuales y futuras. Esto implica el reconocimiento por adelantado si se busca una solución puntual para un acceso simple se una máquina remota, al igual que en una aplicación de gestión de servicios, o mira para conducir los beneficios incrementales de negocio en toda la empresa a través del uso de la analítica, Big Data y otros programas orientados al rendimiento de herramientas de mejora. Capacidades de integración empresarial, escalabilidad, software vs hardware, y el uso de la norma frente a las conexiones de dispositivos patentados son criterios clave que impactan si se tiene una solución IoT o M2M.

2. JWebSocket

JWebSocket [jWS 2014] es una solución desarrollada en Java y JavaScript para soportar comunicación de alta velocidad bidireccional en la Web

El proyecto JWebSocket se ha iniciado para crear aplicaciones innovadoras de transmisión y comunicación en la web basados en HTML5. HTML5 WebSockets sustituirán a los enfoques XHR existentes, así como servicios Comet por una nueva tecnología de comunicación socket TCP bidireccional de alta velocidad y ultra flexibles. JWebSocket es una implementación Java y JavaScript de código abierto del protocolo HTML5 WebSocket con un enorme conjunto de extensiones.

2.1. Componentes de JWebSocket

El paquete jWebSocket contiene:

- Servidor: un servidor WebSocket desarrollado en Java para comunicación servidor a cliente (S2C), soluciones de streaming y servidor controlado de comunicación cliente-a-cliente-servidor (C2C).

- Cliente: una solución cliente basada en JavaScript con el apoyo JSON y un usuario opcional, sesión y de gestión de tiempo de espera. Sin plug-ins necesarios.

El servidor jWebSocket se basa en tecnología Java (Java Runtime Environment (JRE) 1.6 o superior). Para configurar el servidor jWebSocket se necesita descargar el paquete de servidores jWebSocket (jWebSocketServer- <versión> .zip) desde el área de descarga que incluye la jWebSocketServer-<versión>.jar, todas las bibliotecas necesarias y jWebSocketServer- <versión> .bat desde la sección de descargas de esta carpeta. Descomprimir el archivo en una carpeta a elección este archivo contiene una jWebSocket- <versión> carpeta que es la carpeta raíz para el servidor jWebSocket. Esta contiene la siguiente estructura:

conf: contiene el archivo de configuración jWebSocket.xml.

libs: contiene el jWebSocketServer.jar asi como tambien todas las librerias requeridas por

jWebSocket. También los jars de las extensiones como filtros o plugins.

bin: Contiene los archivos ejecutables de Windows.

logs: Aquí se encuentran los archivos de log de jWebSocket.

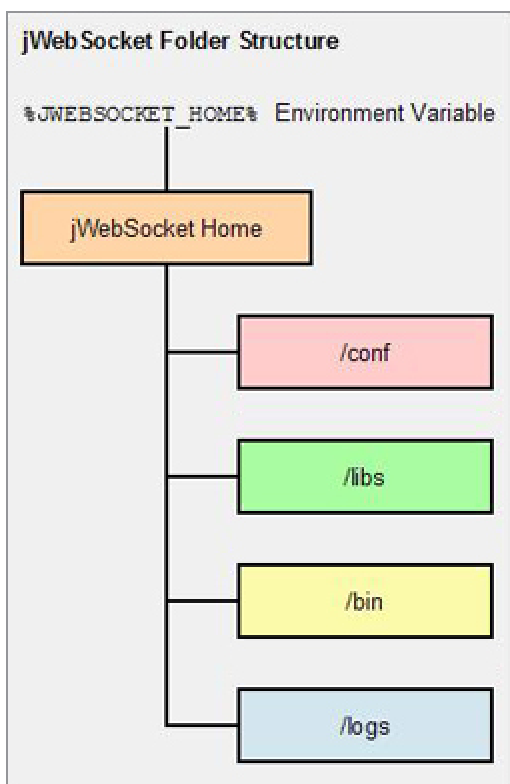

Figura 2.4: Estructura de la carpeta jWebSockets
Estructura de la carpeta del servidor de jWebSockets y sus subcarpetas.

2.2. Listeners, plugins y filters

Para implementar el servidor el proyecto de JWebSocket provee una serie de componentes:

Listeners: Los listeners son el mecanismo para procesar mensajes de los clientes WebSockets además de eventos como la conexión o desconexión. Los listeners son propios de cada aplicación (a diferencia de los plugins que sirven para encapsular funcionalidad).

Plugins: Los plugins amplían la funcionalidad jWebSocket servidor proporcionando métodos para procesar los mensajes entrantes de los clientes, así como otros eventos de cliente conectado o

desconectado. Los mensajes entrantes son filtradas por la cadena de filtros jWebSocket y por lo tanto siempre el mismo nivel de seguridad como los listeners hacen.

Filters: Filtros y cifrado son la parte más importante del modelo de seguridad en jWebSocket. Los filtros - ordenados en cadena - controlan el acceso a los distintos servicios WebSocket. Verifican tokens entrantes y salientes y opcionalmente los rechazan. Este principio se puede utilizar para eventos simples como la validación de argumentos, y también para la detección de virus durante la transferencia de archivos.

2.2.1. Listeners

Los listeners proporcionan una manera fácil de procesar los mensajes entrantes de los clientes WebSocket así como eventos de cliente conectado o cliente desconectado. Los mensajes entrantes son filtradas por la cadena de filtros jWebSocket y así proporcionan el mismo nivel de seguridad de los plug-ins.

A diferencia de los plugins, los listeners se incluyen y en el código de aplicación. De ahí que por lo general los listeners se supone que deben poner en práctica la lógica específica de la aplicación en lugar de la funcionalidad general. Opuesto a los plug-ins y su API el principal beneficio de un listener es que se puede directamente incrustarlo en el código, simplemente implementando una única interfaz.

El primer paso para implementar un jWebSocket Listener es crear una clase que implementa webSocketListener o WebSocketTokenListener tal como muestra el ejemplo.

```
01. public class JWebSocketTokenListenerSample implements WebSocketTokenListener {
02.
03.    private static Logger log =
       Logging.getLogger(JWebSocketTokenListenerSample.class);
04.
05.    public void processOpened(WebSocketEvent aEvent) {
06.       log.info("Client '" + aEvent.getSessionId() + "' connected.");
07.    }
08.
09.    public void processPacket(WebSocketEvent aEvent, WebSocketPacket aPacket) {
10.       // here you can process any non-token low level message, if desired
11.    }
12.
13.    public void processToken(WebSocketTokenEvent aEvent, Token aToken) {
14.       log.info("Client '" + aEvent.getSessionId() + "' sent Token: '" +
          aToken.toString() + "'.");
15.       // here you can interpret the token type sent from the client according to
          your needs.
16.       String lNS = aToken.getNS();
17.       String lType = aToken.getType();
18.
19.       // check if token has a type and a matching namespace
20.       if (lType != null && "my.namespace".equals(lNS)) {
21.          // create a response token
22.          Token lResponse = aEvent.createResponse(aToken);
23.          if ("getInfo".equals(lType)) {
24.             // if type is "getInfo" return some server information
25.             lResponse.put("vendor", JWebSocketConstants.VENDOR);
26.             lResponse.put("version", JWebSocketConstants.VERSION_STR);
27.             lResponse.put("copyright", JWebSocketConstants.COPYRIGHT);
28.             lResponse.put("license", JWebSocketConstants.LICENSE);
29.          } else {
30.             // if unknown type in this namespace, return corresponding error
                message
31.             lResponse.put("code", -1);
32.             lResponse.put("msg", "Token type '" + lType + "' not supported in
                namespace '" + lNS + "'.");
33.          }
34.          aEvent.sendToken(lResponse);
35.       }
36.    }
37.
38.    public void processClosed(WebSocketEvent aEvent) {
39.       log.info("Client '" + aEvent.getSessionId() + "' disconnected.");
40.    }
41. }
```

Figura 2.5: Ejemplo de clase webSocketTokenListener.
Clase que ejemplifica la implementación de un webSocketTokenListener

```
1. // get the token server
2. TokenServer lServer = (TokenServer)JWebSocketFactory.getServer("ts0");
3. if( lServer != null ) {
4.     // and add the sample listener to the server's listener chain
5.     lServer.addListener(new JWebSocketTokenListenerSample());
6. }
```

Figura 2.6: Ejemplo para inclusión de un listener en el servidor de jWebSocket
Ejemplo para incluir un listener de webSocket en el servidor de jWebSockets

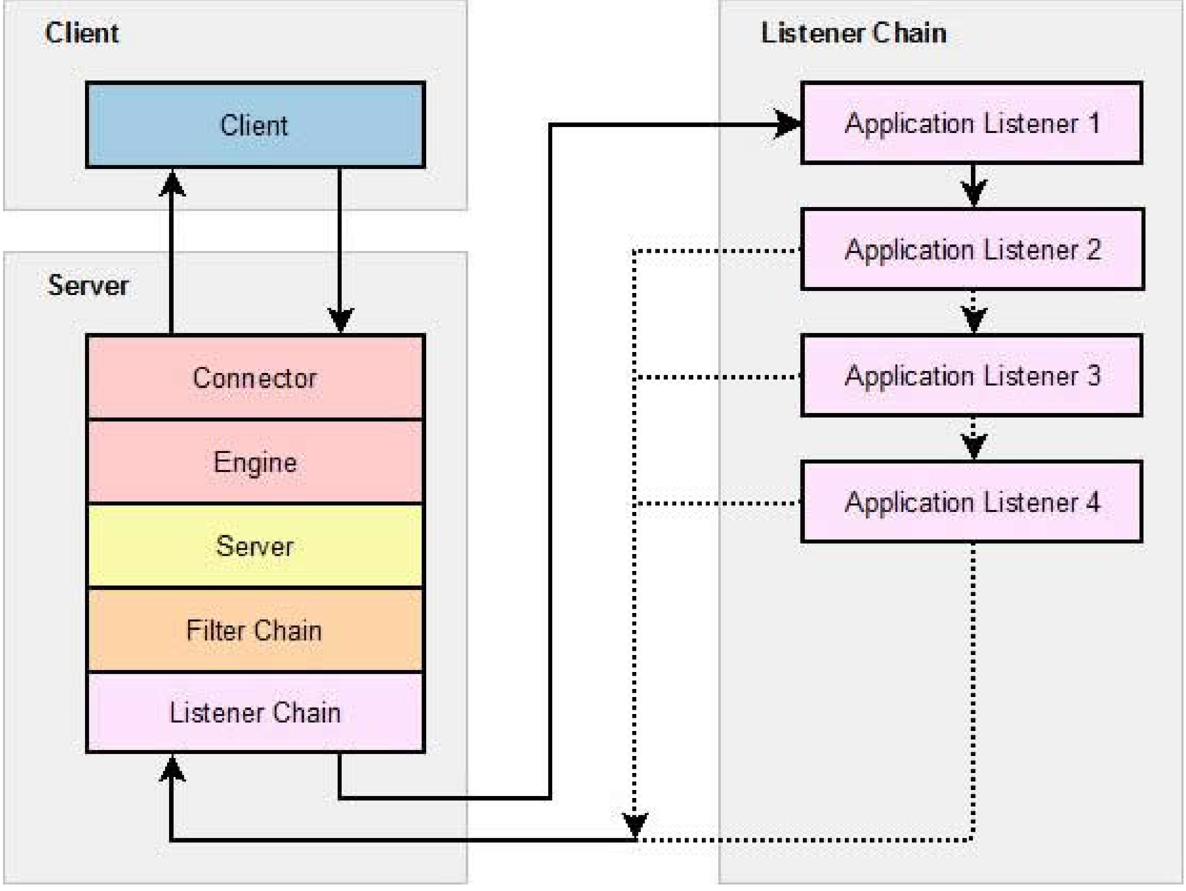

Figura 2.7: Infraestructura de la cadena de listeners
Gráfico que ejemplifica la infraestructura de la cadena de listeners en el servidor

2.2.2. Plugins

En jWebSocket casi toda la funcionalidad se implementa en los plug-ins. Para ello el jWebSocket Server proporciona las capacidades de gestión necesarias. Una de estas capacidades es la cadena de plug-in. En esta cadena de múltiples plug-ins están dispuestos en un orden

determinado. El orden de los plugins se puede cambiar de forma arbitraria y nuevos plugins pueden ser fácilmente adjuntados o insertados. Básicamente, un plugin puede procesar un número casi ilimitado de comandos del cliente mediante la interpretación de los diversos campos de datos dinámicos de las tokens. Un plugin complejo puede constar de varias clases en un paquete separado que puede estar insertado o enlazado dinámicamente en tiempo de ejecución como un archivo .jar separada e incluso distribuible.

Este concepto le permite implementar fácilmente y distribuir nuevos servicios WebSocket de acuerdo a sus necesidades y de sus clientes en base a lo jWebSocket ya proporciona.

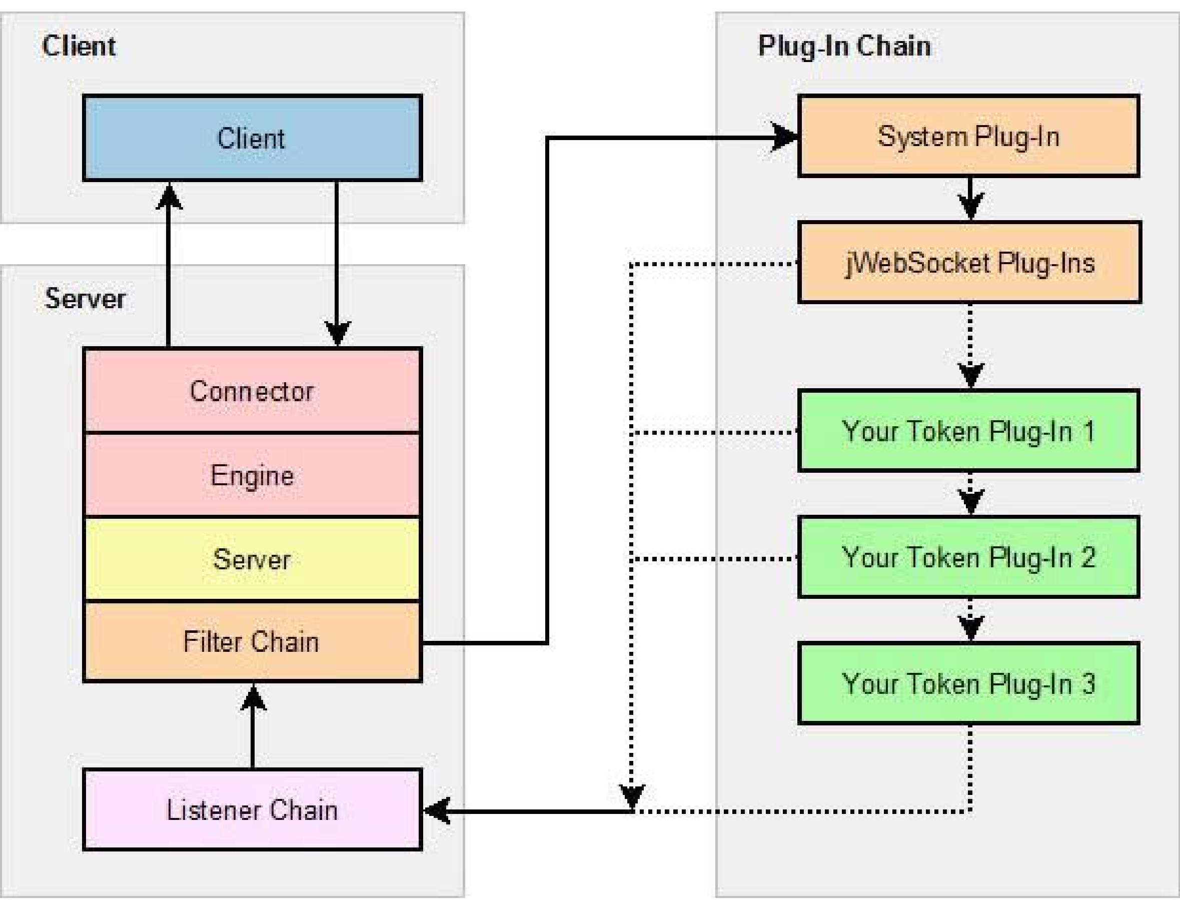

Figura 2.8: Infraestructura de la cadena de plugins
Gráfico que ejemplifica la infraestructura de la cadena de plugins en el servidor

```
01. <!-- plug-ins to be instantiated for jWebSocket -->
02. <plugins>
03.    <plugin>
04.       <name>org.jwebsocket.plugins.system.SystemPlugIn</name>
05.       <ns>org.jWebSocket.plugins.system</ns>
06.       <id>jws.system</id>
07.       <jar>jWebSocketPlugins-<version>.jar</jar>
08.       <!-- plug-in specific settings -->
09.       <settings>       <!-- specify whether open,close,login,logout event should
          be broadcasted -->
10.          <setting key="broadcastOpenEvent">true</setting>
11.          <setting key="broadcastCloseEvent">true</setting>
12.          <setting key="broadcastLoginEvent">true</setting>
13.          <setting key="broadcastLogoutEvent">true</setting>
14.       </settings>
15.       <server-assignments>
16.          <server-assignment>ts0</server-assignment>
17.       </server-assignments>
18.    </plugin>    :
19.    <plugin>
20.       <name>org.jwebsocket.plugins.flashbridge.FlashBridgePlugIn</name>
21.       <id>jws.flashbridge</id>
22.       <ns>org.jWebSocket.plugins.flashbridge</ns>
23.       <jar>jWebSocketPlugins-<version>.jar</jar>
24.       <server-assignments>
25.          <server-assignment>ts0</server-assignment>
26.       </server-assignments>
27.    </plugin>
28. </plugins>
```

Figura 2.9: Configuración para incluir un plugin

Figura que muestra el xml de ejemplo para incluir un plugin al servidor

2.2.3. Filters

Un administrador del servidor puede instalar filtros de entrada y / o salida para garantizar que los paquetes entrantes con contenido cuestionable son rechazados y los paquetes salientes ominosas se suprime.

Si los filtros deben funcionar como una instancia de la seguridad global independiente de los plugins u listeners, pueden ser cargados como .jars separados en el servidor.

En la cadena de comunicación cada paquete entrante pasa a través de los filtros de entrada antes de cualquier proceso posterior, así como cada paquetes de salida pasa por los filtros de salida antes de enviar a los clientes.

Los filtros también se pueden usar dentro de las aplicaciones en lugar de archivos .jar, por ejemplo, para la validación de paquetes. En este caso, una nueva clase simplemente implementa la interfaz filtro con dos métodos simples: processTokenIn(...) y processTokenOut(...).

```xml
<!-- filters to be instantiated for jWebSocket -->
<filters>
   <filter>
      <name>org.jwebsocket.filters.system.SystemFilter</name>
      <ns>org.jWebSocket.filters.system</ns>
      <id>systemFilter</id>
      <jar>jWebSocketPlugins-<version>.jar</jar>
      <!-- plug-in specific settings -->
      <settings>
         <!-- specify whether open,close,login,logout event should be broadcasted
         -->
         <setting key="key">value</setting>
      </settings>
      <server-assignments>
         <server-assignment>ts0</server-assignment>
      </server-assignments>
   </filter>
   <filter>
      <name>org.jwebsocket.filters.custom.CustomTokenFilter</name>
      <id>userFilter</id>
      <ns>org.jWebSocket.filters.custom</ns>
      <jar>jWebSocketPlugins-<version>.jar</jar>
      <server-assignments>
         <server-assignment>ts0</server-assignment>
      </server-assignments>
   </filter>
</filters>
```

Figura 2.10: Configuración para incluir un filtro
Xml de configuración para incluir un filtro en el server

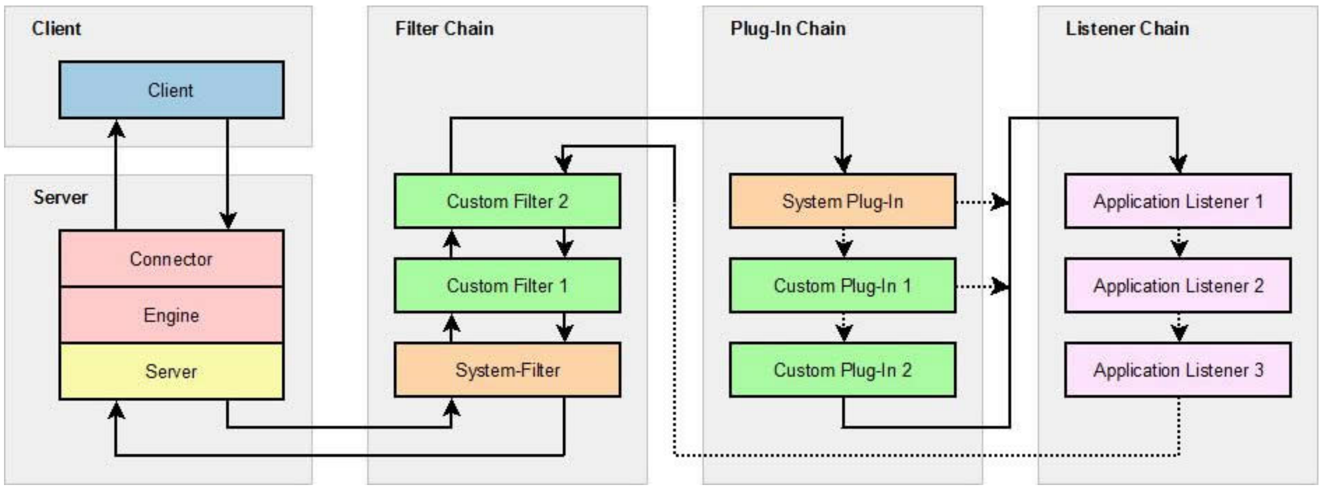

Figura 2.11: Infraestructura completa de la cadena del servidor
Gráfico que ejemplifica la infraestructura de la cadena completa en el servidor

2.3. Conectores, motores y servidores.

El protocolo WebSocket se basa en simple comunicación socket TCP con un saludo inicial. Después de que el saludo inicial se ha procesado con éxito la conexión proporciona un canal bidireccional full duplex de comunicación entre el servidor y el cliente. En función de su dimensión un único servidor puede manejar cientos de conexiones de cliente simultáneas. Debido a la estructura extensible del enfoque jWebSocket varios servidores pueden agruparse para apoyar un número casi ilimitado de clientes.

En jWebSocket cada cliente se conecta a un conector del jWebSocket Server. Los conectores son impulsados por un motor como por ejemplo, la TCPEngine interna jWebSocket o motores de terceros como JBoss Netty. El núcleo de servidor en jWebSocket puede controlar múltiples motores para beneficiarse de sus características individuales como el soporte SSL. El siguiente diagrama primero le da una visión general de la infraestructura de comunicación jWebSocket.

System-Plug-Ins	User Plug-Ins	User Listener 1	User Listener 2	Custom Plug-In 1	Custom Plug-In 2	Custom Listener	Custom Application w/o Plug-In Support
Token Plug-In Chain		Token Listener Chain		Custom Plug-In-Chain		Listener Chain	
Token Server				Custom Server			
JSON Processor	CSV Processor	XML* Processor	Custom Processor	Custom Packet Processor			
Server							
TCP Engine		Netty Engine		MINA Engine*		Jetty Engine*	
TCP Connector	TCP Connector	Netty Connector	Netty Connector	MINA Connector	MINA Connector	Jetty Connector	Jetty Connector

Figura 2.12: Conectores, motores y servidores
Infraestructura del server JwebSockets.

2.4. Token

Un Token es un objeto que contiene uno o varios campos y valores. En Java la clase org.jWebSocket.token.Token incrusta una clase HashMap, que posee varios campos con sus valores correspondientes. A diferencia de JSON o XML, CSV sólo pueden tener campos simples, es decir, que no pueden contener estructuras de objetos complejos.

¿Por qué utilizar diferentes formatos Token?

Cada uno de los JSON, CSV y XML tienen sus beneficios para un propósito en particular en lo que se describe a continuación:

JSON puede ser interpretado fácilmente por los clientes de JavaScript, pero no es tan fácil por los clientes de Java. JSON puede contener los riesgos de seguridad, ya que puede ejecutar código malicioso en el cliente. Así que aquí se tiene que tener cuidado en el lado del servidor que los códigos ejecutables potenciales son despojados de un token. JSON es la mejor opción si usted interactúa principalmente con clientes de navegador y ejecutar un filtro etiqueta script fiable en el servidor.

```
{
field1:value1;
field2:"value2";
field3:  [
         arrayitem1,
         arrayitem2
         ];
field4:  {
         objfield1:value1;
         objfield2:value2
         }
}
```

CSV es el formato más compacto, sin embargo, un inconveniente es que no soporta la estructura objeto complejo, son objetos que contienen tipos de datos simples solamente. CSV es la mejor opción si se tiene que cambiar una gran cantidad de datos con estructuras planas.

```
campo1 = valor1,
campo2 = "valor2",
campo3 = value3
```

XML es el formato más flexible, pero también el formato con el más alto overhead. XML puede contener estructuras de objetos arbitrarios. Por lo tanto XML es la mejor opción si no se tiene que enviar una gran cantidad de datos de manera flexible.

2.4.1 Tokens desde el servidor al cliente

welcome: El token de bienvenida se envía desde el servidor a un cliente, cuando se ha establecido la conexión exitosamente. El token de bienvenida es el único token que contiene la sesión-id para el cliente. No se volverá a enviar de nuevo por el servidor ni puede ser solicitado por el cliente durante el período de sesiones. En todos los casos el método processOpened del cliente se ejecuta tan pronto como se establezca la conexión.

goodBye: El token de despedida se envía desde el servidor a un cliente como respuesta a una solicitud de cierre. El método close del cliente JavaScript soporta una opción de tiempo de espera. Si el tiempo de espera es <= 0, el cliente se desconecta inmediatamente. Si el tiempo de espera es> 0, el cliente envía una señal close al servidor y espera milisegundos de tiempo de espera para la respuesta de despedida. En este caso, el servidor se desconecta después de responder con el mensaje. Si el cliente no recibe la despedida en el tiempo de espera se desconecta de todas formas. En todos los casos el método processClosed del cliente se ejecuta tan pronto como la conexión se terminó.

Response: El token de respuesta se envía como una respuesta de un destino a una solicitud anterior de un cliente. Una respuesta bien puede ser enviado desde el servidor, por ejemplo cuando se ejecutan las llamadas a procedimiento remoto (RPC) o de otro cliente por ejemplo, cuando se ejecuta identificar peticiones.

Event: El token evento se envía como un mensaje del servidor u otro cliente sin una solicitud previa. Los eventos se dispararon cuando otro cliente se conecta o se desconecta o cuando el lado de espera de sesión del servidor es superado y la conexión está a punto de cerrarse.

2.4.2 Tokens desde el cliente a servidor

Estas tokens se pueden asociar con "comandos" desde el cliente al servidor que normalmente respondió con un token de respuesta, si no fue explícitamente borrado por ciertas razones. Los resultados se envían desde el servidor al cliente en un token de respuesta. Por lo general, los campos de resultado, el código y msg se llenan en la respuesta. Debido a que JavaScript no admite llamadas síncronas cada comando proporciona un listener OnResponse opcional que se dispara cuando llega el token de respuesta.

Login: Autentica un cliente después de la conexión se ha establecido. El cliente debe esperar hasta que se reciba la respuesta del servidor antes de notificar al usuario sobre el estado de inicio de sesión o un posible error. Si se envía una señal de inicio de sesión, mientras que otro usuario está autenticado, el usuario anterior automáticamente cerrará la sesión.

Logout: Desregistra el usuario actual pero mantiene la conexión. Opcionalmente otro usuario puede autenticar después basado en la misma conexión subyacente.

Close: El método close del cliente JavaScript apoya la opción de tiempo de espera. Si el tiempo de espera es <= 0, el cliente se desconecta inmediatamente. Si el tiempo de espera es> 0, el cliente envía una señal close al servidor y espera milisegundos de tiempo de espera para la respuesta adiós. En este caso, el servidor desconecta después de responder con el adiós. Si el cliente no recibe la despedida en el tiempo de espera dado se desconecta de todas formas.

Send: El token de envío se remitirá en el servidor al cliente según el targetId dado. En el campo ResponseRequested el remitente puede especificar si desea o no obtener una respuesta. En el caso de errores, por ejemplo, cuando se pudo encontrar ningún cliente con el targetId dado, se devuelve siempre una respuesta con un error.

Vale considerar que un determinado cliente no puede abordarse mediante su nombre de usuario, sino por su ID de cliente único, porque en el fondo, un usuario puede iniciar sesión en múltiples estaciones o instancias del navegador / pestañas, por supuesto, sólo si la aplicación lo permite. Además, un cliente no necesita necesariamente ser autenticado para recibir los mensajes del servidor.

Broadcast: El token de *broadcast* se transmite por el servidor para todos los clientes conectados actualmente, incluyendo opcionalmente el remitente para propósitos especiales. El servidor envía opcionalmente una respuesta. Todo depende de los otros clientes para enviar una respuesta al remitente. Entre otros la transmisión se puede utilizar para, por ejemplo, para los sistemas de chat o para distribuir actualizaciones de un jugador en aplicaciones de juegos, así como para transmitir e identificar las solicitudes en el caso de que el servidor está configurado para no enviar automáticamente la conexión y desconexión.

Echo: El token *echo* envía un mensaje al servidor. El cliente espera un resultado con los mismos datos. Por lo general, las aplicaciones no utilizan este modo, a excepción de las pruebas de conexión y de funcionamiento. A los efectos de mantener activa consulte el token ping.

Ping: El token *ping* es un mensaje simple y corto de un cliente al servidor para indicar que el cliente todavía está vivo. Si el servidor no recibe datos dentro de su tiempo de espera de la sesión se cierra automáticamente la conexión después de que se supere el tiempo de espera.

GetClients: El token de *getClients* retorna la lista de solicitudes de clientes del servidor. Con la opción de modo (por defecto = 0), el cliente puede especificar si se debe devolver todo, cliente autenticado o no autenticadas. El campo de resultado de la respuesta contiene una matriz con los clientes solicitados en el formato [nombre de usuario | guión si no conectado] @ [clientId].

2.5 Canales

Los canales son un modelo de comunicación muy eficaz en jWebSocket para permitir múltiples aplicaciones o varios módulos dentro de una única aplicación para interactuar en los canales lógicos separados sin afectar a los demás. Usted encontrará una descripción de los distintos modelos de comunicación en jWebSocket en nuestra sección de Prensa en el artículo "en tiempo real

con varias plataformas de comunicación con WebSockets". Por favor, consulte también la sección de Infraestructuras y seleccione Canales.

El jWebSocket Canal Plug-in ofrece las siguientes opciones para los canales:

Canales públicos: Todos los clientes pueden suscribirse a un canal público para escuchar en él y opcionalmente publicar datos sobre el canal. A diferencia de los canales privados canales públicos se muestran por la petición getChannels. Opcionalmente canales públicos pueden ser protegidos por una clave de acceso, así como por una clave secreta. Si un canal está protegido el cliente tiene que pasar la clave de acceso para suscribirse a ese canal. Si la clave secreta opcional está configurado para el canal que el cliente tiene que pasar esa clave secreta para publicar mensajes en el canal.

Canales privados: Los canales privados se supone para la comunicación no revelada entre 2 o más clientes. Por tanto, no aparecen en el resultado getChannels. Además canal privado debe ser asegurado por un / par de claves secreto clave de acceso. Para suscribirse a un canal privado de un cliente debe conocer el canal-id y pasar la clave de acceso correcta. El envío de mensajes en un canal privado sólo está permitido al conocer la clave secreta obligatoria.

Canales del sistema: Canales del sistema están reservados para ser utilizados por el servidor y por lo general son creados por el servidor sólo (es decir, a través del archivo de configuración jWebSocket.xml). Canales del sistema no pueden ser actualizados o eliminados por el cliente.

2.5.1. Configuración de Canales

Los canales se crean ya sea en la configuración de arranque del servidor usando el archivo channels.xml o en tiempo de ejecución mediante el método *createChannel* del ChannelPlugIn. El siguiente fragmento XML muestra cómo configurar un canal público:

```
01. <!-- THE "<bean>" TAG IS A DEFAULT CONFIGURATION FOR THE CHANNEL
02.    AND SHOULD NOT BE CHANGED FROM THE USER  -->
03. <bean id="channelManager" class="org.jwebsocket.plugins.channels.ChannelManager"
04.      init-method="initialize">
05.       <constructor-arg>
06.            <map>
07.              <!-- set this to true if you want to allow
08.                               to create new system channels from the
09.                               client side, default is false, not
10.                               recommended!
11.              <entry key="allowCreateSystemChannels">
12.              <value>false</value>
13.              </entry>-->
14.                             <!-- HERE SHOULD BE ADDED ALL THE CHANNELS
15.                             THE USER WANTS USING THE TAG:
16.                             "<entry><value></entry></value>" -->
17.              <entry key="channel:privateA">
18.                  <value>{
19.                                 state:"CREATED",
20.                                 owner:"root",
21.                                 name:"Private A",
22.                                 isPrivate:true,
23.                                 isSystem:false,
24.                                 access_key:"access",
25.                                 secret_key:"secret",
26.                                 token_server:"ts0"
27.                                 }</value>
28.              </entry>
29.            </map>
30.       </constructor-arg>
31.              <constructor-arg ref="storageProvider"/>
32.       </bean>
```

Figura 2.13: Configuración para canales

Xml para la creación de canales

2.6. Cliente Javascript para un navegador web.

Tal como se describe en la siguiente figura se puede ver la infraestructura del cliente web de jWebSockets.

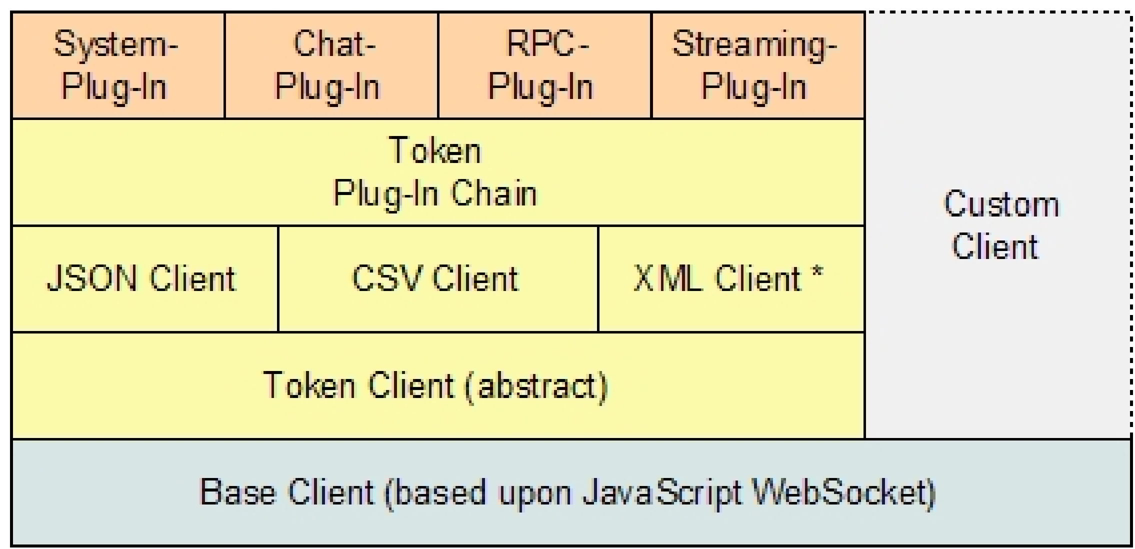

Figura 2.14: Infraestructura del cliente jWebSocket

2.6.1. Hola mundo!

Una comunicación WebSocket entre el cliente y el servidor necesita ser iniciada por el cliente. Una vez establecida la conexión, el cliente envía mensajes ya sea en el servidor o a otros clientes a través del servidor. En la dirección opuesta el servidor envía mensajes al cliente mediante el uso de la misma conexión. A menos que la conexión se termine, ya sea por el servidor o por el cliente de ambos lados pueden bidireccionalmente intercambiar mensajes arbitrariamente.

Lo único necesario para usar jWebSocket en las páginas web es poner una sola etiqueta de script en la sección head del código HTML.

```
<script type="text/javascript" src="<path_to_jWebSocket.js>/jwebsocket.js">
</script>
```

jWebSocket proporciona la clase jWebSocketJSONClient dentro del espacio de nombres específico jWebSocket. Esta clase proporciona los métodos para conectarse y desconectarse, así como para el intercambio de mensajes con el servidor mediante el protocolo JSON. El espacio de nombres evita conflictos de nombres con otros frameworks.

```
01. // jws.browserSupportsWebSockets checks if web sockets are available
02. // either natively, by the FlashBridge or by the ChromeFrame.
03. if( jws.browserSupportsWebSockets() ) {
04.    jWebSocketClient = new jws.jWebSocketJSONClient();
05.    // Optionally enable GUI controls here
06. } else {
07.    // Optionally disable GUI controls here
08.    var lMsg = jws.MSG_WS_NOT_SUPPORTED;
09.    alert( lMsg );
10. }
```

Figura 2.15: Instanciación del cliente de jWebSocket
Código de ejemplo para instanciar un cliente javascript.

Con el fin de iniciar la conexión desde el cliente al servidor el método de inicio de sesión de la jWebSocketClient se puede utilizar. Este método se conecta al servidor y pasa el nombre de usuario y la contraseña para la autenticación.

```
01. log( "Connecting to " + lURL + " and logging in as '" + gUsername + "'..." );
02. var lRes = jWebSocketClient.logon( lURL, gUsername, lPassword, {
03.    // OnOpen callback
04.    OnOpen: function( aEvent ) {
05.       log( "<font style="color:#888">jWebSocket connection established.</font>"
          );
06.    },
07.    // OnMessage callback
08.    OnMessage: function( aEvent, aToken ) {
09.       log( "<font style="color:#888">jWebSocket '" + aToken.type + "' token
          received, full message: '" + aEvent.data + "'</font>" );
10.    },
11.    // OnClose callback
12.    OnClose: function( aEvent ) {
13.       log( "<font style="color:#888">jWebSocket connection closed.</font>" );
14.    }
15. });
```

Figura 2.16: Autenticación contra el servidor jWebSocket
Código de ejemplo para la autenticación del cliente javascript.

El servidor asigna un identificador único para el cliente, por lo que un determinado cliente siempre puede abordarse de forma única, incluso si con las mismas credenciales de usuario se inicia sesión en varias instancias del navegador.

Si la conexión se ha establecido correctamente el cliente envía sus mensajes a través del método de envío a otro cliente o lo transmite a todos los clientes conectados mediante el método de difusión de la jWebSocketClient.

```
01. // lMsg is a string
02. if( lMsg.length > 0 ) {
03.    var lRes = jWebSocketClient.broadcastText(
04.       "",    // broadcast to all clients (not limited to a certain pool)
05.       lMsg   // broadcast this message
06.    );
07.    if( lRes.code != 0 ) {
08.       // display error
09.    }
10. }
```

Figura 2.17: Mensaje de broadcast
Ejemplo para mandar un mensaje de broadcast.

El envío de mensajes es siempre no-bloqueante, es decir, el *send* y *broadcast* no esperan a que se devuelva un resultado. Un resultado opcional se devuelve de forma asincrónica.

Los mensajes del servidor al cliente son enviados de forma asíncrona. Por lo tanto la clase jWebSocketClient ofrece el evento OnMessage. Como ya se ha mostrado anteriormente en el método de inicio de sesión la aplicación simplemente añade un listener a este evento y procesa el mensaje como deseado.

```
1. :
2. // OnMessage callback
3. OnMessage: function( aEvent, aToken ) {
4.    log( "<font style='color:#888'>jWebSocket '" +        aToken.type + "' token
      received, full message: '" +        aEvent.data + "'</font>"
5.    );
6. },
7. :
```

Figura 2.18: Procesamiento de mensajes entrantes
Ejemplo para el procesamiento de mensajes entrantes.

Bajo demanda tanto el servidor como el cliente pueden terminar una conexión existente. En el cliente esto se hace por el método de cierre de la jWebSocketClient.

```
1. if( jWebSocketClient ) {
2.    jWebSocketClient.close();
3. }
```

Figura 2.19: Cierre de la conexión en el cliente
Ejemplo de cierre de conexión.

El servidor termina automáticamente la conexión después de un cierto tiempo de inactividad en la conexión. En este caso, el evento que se dispara es el OnClose que puede ser manejado por el callback correspondiente como se muestra en el método de inicio de sesión anterior. El tiempo de espera se puede configurar y se puede ejecutar opcionalmente un keep-alive o un mecanismo de control de reconexión.

Capítulo 3: Tecnologías asociadas

En el capítulo dos hablamos de un posible framework a utilizar para la implementación de la comunicación sobre Internet de las cosas. Ya que provee de comunicación vía webSockets, canales de comunicación, un server al que se puede extender de varia maneras via plugins o listeners. Sin embargo no cuenta con funcionalidad necesaria para un mundo tan cambiante como el que propone internet de las cosas. Para esto se incorporan nuevas tecnologías o paradigmas que no son tenidos en cuenta en una solución como la de jWebSockets.

3.1. Paquetes UDP

User Datagram Protocol (UDP) [W3 UDP 2015] es un protocolo del nivel de transporte basado en el intercambio de datagramas (Encapsulado de capa 4 Modelo OSI). Permite el envío de datagramas a través de la red sin que se haya establecido previamente una conexión, ya que el propio datagrama incorpora suficiente información de direccionamiento en su cabecera. Tampoco tiene confirmación ni control de flujo, por lo que los paquetes pueden adelantarse unos a otros; y tampoco se sabe si ha llegado correctamente, ya que no hay confirmación de entrega o recepción. Su uso principal es para protocolos como DHCP, BOOTP, DNS y demás protocolos en los que el intercambio de paquetes de la conexión/desconexión son mayores, o no son rentables con respecto a la información transmitida, así como para la transmisión de audio y vídeo en real, donde no es posible realizar retransmisiones por los estrictos requisitos de retardo que se tiene en estos casos.

UDP no otorga garantías para la entrega de sus mensajes (por lo que realmente no se debería encontrar en la capa 4) y el origen UDP no retiene estados de los mensajes UDP que han sido enviados a la red. UDP sólo añade multiplexado de aplicación y suma de verificación de la cabecera y la carga útil. Cualquier tipo de garantías para la transmisión de la información deben ser implementadas en capas superiores.

La cabecera UDP consta de 4 campos de los cuales 2 son opcionales. Los campos de los puertos fuente y destino son campos de 16 bits que identifican el proceso de origen y recepción. Ya que UDP carece de un servidor de estado y el origen UDP no solicita respuestas, el puerto origen es opcional. En caso de no ser utilizado, el puerto origen debe ser puesto a cero. A los campos del puerto destino le sigue un campo obligatorio que indica el tamaño en bytes del datagrama UDP incluidos los datos. El valor mínimo es de 8 bytes. El campo de la cabecera restante es una suma de comprobación de 16 bits que abarca una pseudo-cabecera IP (con las IP origen y destino, el protocolo y la longitud del paquete UDP), la cabecera UDP, los datos y 0's hasta completar un múltiplo de 16. El checksum también es opcional en IPv4, aunque generalmente se utiliza en la práctica (en IPv6 su uso es obligatorio). A continuación se muestra los campos para el cálculo del checksum en IPv4.

El protocolo UDP se utiliza por ejemplo cuando se necesita transmitir voz o vídeo y resulta más importante transmitir con velocidad que garantizar el hecho de que lleguen absolutamente todos los bytes.

La mayoría de las aplicaciones claves de Internet utilizan el protocolo UDP, incluyendo: el Sistema de Nombres de Dominio (DNS), donde las consultas deben ser rápidas y solo contarán de una sola solicitud, luego de un paquete único de respuesta, el Protocolo de Administración de Red

(SNMP), el Protocolo de Información de Enrutamiento (RIP) y el Protocolo de Configuración dinámica de host(DHCP).

Las características principales de este protocolo son:

1. Trabaja sin conexión, es decir que no emplea ninguna sincronización entre el origen y el destino.

2. Trabaja con paquetes o datagramas enteros, no con bytes individuales como TCP. Una aplicación que emplea el protocolo UDP intercambia información en forma de bloques de bytes, de forma que por cada bloque de bytes enviado de la capa de aplicación a la capa de transporte, se envía un paquete UDP.

3. No es fiable. No emplea control del flujo ni ordena los paquetes.

4. Su gran ventaja es que provoca poca carga adicional en la red ya que es sencillo y emplea cabeceras muy simples.

5. Un paquete UDP puede ser fragmentado por el protocolo IP para ser enviado fragmentado en varios paquetes IP si resulta necesario.

6. Puesto que no hay conexión, un paquete UDP admite utilizar como dirección IP de destino la dirección de broadcast o de multicast de IP. Esto permite enviar un mismo paquete a varios destinos.

3.2. WebSockets

Internet se ha creado en gran parte a partir del llamado paradigma solicitud/respuesta de HTTP. Un cliente carga una página web y no ocurre nada hasta que el usuario ejecuta una acción. Por el año 2005, AJAX [W3 AJAX] empezó a hacer que Internet pareciera más dinámico. Aún así, todas las comunicaciones HTTP eran dirigidas por el cliente, lo que requería la interacción del usuario o hacía necesario preguntarle periódicamente cada vez que se cargaban nuevos datos del servidor.

Hace ya algún tiempo que existen tecnologías que permiten al servidor enviar datos al cliente en el mismo momento que detecta que hay nuevos datos disponibles. Se conoce como "Push" o "Comet" [Comet - Server Push]. Uno de las formas más comunes para crear la ilusión de una conexión iniciada por el servidor se denomina *Long Polling*. Con el *Long Polling*, el cliente abre una conexión HTTP con el servidor, el cual la mantiene abierta hasta que se envíe una respuesta. Cada vez que el servidor tenga datos nuevos, enviará la respuesta. El *Long Polling* y otras técnicas funcionan bastante bien. Se utilizan todos los días en aplicaciones como el chat de Gmail.

Sin embargo, todas estas soluciones comparten un problema: el exceso de HTTP, lo que no las hace aptas para aplicaciones de baja latencia.

La especificación WebSocket define un API que establece conexiones "socket" entre un navegador web y un servidor. Existe una conexión persistente entre el cliente y el servidor, y ambas partes pueden empezar a enviar datos en cualquier momento.

Cuando se establece una conexión con el servidor (cuando el evento *open* se active), se

puede enviar datos al servidor con el método *send*('nuestro mensaje') en el objeto de conexión. Antes sólo se admitían strings, pero la última especificación también permite enviar mensajes binarios. Para enviar datos binarios, se puede usar un objeto *Blob* o *ArrayBuffer*.

De la misma forma, el servidor puede enviar mensajes en cualquier momento. Cada vez que esto ocurra, se activa la devolución de llamada *onmessage*. La devolución de llamada recibe un objeto "*event*", lo que permite acceder al mensaje actual mediante la propiedad "*data*".

También permite recibir mensajes binarios. Los marcos binarios se pueden recibir en formato *Blob* o *ArrayBuffer*. Para especificar el formato del binario recibido, se establece la propiedad *binaryType* del objeto *WebSocket* en "*blob*" o en "*arraybuffer*". El formato predeterminado es "*blob*".

WebSocket es una tecnología que proporciona un canal de comunicación bidireccional y full-duplex sobre un único socket TCP. Está diseñado para ser implementado en navegadores y servidores web, pero puede utilizarse por cualquier aplicación cliente/servidor. La API de WebSocket está siendo normalizada por el W3C, y el protocolo WebSocket, a su vez, está siendo normalizado por el IETF [IETF 2010]. Como las conexiones TCP ordinarias sobre puertos diferentes al 80 son habitualmente bloqueadas por los administradores de redes, el uso de esta tecnología proporciona una solución a este tipo de limitaciones proveyendo una funcionalidad similar a la apertura de varias conexiones en distintos puertos, pero multiplexando diferentes servicios WebSocket sobre un único puerto TCP (a costa de una pequeña sobrecarga del protocolo).

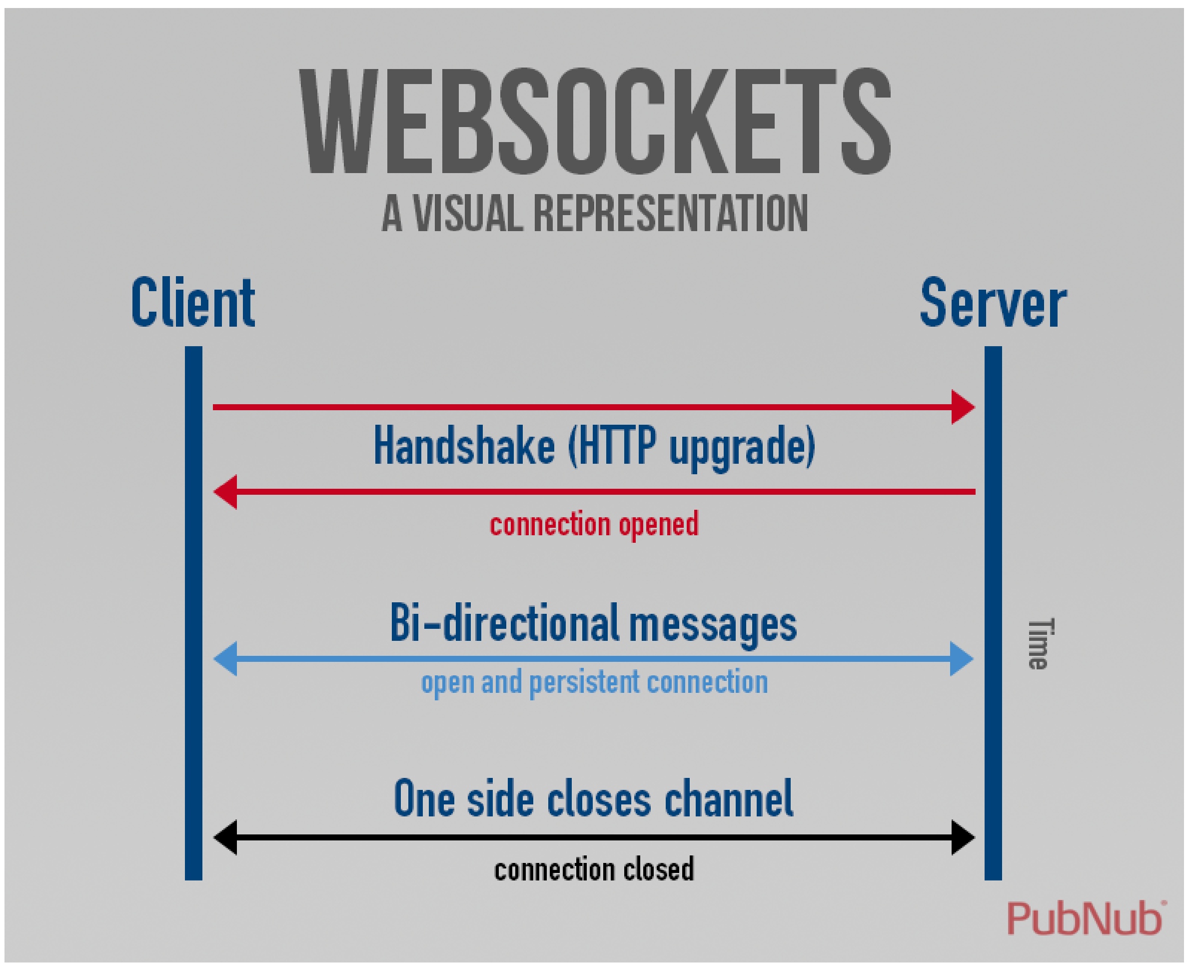

Figura 3.1: Comunicación con webSockets
WebSockets permite la comunicación bidireccional entre el cliente y el servidor.

3.3. Publicación / Suscripción

La mensajería de publicación/suscripción le permite separar al proveedor de información de los consumidores de dicha información. Para poder enviar y recibir información, no es necesario que la aplicación de envío y la de recepción sepan nada la una de la otra.

La publicación/suscripción [Pub-Sub] elimina la necesidad de que la aplicación emisora sepa algo sobre la aplicación de destino. Lo único que tiene que hacer la aplicación emisora es transferir un mensaje que contenga la información que desea transmitir y asignarle un tema, que denote el asunto de la información y dejar que el hub maneje la distribución de dicha información. Del mismo modo, no es necesario que la aplicación de destino sepa nada sobre el origen de la información que recibe.

El sistema de publicación/suscripción más sencillo con un un publicador, un gestor de colas, y un suscriptor. Se envía una suscripción del suscriptor al gestor de colas, se envía una publicación

del publicador al gestor de colas y, a continuación, el gestor de colas reenvía la publicación al suscriptor.

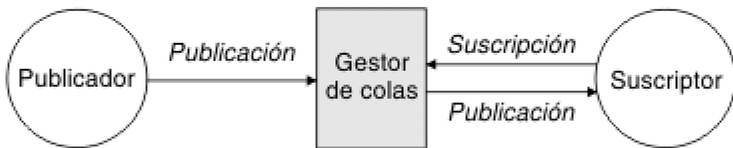

Figura 3.2: Esquema de publicación/suscripción simple

En un esquema publicación/suscripción se utiliza un gestor intermedio para gestionar el envío de información entre las partes

Un sistema de publicación/suscripción típico tiene más de un publicador y más de un suscriptor y, a menudo, más de un gestor de colas. Una aplicación puede ser tanto un publicador como un suscriptor.

En el modelo publicación/suscripción, los suscriptores reciben por lo general sólo un subconjunto de los mensajes totales publicados. El proceso de selección de los mensajes para la recepción y el procesamiento se denomina filtrado. Hay dos formas comunes de filtrado: basados en el contenido o en temas.

En un sistema basado en el tema, los mensajes se publican en "temas" o canales lógicos con nombre. Los suscriptores en un sistema basado en el tema recibirán todos los mensajes publicados en los temas a los que se suscriben, y todos los suscriptores a un tema recibirán los mismos mensajes. El editor es responsable de definir las clases de mensajes a los que los suscriptores pueden suscribirse.

En un sistema basado en el contenido, mensajes sólo se entregan a un abonado si los atributos o el contenido de esos mensajes coinciden con restricciones definidas por el abonado. El suscriptor es responsable de clasificar los mensajes.

Algunos sistemas soportan un híbrido de los dos; editores publican mensajes de un tema mientras que los suscriptores se registran suscripciones basadas en el contenido a uno o más temas.

En muchos sistemas de publicación/suscripción, los editores envían mensajes a un intermediario de mensajes o eventos, y los suscriptores registran suscripciones con ese intermediario, dejando que él realice el filtrado. El intermediario normalmente realiza una función de almacenamiento y enruta los mensajes de los editores a los suscriptores. Además, el intermediario puede dar prioridad a los mensajes en una cola antes de enviarlos.

Los suscriptores pueden registrarse para mensajes específicos en tiempo de compilación, el tiempo de inicialización o de tiempo de ejecución. En los sistemas de interfaz gráfica de usuario, los suscriptores pueden codificarse para manejar los comandos de usuario (por ejemplo, hacer clic en un

botón), que corresponde a la construcción del registro del tiempo. Algunos marcos de referencia y productos de software utilizan los archivos de configuración XML para registrar suscriptores. Estos archivos de configuración se leen en tiempo de inicialización. La alternativa más sofisticada es cuando se pueden agregar o quitar en tiempo de ejecución de los suscriptores. Se utiliza este último método, por ejemplo, en los disparadores de base de datos, listas de correo y RSS.

3.4. Canales de comunicación

Un canal de comunicación es el medio de transmisión por el que viajan los mensajes entre el emisor y el receptor. Estos canales pueden ser personales o masivos: los canales personales son aquellos en donde la comunicación es directa, un emisor se comunica directamente con el receptor el teléfono es un buen ejemplo de esto, ya que quieren quieren comunicarse lo hacen solo entre 2 puntos el del emisor y el del receptor. Los masivos por el contrario comunican varios receptores con el emisor esto podría ejemplificarse con lo que sucede con los canales de TV, donde un emisor bajo el mismo canal llega a varios receptores con el mismo mensaje. Con los mensajes masivos se optimizan las señales y procesamiento ya que por la misma señal corre el mismo mensaje para todos los receptores, si por el contrario el mensaje fuera punto a punto se necesita replicar el mensaje para cada uno de los receptores que quieran consumir el la información de emisor.

3.5. WordNet

Es una base de datos léxica del Idioma Inglés. Agrupa palabras en inglés en conjuntos de sinónimos llamados synsets, proporcionando definiciones cortas y generales, y almacena las relaciones semánticas entre los conjuntos de sinónimos. Su propósito es doble: producir una combinación de diccionario y una lista de sinónimos, habitualmente acompañada por otra lista de antónimos cuyo uso sea más intuitivo, y soportar análisis automático de texto y a aplicaciones de Inteligencia Artificial.

WordNet [Wordnet 2014] fue creado y es mantenido por el Cognitive Science Laboratory de la Universidad de Princeton. El desarrollo comenzó en 1985. A través de los años, el proyecto ha recibido financiamiento de agencias gubernamentales interesadas en la traducción automática.

La base de datos de WordNet contiene 155,287 palabras organizadas en 117,659 synsets para un total de 206,941 pares de palabras; en forma comprimida tiene 12 megabytes de tamaño.

WordNet distingue entre sustantivos, verbos, adjetivos y adverbios porque ellos siguen diferentes reglas gramaticales. Cada synsets contiene un grupo de palabras que son sinónimos o collocations (un collocation es una secuencia de palabras que unidas toman un significado específico); diferentes significados de una palabras están en distintos synsets.

La mayoría de los synsets están conectados a otros synsets mediante numerosas relaciones semánticas. Estas relaciones varían según el tipo de palabra, y se incluyen:

- Sustantivos
  - *hypernyms*: *Y* es un hypernym de *X* si cada *X* es un (del tipo de) *Y* (canino es un

hypernym de perro)

- *hyponyms*: *Y* es un hyponym de *X* si cada *Y* es un (del tipo de) *X* (perro es un hyponym de canino)
- *coordinate terms*: *Y* es un coordinate term de *X* si *X* y *Y* comparten un hypernym (lobo es un coordinate term de perro y perro es un coordinate term de lobo)
- *holonym*: *Y* es un holonym de *X* si *X* es parte de *Y* (edificio es un holonym de ventana)
- *meronym*:*Y* es un meronym de *X* si *Y* es una parte de *X* (ventana es una parte de edificio)

- Verbos
  - *hypernym*: el verbo *Y* es un hpernym del verbo *X* si la actividad *X* es un (del tipo de) *Y* (percibir es un hypernym de escuchar)
  - *troponym*: el verbo *Y* es un toponym del verbo *X* si la actividad *Y* está realizando *X* de alguna manera (susurrar es un troponym de hablar)
  - *entailment*: el verbo *Y* es entailment de *X* si haciendo *X* se debe estar haciendo *Y* (dormir es un entailment de roncar)
  - *coordinate terms*: son los verbos que comparte un hypernym común (susurrar y gritar)
- Adjetivos
  - *sustantivos relacionados*
  - *similar a*
  - *participios de verbos*
- Adverbios
  - *La raíz de los adjetivos*

Mientras que las relaciones semánticas se aplican a todos los miembros de un synset porque comparten significado, no todos son mutuamente sinónimos, las palabras también pueden estar conectadas a otras palabras a través de relaciones léxicas, incluyendo antónimos.

WordNet también provee el *polysemy count* de una palabra: el número de synsets que contienen la palabra. Si una palabra participa en varios synsets (es decir tiene varios significados) es común que unos significados sean más comunes que otros. WordNet los califica por la *frecuency score*: en el cual varios textos de ejemplo tienen todas las palabras semánticamente etiquetadas en el correspondiente synset, y después a través de un contador se indica la frecuencia en la que una palabra aparece con un significado específico. Las funciones morfológicas del software distribuidas

con la base de datos tratan de deducir el lema o raíz de una palabra de la entrada del usuario; sólo la raíz de cada palabra es almacenada en la base de datos.

Los sustantivos y los verbos están organizados en jerarquías. Por ejemplo, el primer significado dela palabra “perro” tendrá la siguiente jerarquía hypernym; las palabras en el mismo nivel son sinónimos unas de otras. Cada conjunto de sinónimos (synset), tiene un único indexador y comparte sus propiedades, como una definición gloss (o diccionario).

dog, domestic dog, Canis familiaris
=> canine, canid
=> carnivore
=> placental, placental mammal, eutherian, eutherian mammal
=> mammal
=> vertebrate, craniate
=> chordate
=> animal, animate being, beast, brute, creature, fauna
=> ...

En el nivel superior, estas jerarquías están organizadas en tipos básicos, 25 grupos primitivos de sustantivos y 15 de verbos. Estos grupos primitivos están conectados a un nodo raíz abstracto, que ha sido asumido desde algún tiempo por varias aplicaciones que usan WordNet. En el caso de los adjetivos, la organización es diferente, la jerarquía y el concepto involucrado con los lexicographic files, no se aplican de la misma forma que lo hacen para los sustantivos y los verbos. El grafo de sustantivos es mucho más profundo que otras partes del lenguaje. Los verbos tienen una estructura más densa y los adjetivos están organizados en varios clústeres diferentes. Los adverbios están organizados en términos de los adjetivos de los que se derivan, y por tanto heredan su estructura de la de los adjetivos.

La relación hypernym/hyponym entre los synsets puede ser interpretada como una relación de especialización entre categorías conceptuales. En otras palabras, WordNet puede ser interpretado y usado como una ontología en informática. Sin embargo para utilizarlo como una ontología debe ser corregido antes de ser usado, ya que contiene cientos de inconsistencias semánticas básicas como la existencia de especializaciones comunes para categorías exclusivas y redundancias en la jerarquía de especialización. Por tanto, transformar WordNet en una ontología léxica usable para la representación del conocimiento debe normalmente involucrar la distinción de las relaciones de especialización en subtipos de e instancias de relaciones, y asociar identificadores únicos e intuitivos para cada categoría. Aunque esas correcciones y transformaciones han sido desarrolladas y documentadas como parte de WordNet 1.7, la mayoría de los proyectos alegan rehusar WordNet para aplicaciones basadas en conocimientos (típicamente aplicaciones para la recuperación de información orientada al conocimiento) simplemente usándolo directamente. WordNet también ha sido convertido a una especificación formal, por medio de una metodología híbrida bottom-up top-down para extraer automáticamente relaciones de asociación de WordNet, e interpretar estas asociaciones en términos de conjuntos de relaciones conceptuales, formalmente definido en el *DOLCE foundational ontology*.

A diferencia de otros diccionarios, WordNet no incluye información sobre la etimología, pronunciación y la forma de los verbos irregulares y contiene sólo información limitada sobre uso.

A pesar de que WordNet contiene un rango suficientemente amplio de palabras comunes, no cubre el vocabulario de un dominio específico. Como está diseñada en primer lugar para actuar como capa subyacente para diferentes aplicaciones, estas aplicaciones no pueden ser usadas en dominios específicos que nos son cubiertos por WordNet. WordNet es el lexicón computacional de Inglés comúnmente más usado para desambiguar el significado de las palabras (word sense disambiguation (WSD)), una tarea que tiene como objetivo asignar el significado más apropiado (i.e. synsets) a las palabras en contexto.

WordNet ha sido usado para diferentes y numerosos propósitos en sistemas de información, que incluyen desambiguación del significado de palabras, recuperación de información, clasificación automática de texto, resumen automático de texto, traducción automática e incluso generación de crucigramas.

La empresa Simpli construyó un buscador para Internet que utilizaba una base de conocimientos basada principalmente en WordNet para desambiguar y expandir palabras claves y synsets para ayudar en la recuperación de información online. Otro prominente ejemplo del uso de WordNet es determinar la similitud entre palabras. Se han propuesto varios algoritmos, que incluyen considerar la distancia entre las categorías conceptuales de las palabras, así como también considerar la estructura jerárquica de la ontología WordNet.

El término ontología en informática hace referencia a la formulación de un exhaustivo y riguroso esquema conceptual dentro de uno o varios dominios dados; con la finalidad de facilitar la comunicación y el intercambio de información entre diferentes sistemas y entidades.

Un uso común tecnológico actual del concepto de ontología, en este sentido semántico, lo encontramos en la inteligencia artificial y la representación del conocimiento.

Los programas informáticos pueden utilizar así este punto de vista de la ontología para una variedad de propósitos, incluyendo el razonamiento inductivo, la clasificación, y una variedad de técnicas de resolución de problemas.

Dichas ontologías son valorables comercialmente, creándose competencia para definirlas. Peter Murray-Rust se ha quejado de que esto conduce a *"una guerra semántica y ontológica debido a la competencia entre estándares"*. Por consiguiente, cualquier estándar de ontología fundacional es posible que sea contestado por los agentes políticos o comerciales, cada uno con su propia idea de 'lo que existe' (en el sentido filosófico de ontología).

Una variación ha sido propuesta recientemente, véase el sitio de MathWorld (http://mathworld.wolfram.com), cuyo autor propugna que el universo se modela mejor en los términos de los programas informáticos (computacionales) que con los términos matemáticos convencionales.

Capítulo 4: Una nueva solución

La combinación de herramientas ayudan a solucionar nuevos problemas derivados del nuevo paradigma de IoT. Problemas tales como encontrar el hub de la red, mantener conexiones bidireccionales sin polling, envío de mensajes desde el hub hacia los dispositivos conectados o mensajes enviados sobre canales tipificados… haciendo uso de paquetes de datos UDP, la nueva tecnología de webSockets, y el uso de un diccionario léxico como wordNet, se pueden solucionar los nuevos problemas que traen aparejados nuevos paradigmas como internet de las cosas.

4.1. El modelo de solución

Los dispositivos cuando se conectan a la red utilizan paquetes UDP para conseguir la dirección del hub y poder establecer una conexión mediante webSockets con el hub, una vez lograda esa conexión se envía qué tipo de tópico le interesa y en el hub le asigna el canal correspondiente a ese tópico. El tópico enviado es una palabra en lenguaje cotidiano para flexibilizar las conexiones es por esto que se necesita la herramienta de wordNet para desambiguar los sinónimos que deben retornar el mismo canal.

4.2. Encontrar el hub en la red

IoT es la red de objetos físicos cotidianos que están conectados a internet y son capaces de comunicarse y generar información para otros objetos. Pero para poder enviar datos o recibirlos, no basta que estén conectados a internet sino también conocer al resto de los objetos que formarán la red de objetos intercomunicados.

Cuando un objeto se conecta a una red necesita un mecanismo para transmitir sus datos al resto de los objetos o bien para recibir los datos del resto de los objetos que conforman la red de objetos. Estos objetos tienen muy poco poder de procesamiento ya que son objetos cotidianos y no grandes computadoras, es por esto que se debe minimizar al máximo las tareas de comunicación y delegar estas estrategias al hub que está coordinando la red y sus objetos. Un objeto para poder delegar al hub los problemas de comunicación con el resto de los objetos interesados en la información proporcionada primero debe encontrarlo en la LAN.

Encontrar el hub de la red no es una tarea sencilla ya que el hub no tiene una dirección fija y aunque la tuviese los objetos pueden cambiar de red constantemente y necesitan todo el tiempo conocer la ubicación del nuevo hub que coordina la red [W3 Discovery API]. Una forma de resolver esto es lograr que el objeto envie un mensaje a todos los objetos de la red, que incluiría al hub, y esperar que el hub reconozca el mensaje y retorne su dirección. Esto puede lograrse con paquetes UDP ya que pueden enviarse a todos los objetos de la red; estos paquetes tendrán información además de la propia del emisor , sobre que se esta requiriendo de los destinatarios. Si el destinatario es un hub y entiende el mensaje dentro del paquete UDP entonces contesta al emisor con su propia dirección ip.

Actualmente hay varios mecanismos para encontrar servicios en la Lan. Los más conocidos son: SSDP, UPnP, mDNS, DIAL.

4.2.1 SSDP (Protocolo Simple de Descubrimiento de Servicios)

El Protocolo Simple de Descubrimiento de Servicios (Simple Service Discovery Protocol) [SSDP 2014] es un protocolo que sirve para la búsqueda de dispositivos UPnP en una red. Utiliza UDP en unicast o multicast en el puerto 1900 para anunciar los servicios de un dispositivo. Solo la información más importante acerca el dispositivo y el servicio ofrecido está contenido en los mensajes intercambiados.

El protocolo se utiliza también para buscar ciertos servicios en la red. Un Punto de Control UPnP (UPnP Control Point) manda un mensaje a todos los dispositivos en la red por el mismo canal, igual que los anuncios. Si alguno de ellos ofrece el servicio deseado le contesta con un mensaje normal HTTP con el código '200 OK'. Este mensaje se manda en UDP unicast.

Estructura

Un paquete SSDP consiste en un HTTP-Request con la instrucción 'NOTIFY' para anunciar o con 'M-SEARCH' para buscar un servicio. El HTTP-Body queda vacío. La cabecera contiene atributos específicos de UPnP.

- NTS (Notification Sub Type) puede tener el valor "ssdp:alive" para registrarse o "ssdp:byebye" para anular el registro del dispositivo.
- USN (Unique Service Name) contiene una identificación única.
- LOCATION contiene la URL de la descripción

4.2.2. UPnP

Universal Plug and Play (UPnP) es un conjunto de protocolos de comunicación que permite a periféricos en red, como ordenadores personales, impresoras, pasarelas de Internet, puntos de acceso Wi-Fi y dispositivos móviles, descubrir de manera transparente la presencia de otros dispositivos en la red y establecer servicios de red de comunicación, compartición de datos y entretenimiento. UPnP está diseñado principalmente para redes de hogar sin dispositivos del ámbito empresarial [UPnP 2014].

Es una iniciativa de la industria informática para permitir una conectividad simple y robusta entre los dispositivos autónomos y ordenadores personales de diferentes fabricantes.

El concepto de UPnP es una extensión de Plug and play, una tecnología para conectar dispositivos de manera directa y sin necesidad de configuración a un ordenador, aunque UPnP no está relacionada directamente con la tecnología plug-and-play. Los dispositivos UPnP son "plug-and-play" en el sentido de que una vez conectados a una red son capaces de establecer de manera automática comunicaciones con otros dispositivos.

Protocolo UPnP (universal plug and play)

Direccionamiento: La base de UPnP es el direccionamiento IP: cada dispositivo debe

implementar un cliente DHCP (protocolo de configuración dinámica de host) y buscará un servidor DHCP en cuanto se conecte por primera vez a la red. Si no hay ningún servidor DHCP disponible, el dispositivo debe asignarse a sí mismo una dirección. El proceso por el cual un dispositivo UPnP se auto-asigna una dirección se denomina AutoIP.

Descubrimiento: Una vez que un dispositivo ha establecido una dirección IP, el siguiente paso en UPnP es el descubrimiento. El protocolo de descubrimiento de UPnP se denomina Simple Service Discovery Protocol (SSDP). SSDP permite a los dispositivos que acaban de añadirse a una red anunciar sus servicios a los puntos de control presentes en la red. Asimismo, cuando se añade un punto de control a la red, SSDP le permite buscar los dispositivos que le interese controlar. El intercambio fundamental en ambos casos es un mensaje de descubrimiento que contiene datos básicos del dispositivo o uno de sus servicios, por ejemplo, su tipo, identificador y un enlace una URL en la que obtener información más detallada.

Descripción: Después de que un punto de control haya descubierto un dispositivo, todavía dispone de muy poca información acerca de él. El punto de control debe obtener la descripción del dispositivo desde la URL proporcionada por el dispositivo en el mensaje de descubrimiento para conocer mejor sus capacidades y poder interactuar con él. La descripción de un dispositivo se codifica en XML e incluye información específica del fabricante como nombre de modelo y número, número de serie, nombre de fabricante, URLs a sitios web específicos de fabricante, etc. La descripción también incluye una lista de dispositivos o servicios embebidos, así como URLs de control, manejo de eventos y presentación. Para cada servicio, la descripción incluye una lista de los comandos, o acciones, a las que responderá el servicio, y parámetros, o argumentos, de cada acción; la descripción de un servicio también incluye una lista de variables; estas variables modelan el estado del servicio en tiempo de ejecución, y se describen en términos de su tipo de datos, rango y las características de sus eventos.

Control: Al obtener la descripción del dispositivo, el punto de control puede enviar acciones a los servicios de un dispositivo. Para ello, el punto de control envía un mensaje de control apropiado a la URL de control del servicio (proporcionada en la descripción del dispositivo). Los mensajes de control también se codifican en XML mediante Simple Object Access Protocol (SOAP). El servicio responderá con un mensaje de control con los resultados de la acción de forma similar a una llamada a una función. Los efectos de la acción, en caso de existir, se modelan mediante cambios en las variables que describen el estado del servicio.

Notificación de eventos: Una capacidad adicional de UPnP es la notificación de eventos o eventing. El protocolo de notificación de eventos definido en la arquitectura UPnP se conoce como General Event Notification Architecture (GENA). La descripción UPnP de un servicio incluye una lista de las acciones a las que el servicio responde y otra con las variables que modelan el estado del servicio en tiempo de ejecución. El servicio enviará actualizaciones cuando cambien dichas variables a cualquier punto de control que se haya suscrito para recibir dicha información. El servicio publica actualizaciones enviando mensajes codificados en XML de evento que contiene los nombre de una o más variables de estado y el valor actual de dichas variables. Cuando un punto de control se suscribe por primer vez se le envía un mensaje especial de eventos; que contiene el nombre y los valores de todas las variables que generan eventos y permite al suscriptor conocer el estado actual del servicio. La notificación de eventos UPnP se ha diseñado para mantener a todos los puntos de control informados por igual sobre los efectos de cualquier acción, de este modo se permite soportar escenarios con múltiples puntos de control. Por tanto, los mensajes de eventos se envían a todos los suscriptores, los suscriptores reciben mensajes para todas las variables que han cambiado a las

que se han suscrito y los mensajes se envían sin importar el motivo que modificó la variable de estado (tanto en respuesta a una petición, como por el cambio del estado interno del servicio).

***Presentación**:* El último paso en UPnP es la presentación. Si un dispositivo tiene una URL de presentación, entonces el punto de control podrá obtener una página desde dicha URL, mostrarla en un navegador y, dependiendo de las características de la página, permitirá al usuario controlar el dispositivo y/o consultar su estado. El grado de control que se puede obtener depende en gran medida del dispositivo y de la interactividad presente en la interfaz de presentación.

4.2.3. Multicast DNS (mDNS)

El sistema de nombre de dominio multicast (mDNS) resuelve los nombre de host a direcciones IP dentro de pequeñas networks que no incluyan un servidor de nombres local. Es un servicio de configuración cero, utilizando esencialmente las mismas interfaces de programación, formatos de paquetes y la semántica de funcionamiento que el unicast Sistema de nombres de dominio (DNS). Aunque está diseñado para ser independiente capaz, puede trabajar en conjunto con los servidores DNS de unidifusión [mDNS 2014].

El protocolo mDNS se publicó como RFC 6762, utiliza Datagram Protocol (UDP) IP multicast de usuario, y es implementado por Apple Bonjour y servicios Linux NSS-mDNS.

4.2.4. DIAL (DIscovery And Launch)

El protocolo DIAL [DIAL 2015] tiene dos componentes: DIAL Discovery Service y el Servicio REST DIAL.

DIAL Discovery Service permite a un dispositivo cliente DIAL descubrir servidores DIAL en su red local y obtener acceso a los servicios DIAL REST de estos dispositivos.

El servicio DIAL REST permite a un cliente DIAL consultar, lanzar, y opcionalmente frenar aplicaciones en un dispositivo de servidor DIAL.

EL descubrimiento de servicios DIAL se logra utilizando una nueva búsqueda de destino dentro del protocolo SSDP definido por UPnP y un encabezado adicional en la respuesta a una petición HTTP para el dispositivo UPnP.

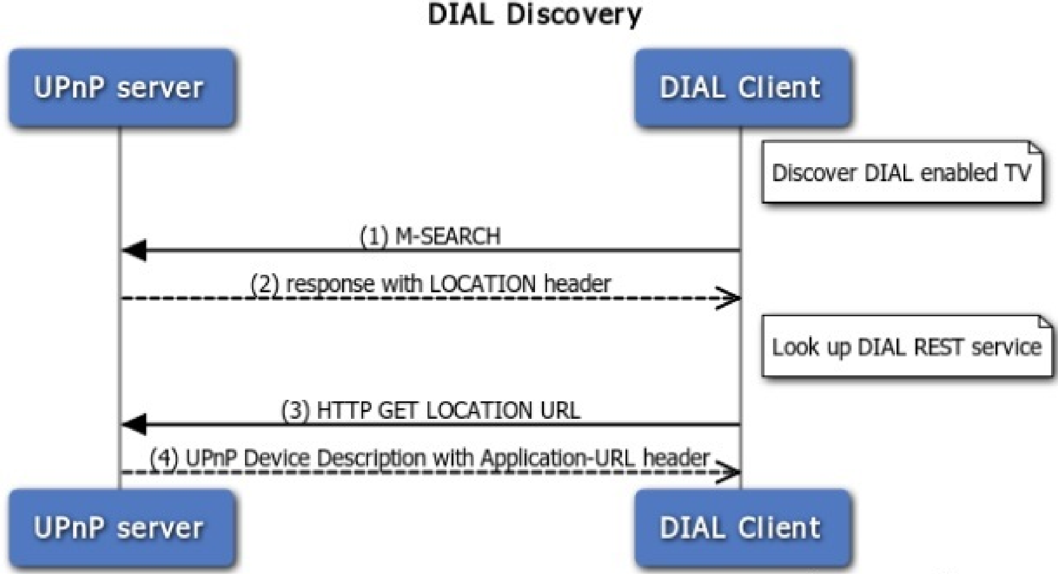

Figura 4.1.: Protocolo DIAL
Gráfico que ejemplifica el protocolo de descubrimiento de DIAL basado en SSDP

4.2.5. Protocolo de descubrimiento para WebSockets

Teniendo en cuenta la nueva naturaleza del problema de tener que descubrir un servidor de WebSockets dentro de la LAN y evaluando el resto de los servicios de descubrimiento que se están utilizando actualmente se optó por usar un protocolo simplificado similar a SSDP, ya que lo que necesita un cliente WebSocket para comunicarse con el servidor WebSocket correspondiente es solamente su dirección.

Funcionamiento

Cliente: Un objeto WebSocket envía un paquete UDP de broadcast a la red para detectar si en ella existe algún servidor de WebSockets. Este paquete contiene en su body un objeto json con la información de la operación a realizar, en este caso la búsqueda de un servidor de WebSockets:

{ DATA : “Find WebSocket Server” }

Una vez enviado el paquete se queda a la espera de la respuesta.

Servidor: Si existe en la LAN un servidor de WebSockets que está atendiendo paquetes UDP, al recibir uno con un objeto json como el enviado por el cliente responde al pedido con la dirección donde atenderá los requerimiento de WebSocket, el paquete de respuesta también es un paquete UDP unidireccional con un objeto json con la información de la dirección:

{ DATA : "ws://127.0.0.1:8080/webSocketServer/websocket" }

Cuando el cliente recibe el paquete de dirección del server ya esta en condiciones de iniciar una coneccion a nivel de WebSockets.

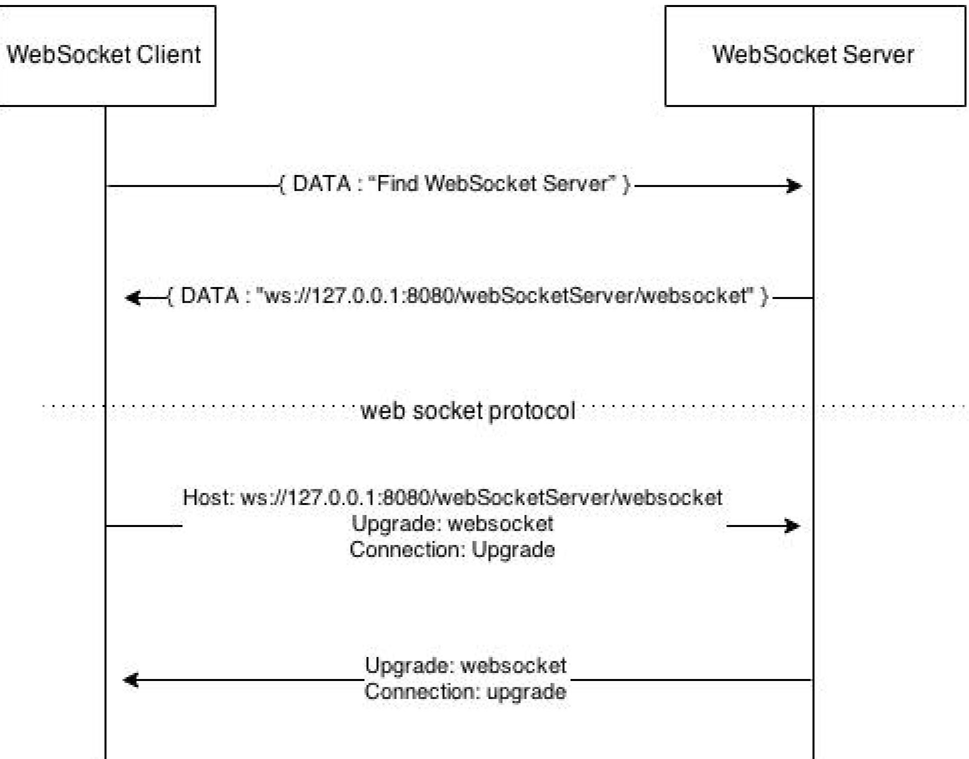

Figura 4.2.: Protocolo de Descubrimiento
Gráfico que ejemplifica el protocolo de descubrimiento

4.3. El canal de comunicación

Aunque los objetos cuenten con la ayuda del hub para poder comunicar la información a través de la red, no basta con esto ya que en una red plagada de miles de objetos que están enviando información constantemente es necesario algún mecanismo más eficiente para no saturar la red con miles de mensajes circulando por ésta. Por ésto y para realizar una comunicación más efectiva el hub maneja el concepto de canales de comunicación, donde por cada uno de estos canales circulan datos específicos relacionados a un tópico especial, por ejemplo por un canal podrían circular datos de temperatura ambiental, por otro datos de humedad ambiental, y por otro temperaturas de un producto.

Cada objeto cuando logra conectarse con el hub comunica en qué tipo de tópico esta interesado, estos tópicos son palabras que sirven para identificar que tipo de información viaja por ese canal, esto nos garantiza que los objetos sob

Como la comunicación debe realizarse de la manera más eficiente posible no van a estar mandando mensajes entre objetos directamente sino que se hará a través de un hub, quien coordinará los mensajes, canales y demás particularidades de la comunicación.

4.3.1. Canales definidos por palabras claves / tópicos

Se quiere llevar la comunicación entre dispositivos a la formas menos restrictivas y rígidas, para facilitar la comunicación, para esto el hub al tener que gestionar los canales y sus topics, utilizará los beneficios que provee WordNet, la base de datos que contiene las relaciones entre palabras y sus sinónimos.

Cuando un dispositivos se conecta al hub, envía una petición para la creación de un canal con un grupo de palabras claves que identifican el canal sobre el cual se enviará y/o recibirá información, para poder comunicarse con el resto de los objetos de la red.

Para la creación del canal el hub utilizará el grupo de palabras recibidas y buscará sus sinónimos en la base de WordNet, con ese resultado de sinónimos y el conjunto original de topics conforman las palabras claves del nuevo canal quien ahora tendrá un conjunto de palabras claves que lo identifican mucho más amplio que el enviado desde el objeto, ganando con esto que las comunicaciones entre dispositivos sean menos restrictivas .

Cuando los objetos luego de conectarse al hub hacen un pedido para la creación de un canal con las palabras claves pertinentes, el hub deberá comprobar antes de crear un nuevo canal si ya no existe alguno que matchee con alguna de las palabras claves enviadas previamente, de encontrarse se retorna ese canal en caso contrario se crea uno nuevo como ya se comentó previamente.

4.4. WebSockets para la comunicación a través de canales

Una vez que los dispositivos se conectan al hub y obtienen el canal de comunicación, están en condiciones de recibir o enviar información a través de este canal.

Un dispositivo que genera información (sensor de temperatura, de humedad, de presión, etc) una vez que sensa un dato, debe enviar el dato al hub, para que éste lo reenvía al resto de los dispositivos interesados. Al enviar información al hub utilizando el webSocket, especificando sobre que canal debe enviar la los datos, cada objeto se independiza de los problemas y la coordinación del envío a múltiples objetos ya que esto queda resuelto del lado del hub y los canales taggeados. El problema de esto es la comunicación desde el hub hacia los clientes. Aquí es donde se aprovechan los beneficios de webSockets ya que precisamente lo que proveen es la bidireccionalidad de las comunicaciones.

WebSockets es la evolución de ajax polling y de long polling, ya que la comunicación puede iniciarse desde el servidor hacia el cliente cosa que no puede implementarse limpiamente con *ajax* o *long polling*. Si bien hay técnicas para simular la comunicación desde el servidor estas consumen demasiados recursos.

Ajax: está tecnología simula la entrega de mensajes desde el server al cliente consultando

permanentemente al servidor si tiene alguna dato que deba enviarle, si lo tiene, este dato viaja en la respuesta [Ajax 2014].

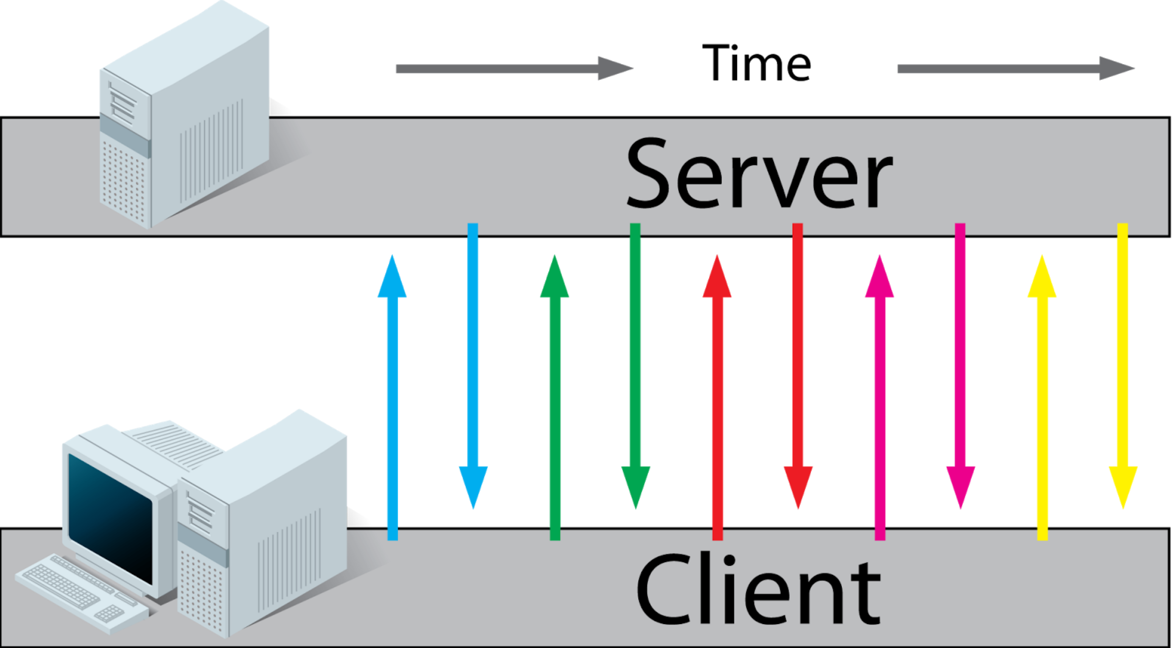

Figura 4.3.: Comunicación servidor-cliente por ajax
Gráfico que ejemplifica las conexiones necesarias para implementar comunicación servidor cliente con ajax.

Long Polling: está tecnología simula la entrega de mensajes desde el server al cliente estableciendo una conexión al servidor y dejándola abierta hasta que el servidor tenga información para enviarle, cuando el cliente recibe el dato cierra esa conexión y abre una nueva para el proximo envio de informacion [Long Polling 2014].

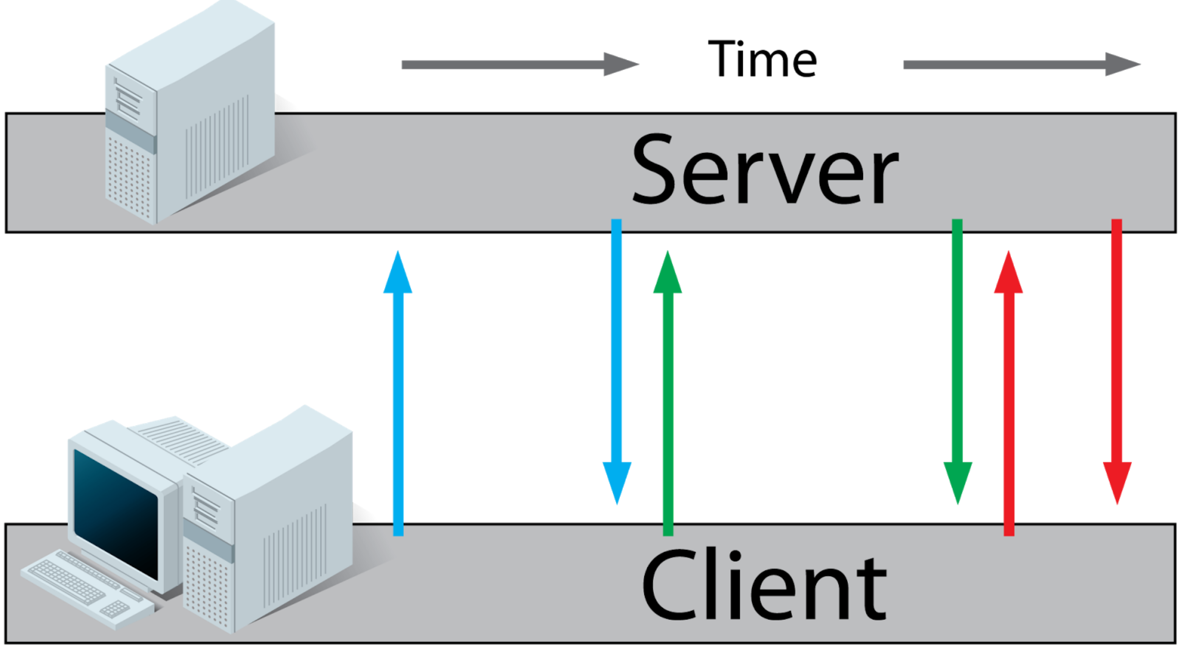

Figura 4.4.: Comunicación servidor-cliente por long polling
Gráfico que ejemplifica las conexiones necesarias para implementar comunicación servidor cliente con long polling.

WebSockets: La especificación WebSocket define un API que establece conexiones "socket" entre un cliente y un servidor. Existe una conexión persistente entre el cliente y el servidor, y ambas partes pueden empezar a enviar datos en cualquier momento [WS 2014].

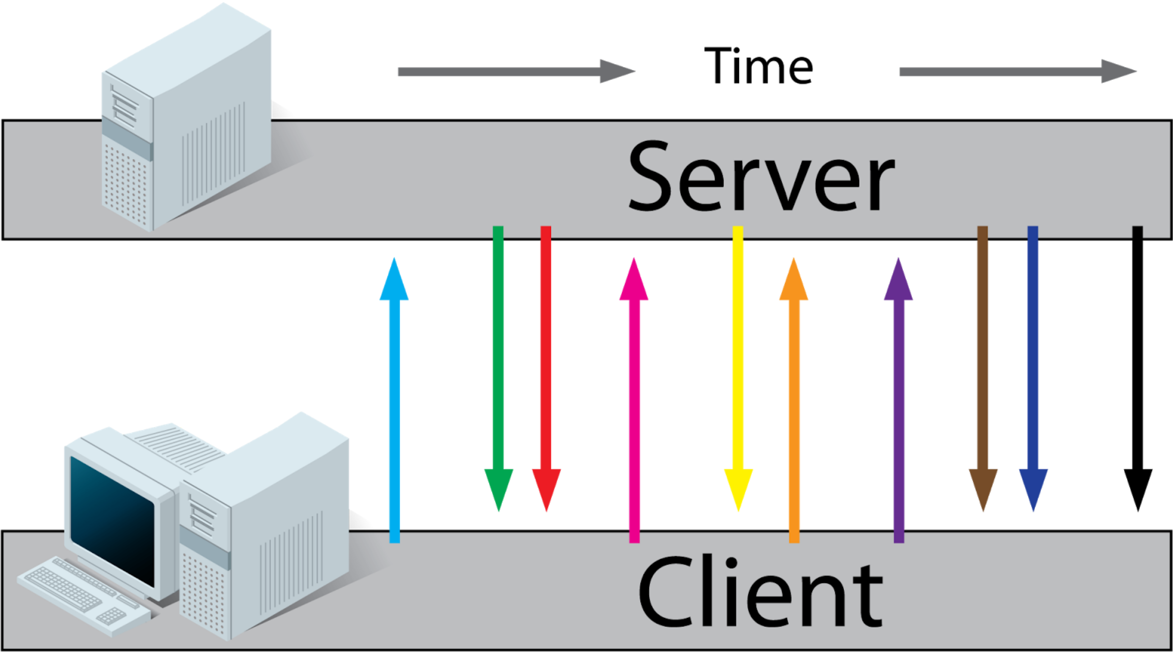

Figura 4.5.: Comunicación servidor-cliente por webSockets
Gráfico que ejemplifica las conexiones necesarias para implementar comunicación servidor cliente con long polling.

Capítulo 5: Implementación EIoT

Se desarrolló la solución EIoT (Easy Internet of Things) que consta de tres tipos de aplicaciones, un servidor web desarrollado en Java, quien cumple el rol de hub atendiendo y gestionando los requerimiento de los dispositivos y los canales de comunicación, una aplicación android quien actúa de dispositivo inteligente y una aplicación web que se usa de dashboard para ver en tiempo real el envío de mensajes sobre los canales.

Para el servidor se utilizó Java 7, sobre un servidor apache tomcat 7 que tiene soporte para websockets desde su versión 7.0.27 [Tomcat 2015].

Para la representación del objeto inteligente se utilizó una aplicación Android nativa [Android 2014].

Para el tablero de control, se desarrolló una aplicación web html5 con soporte para webSockets [HTML5 2014].

5.1. Buscar el hub en la red.

La aplicación para el objeto cliente fue desarrollada en Android utilizando componentes nativos del sistema.

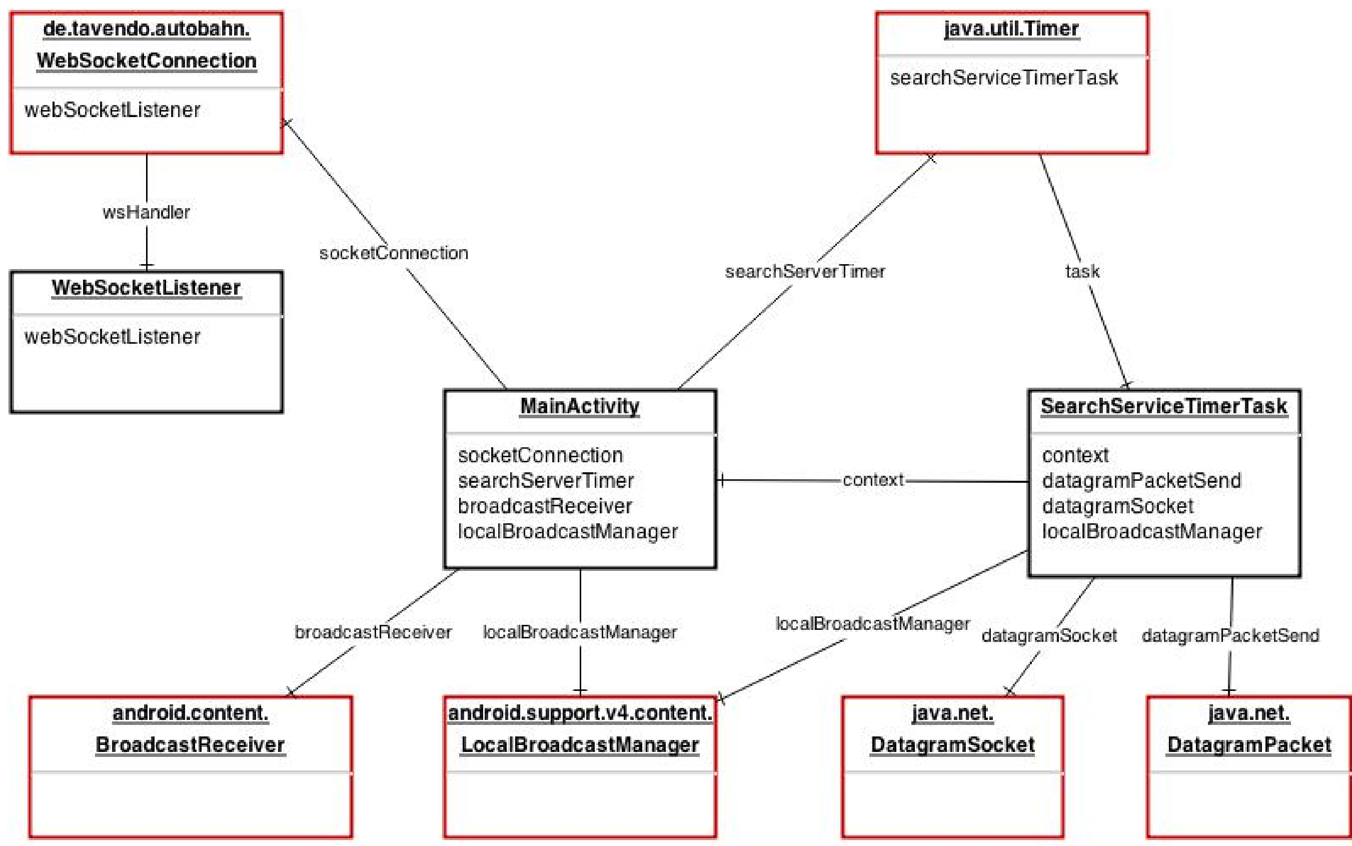

Figura 5.1: Diagrama de componentes de la aplicación cliente
Diagrama de componentes de un objeto inteligente desarrollado en android ([1])

El objeto para formar parte de la red de objetos conectados, necesita localizar la dirección IP del hub que maneja los objetos inteligentes de la red y los canales de comunicación para el envío y recepción de mensajes.

Para poder obtener la dirección del hub, se necesita de algún mecanismo de envío de paquetes que abarque a todos los componentes de la red, así eventualmente puede llegar al server y obtener de él su dirección. Para esto, al iniciar se instancia un objeto Timer para ejecutar cada 10 segundos el método run de la clase SearchServiceTimerTask, encargada de enviar paquetes UDP a la red para poder llegar al hub.

Cuando se crea la instancia del objeto SearchServiceTimerTask, éste instancia su variable datagramSocket que es un objeto DatagramSocket [DatagramSocket 2010] . Ésta proporciona acceso a un socket UDP, lo que permite que los paquetes UDP puedan ser enviados y recibidos a la red. El mismo DatagramSocket puede ser usado para recibir los paquetes tanto como para enviarlos. Como cada DatagramSocket se une a un puerto en la máquina local, el cual es usado para dirigir a los paquetes, instanciamos al datagramSocket con el puerto 5558.

A través del datagramSocket se quiere enviar un paquete UDP para esto se usa la clase

[1] https://www.draw.io/#G0B0NZSLb7uD2rT1R0WElhb1Rjak0

DatagramPacket [DatagramPacket 2010] que representa un paquete de datos destinados a la transmisión mediante el uso de UDP. Los paquetes son contenedores de una pequeña secuencia de bytes, e incluye información de direccionamiento, como una dirección IP y un puerto.

En los bytes del mensaje del datagramPacket enviamos un objeto Json:

{ DATA : "Find WebSocket Server" }

Ya que este es el mensaje que esta esperando el servidor para retornar su dirección IP.

Una vez que el paquete fue enviado a la red, el datagramSocket queda bloqueado a la espera de una respuesta del hub.

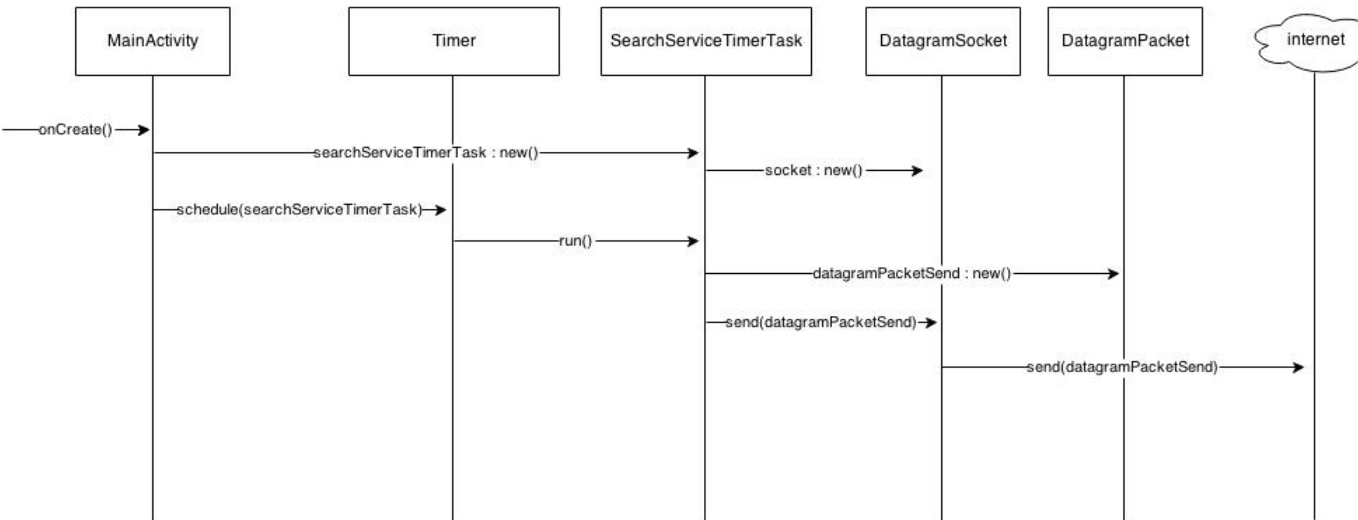

Figura 5.2: Diagrama de interacción de la aplicación cliente
Diagrama de interacción del cliente para encontrar el hub en la red ([2]).

5.2. El hub.

El hub es quien maneja los canales de comunicación entre los objetos en la red, necesita previamente establecer con ellos una conexión, para ello cuando inicia, queda a la espera de todos los los paquetes UDP que circulan en la red ya que los objetos enviaran este tipo de paquetes para poder localizar la ubicación del hub.

La aplicación servidor contiene una clase *ContextListener* con la anotación *@WebListener*, esta anotación hace que se ejecute al momento de inicializar la aplicación el método *contextInitialized* es allí donde se instancia la clase *UdpListener*, la encargada de estar escuchando los paquetes UDP que son enviados a la red, esta clase se ejecuta sobre un nuevo thread para no

[2] https://www.draw.io/#G0B0NZSLb7uD2rMUNPMWg4U0RYR0E

interrumpir la ejecución central del server que es atender la comunicación de los objetos ya registrados.

Cuando la clase *UdpListener* se ejecuta, instancia un *DatagramSocket* ya que este objeto proporciona acceso a un socket UDP, lo que permite que los paquetes UDP puedan ser recibidos de la red.

Una vez instanciado el DatagramSocket se ejecuta un while infinito para poder recibir todos los paquetes UDP enviados a la red, ya que esta es la forma que tienen los objetos para encontrar el hub. Dentro de este while lo primero que hace es estar a la espera de un paquete UDP esto lo hace el datagramSocket con el método *recive* ya que es bloqueante hasta que se reciba algún paquete UDP de la red.

Cuando se recibe un mensaje se inspecciona el contenido, primero para convertirlo en un objeto *JSON*, y luego para buscar en él la clave DATA, si la clave *DATA* existe y el valor es "Find WebSocket Server" entonces se crea un nuevo datagramPacket, para responder al objeto que envió el mensaje de solicitud de conexión, con la dirección del hub. Pero si la clave no existe o el valor no es el esperado, se descarta y se vuelve a esperar por un nuevo paquete de solicitud de conexión.

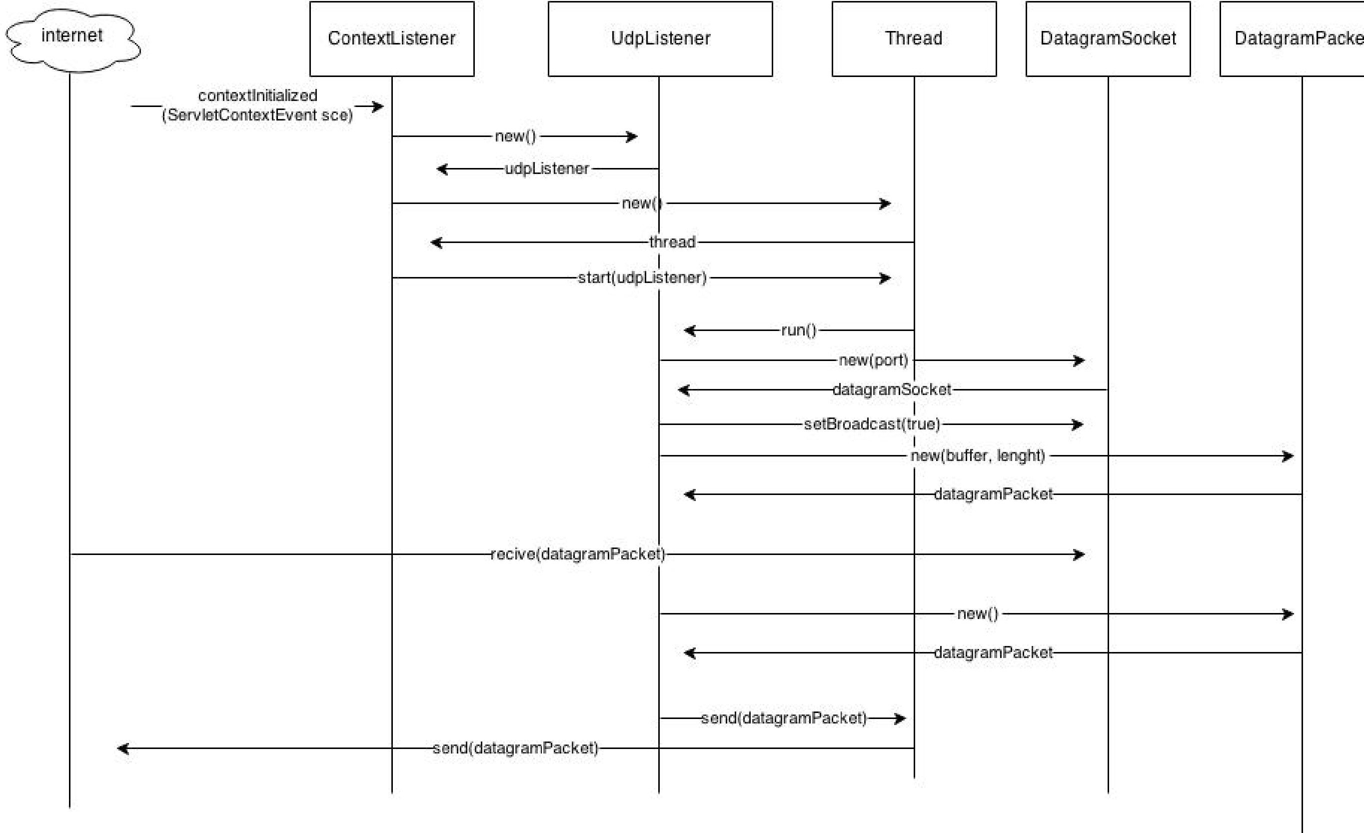

Figura 5.3: Diagrama de interacción del hub
Diagrama de interacción del hub para la respuesta al pedido de dirección ([3]).

5.3. Descubrimiento de servicios, el hub.

El objeto *SearchServiceTimerTask* envió un broadcast de un *datagramPacket* a la red para poder localizar al hub y así obtener su dirección ip, este recibió el paquete UDP y respondió con otro datagramPacket con su dirección.

El *SearchServiceTimerTask* estaba bloqueado esperando la respuesta ya que luego de enviar el datagramPacket, ejecuta el método bloqueante *recive* del datagramSocket.

Cuando el paquete de respuesta del hub llega, reactiva el *recive* que estaba bloqueando la ejecución, se crea un *Intent* con la información obtenida del server, en particular la dirección IP y se envía en un broadcast a traves del *LocalBroadcastManager.* Con la información del broadcast el objeto está listo para comunicarse con el hub.

[3] https://www.draw.io/#G0B0NZSLb7uD2rbEJnVTVhdTVROFU

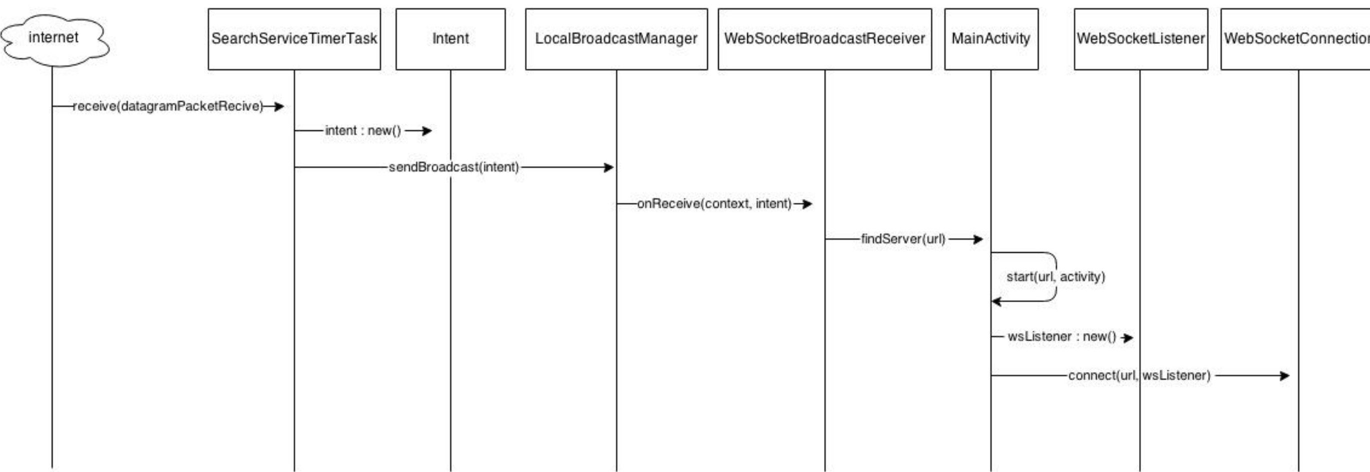

Figura 5.4: Diagrama de interacción del cliente: respuesta del hub
Diagrama de interacción del cliente ante la respuesta del hub al pedido de dirección ([4]).

5.4. Creación de canales, solicitud al hub.

Una vez que el objeto obtuvo la dirección del hub, ya puede empezar la comunicación.

Cuando el mecanismo interno de broadcast del objeto comunica que el hub fue encontrado y comunica la dirección de este, entonces el método *foundServer* es invocado con la dirección del hub en este método se crea un *WebSocketListener* quién escuchará los mensajes provenientes del hub y con él se invoca al *connect* del *socketConnection*, cuando esa conexión está establecida el método *open* del *WebSocketListener* es invocado, este hace los cambios pertinentes en la vista comunicando que la conexión fue establecida y se le manda al hub un mensaje para informarle que tipo de canal es el que le interesa. Este interés está dado por un par de palabras en inglés a las que llamaremos *topics* que darán significado a los datos que se transmitirán por ese canal en particular.

El mensaje a enviar es en formato json con los siguientes campos:

```
{
        OPERATION: “CREATE_CHANNEL”,
        TOPICS: “hot”);
}
```

[4] https://www.draw.io/#G0B0NZSLb7uD2rVkptMWxwaURTTW8

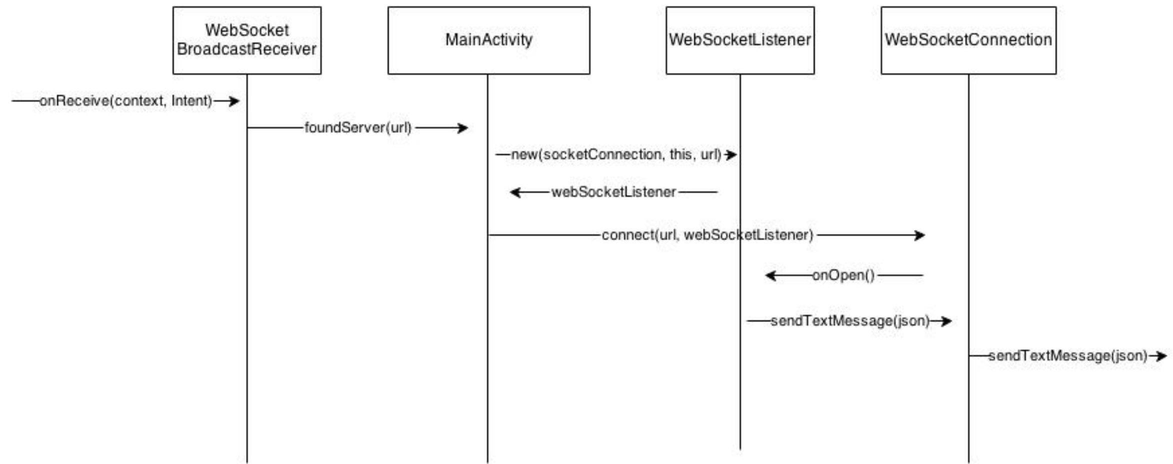

Figura 5.5: Diagrama de interacción del cliente
Diagrama de interacción del cliente ante la conexión establecida y el pedido de creación del canal al server ([5]).

5.5. Creación de canales, el hub.

El objeto *WebSocketListener* del hub está a la espera de recibir mensajes provenientes de los objetos de la red a través de websocket el método onTextMessage es el llamado cada vez que un mensaje proveniente de la red es recibido. Éste evalúa qué tipo de operación es la que el mensaje trae y en base a ello la ejecuta. Para el caso de la creación de canales la operación dentro del mensaje será “*CREATE_CHANNEL*” este mensaje trae además los topics con los que el canal debe crearse.

Cuando un mensaje de creación de canales es recibido se le indica al *channelManager* que cree un canal con los topics determinados, este antes de crear un nuevo canal con las palabras claves indicadas, verifica si ya no existe un canal con dichas palabras. Si entre todos los canales encuentra uno que coincida con las palabras enviadas, entonces este canal es enviado y no se creará uno nuevo. Por el contrario si ninguno de los canales existentes machea con las palabras enviadas entonces si se creara un nuevo canal.

5.5.1. Creación de canales, según palabras claves.

Si el *channelManager* necesita crear un canal según las palabras claves enviadas, no solo identifica el canal con esas palabras sino que también lo hará con las palabras que el *WordNet* le retorne como sinónimos de las palabras originales. Con este nuevo conjunto de palabras, las originales y las retornadas por *WordNet* es que quedará conformado el canal. Con esto encontrar el canal será más fácil para el resto de los objetos de la red ya que no sólo es identificable por un grupo de palabras sino también por todos sus sinónimos.

[5] https://www.draw.io/#G0B0NZSLb7uD2rem1wTXZ4N3Y2Ync

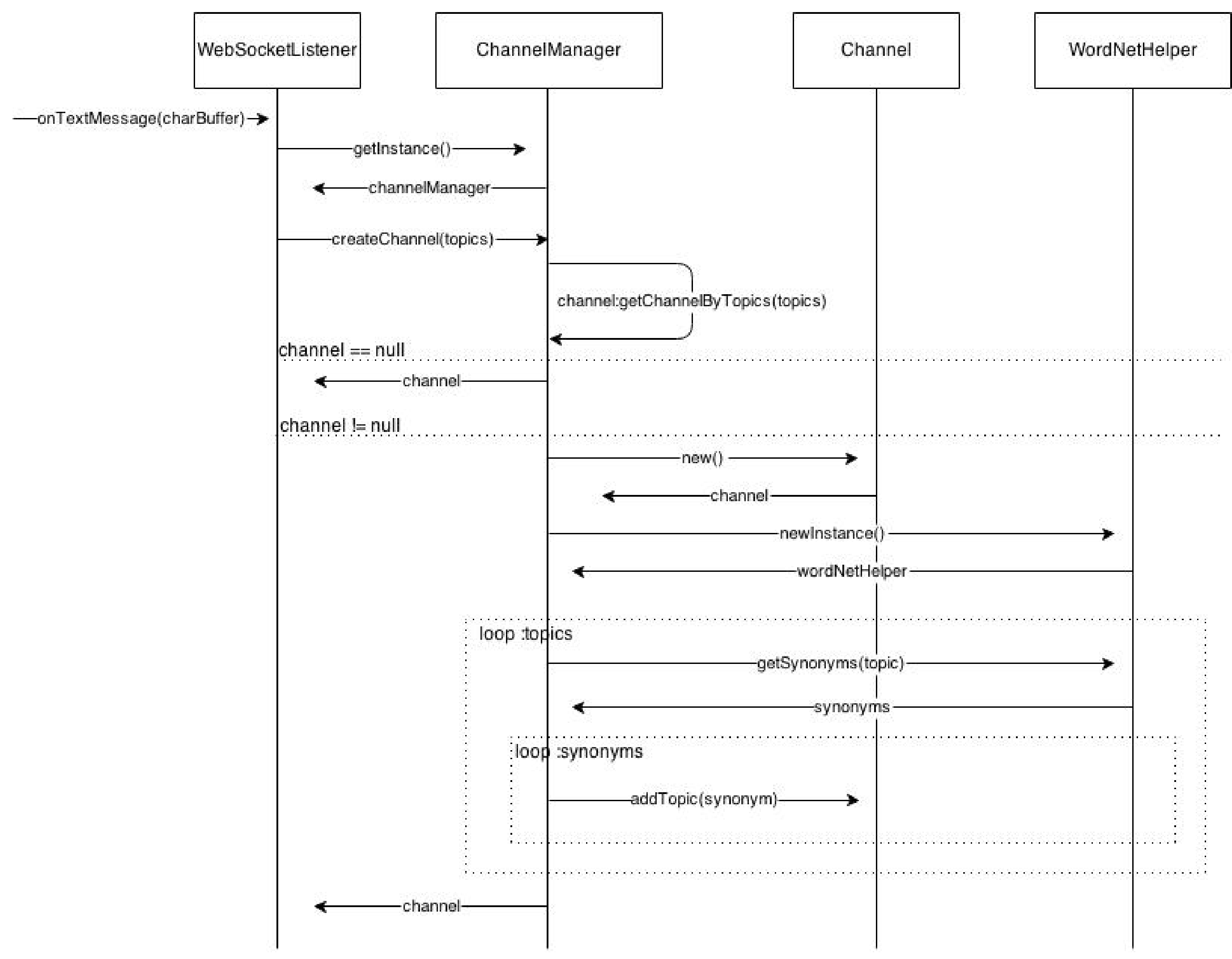

Figura 5.6: Creación de canales según palabras claves en el servidor
Diagrama de interacción del servidor para la creación de canales según palabras claves ([6]).

5.6. Envío de información al hub.

Una vez que la conexión con el hub está establecida y los canales de comunicación fueron creados, los objetos son capaces de comunicar a través del canal la información que necesiten enviar. Los objetos inician un *timer* para poder mandar la informacion necesaria cada cierta cantidad de tiempo.

Para comunicar la información a través del canal se necesita mandar al hub un objeto Json que contenga el tipo de operación que se está realizando, a que canal se desea transmitirlo y la

[6] https://www.draw.io/#G0B0NZSLb7uD2rMGJGMzJROVpCQ0E

información propiamente dicha.

El objeto Json podria quedar asi

```
{
        OPERATION: "SEND_DATA_TO_CHANNEL",
        CHANNEL_ID: "CHANNEL-1"
        DATA: "33"
}
```

Con esta información el hub es capaz de enviar a todos los objetos asociados al canal la información recibida.

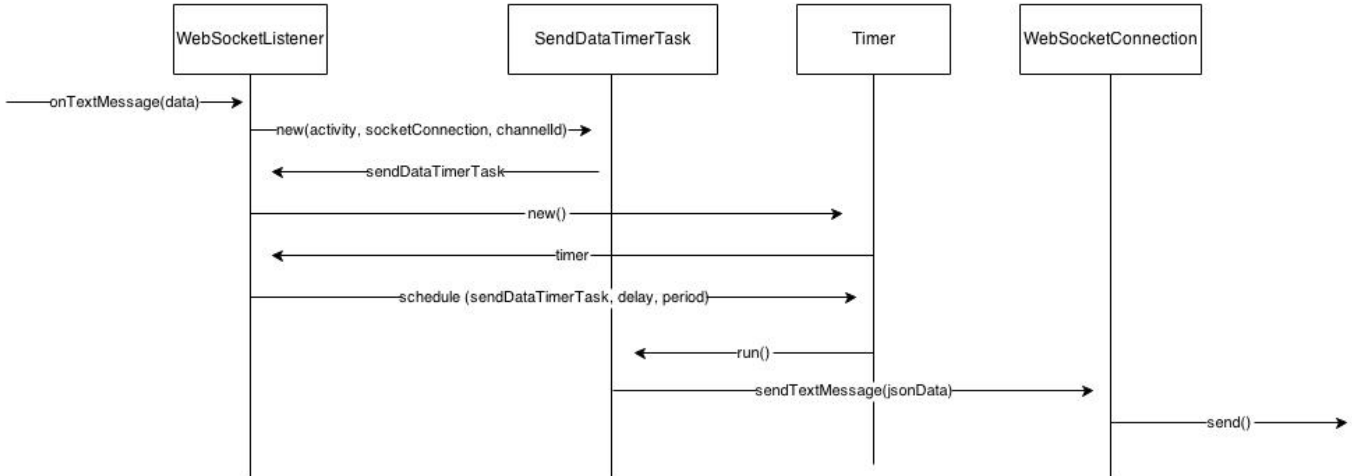

Figura 5.7: Diagrama de interacción: Envío de información por parte del cliente
Diagrama de interacción del cliente para enviar datos al canal correspondiente ([7]).

5.7. Comunicación a través de canales.

Cuando el *WebSocketListener* recibe un mensaje con la operación "SEND_DATA_TO_CHANNEL" está recibiendo información que debe ser enviado al canal especificado para que todos los objetos que estén suscritos al canal la reciban. En el mensaje esta la informacion del canal al cual mandar la información y la información en sí. Todos aquellos objetos que previamente se hayan suscrito al canal recibiran la informacion enviado por el objeto emisor.

[7] https://www.draw.io/#G0B0NZSLb7uD2rMIc0LXNyQ0d4LTQ

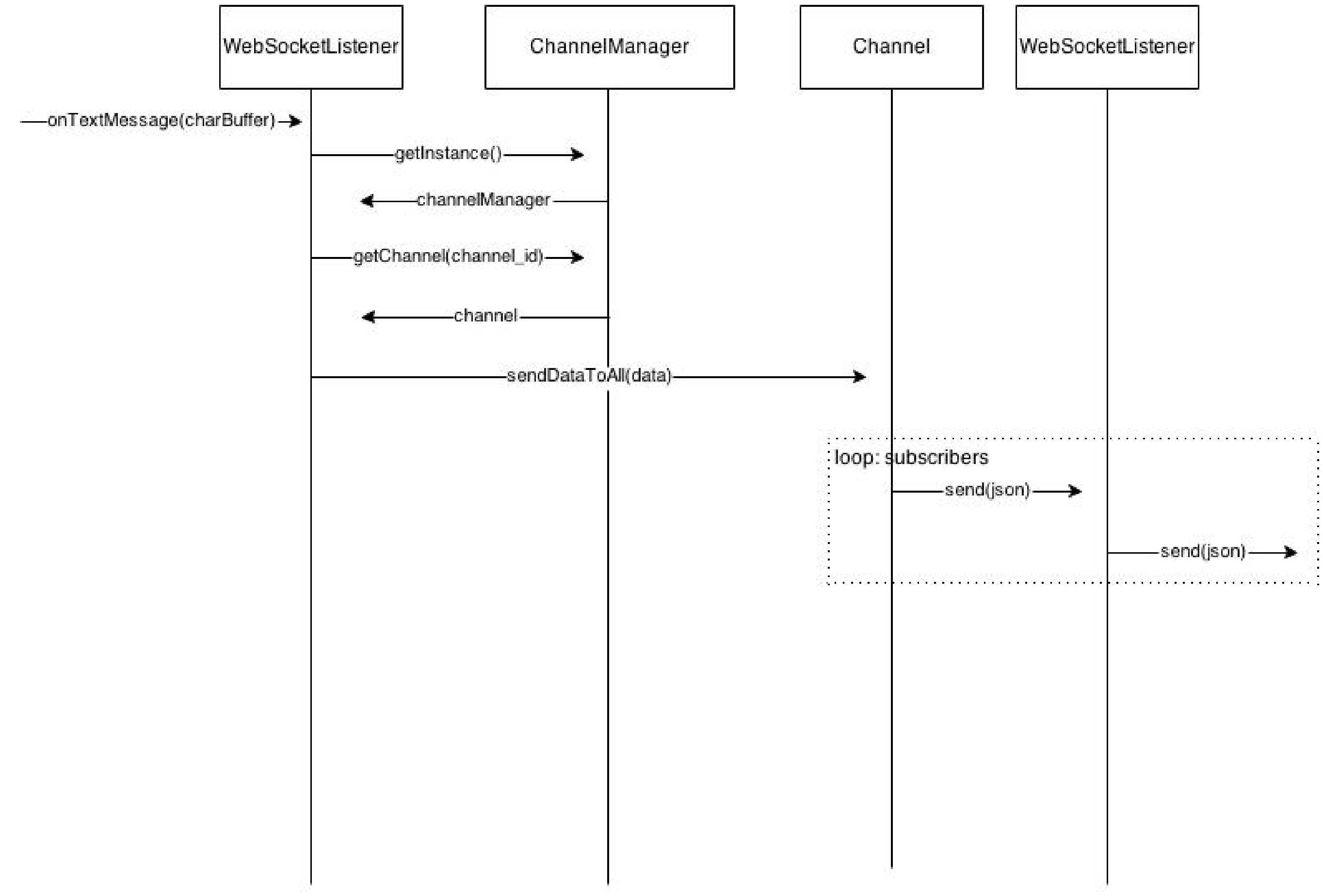

Figura 5.8: Recepción y envío de información a través de un canal
Diagrama de interacción del servidor para recibir y reenviar los datos al canal correspondiente ([8]).

5.8. Recepción de información a través de canales.

Los objetos que son consumidores de información y que ya se suscribieron al hub y a un canal de comunicación, están a la espera de que por ese canal llegue información de algún otro objeto productor. Es el *webSocket* quien provee la posibilidad de escuchar mensajes del hub y solo le llegaran los mensajes provenientes del canal al cual fue previamente suscrito.

5.9. Consola de control

Para poder visualizar todo el flujo de mensajes de la red y sus canales existe una aplicación web que se conecta al hub también mediante *websockets*. Este se suscribe a todos los canales creados y puede recibir y ver todos los canales y la información que se manda a través de ellos.

[8] https://www.draw.io/#G0B0NZSLb7uD2rMlc0LXNyQ0d4LTQ

Esta aplicación web se registra a todos los canales creados como receptor de la información, entonces recibe todo los canales y los mensajes que el hub esté manejando.

5.10. Arquitectura

EIoT está implementada sobre dos aplicaciones un servidor desarrollado en java y un cliente en html5.

5.10.1. Servidor

Esta dividido en 4 capas:

- Listeners: Aquí se encuentran todos los listeners de la aplicación, tanto los listeners web, como los udp, como los de webSockets.
- Manager: Aquí se encuentra el manager de canales quien tiene la lógica del manejo de canales y objetos de la aplicación
- Servlets: Aquí se encuentran los servlets de webSockets, responsable de atender los requerimientos web
- Model: Aquí se agrupan los objetos del modelo, en este caso los canales de comunicación y sus objetos inteligentes.

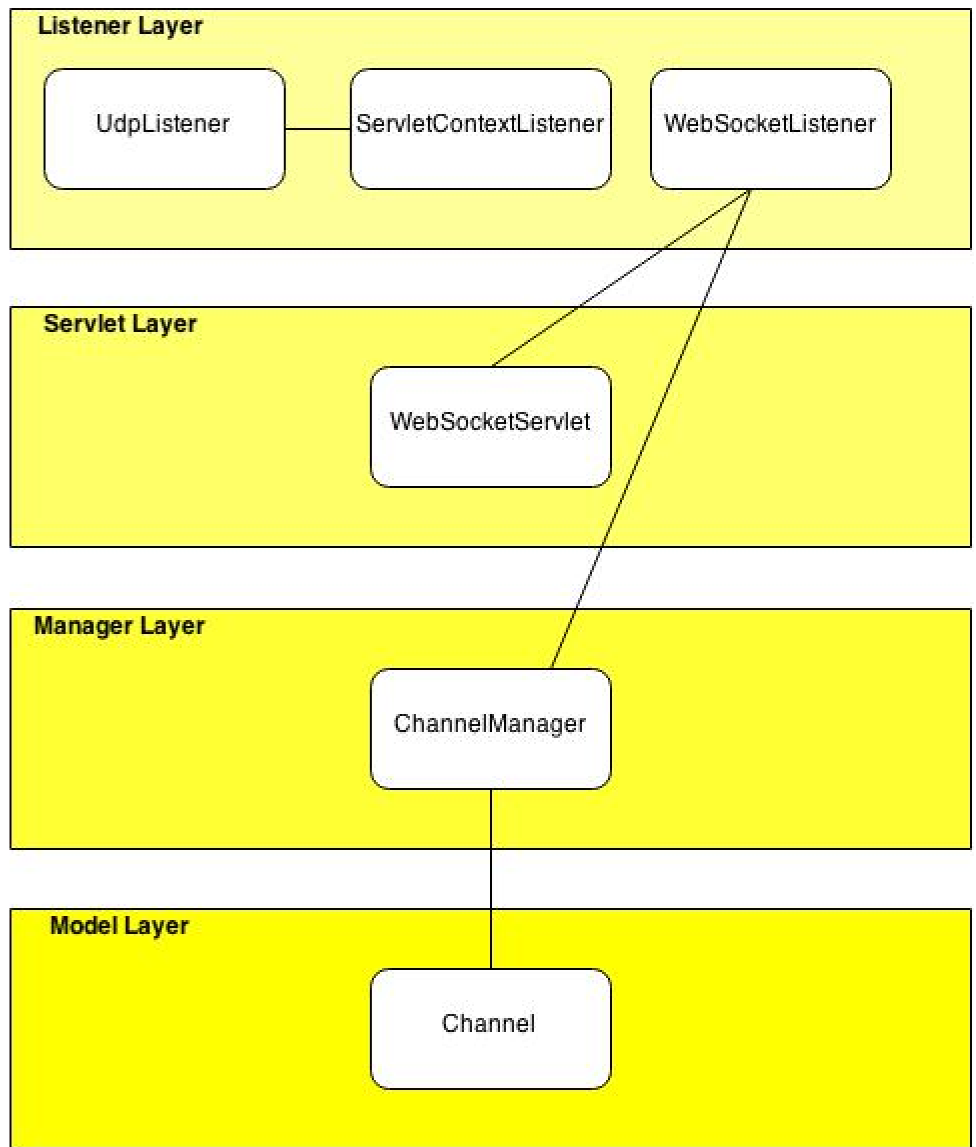

Figura 5.9: Arquitectura de la aplicación servidor
Diagrama de las distintas capas del servidor y la relación entre ellas.

5.10.2. Cliente

Esta dividido en 4 capas:

- Presentación: Aquí se encuentran las clases encargadas de actualizar la parte visual de la aplicación.
- Broadcast: Aquí se encuentran las clases que manejan los mensajes internos de la aplicación.

- Task: Aquí se encuentran las tareas asincrónicas de la aplicación. Como la necesaria para encontrar la dirección del server o para enviar las información producida al canal correspondiente.
- Listener: Aquí se agrupan los listeners de la aplicación en este caso esta el necesario para los mensajes de webSockets..

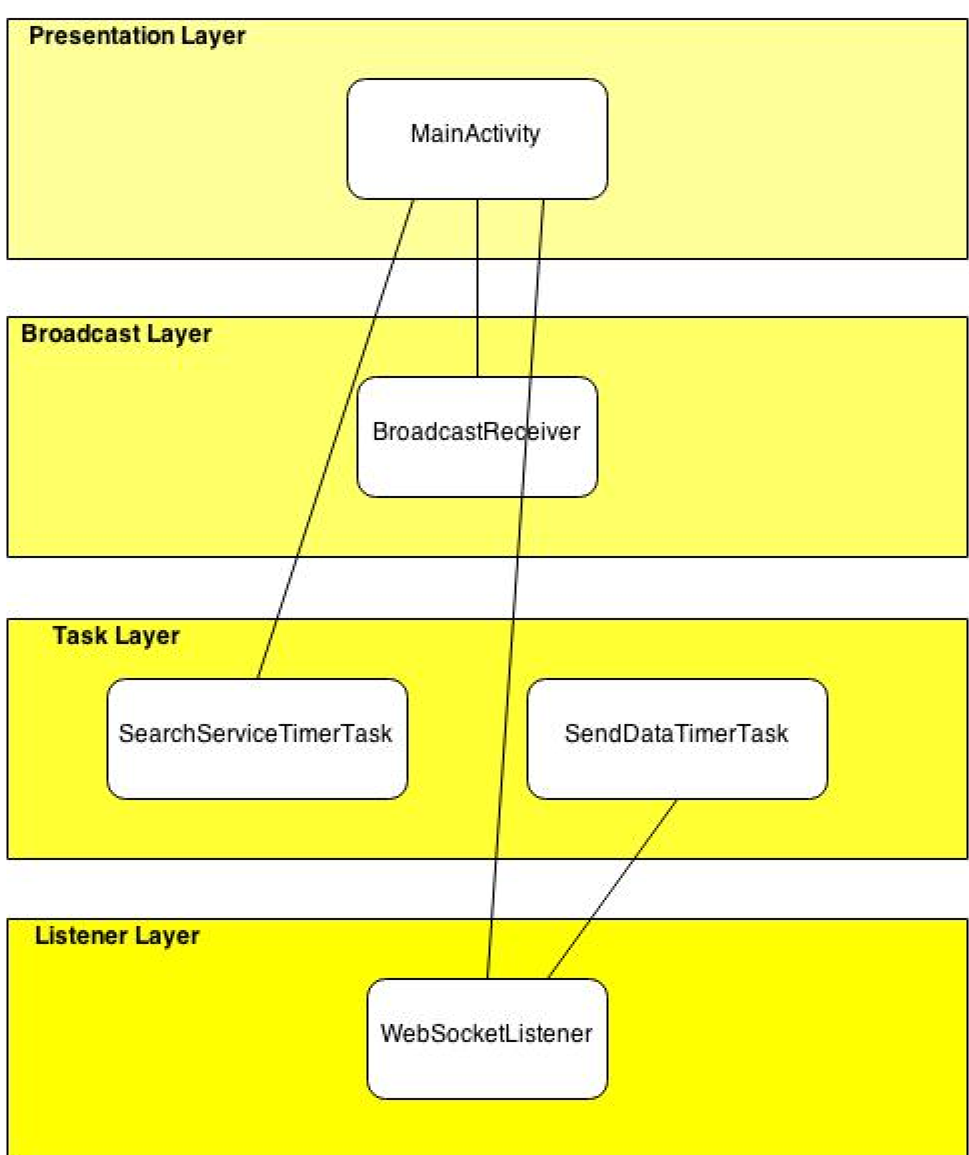

Figura 5.10: Arquitectura de la aplicación cliente
Diagrama de las distintas capas del cliente y la relación entre ellas.

5.11. Comparación con JWebSockets

Si bien la aplicación de JWebSockets parece ser bastante completa y robusta no cumpliría con requerimientos necesarios para trabajar con internet de las cosas de manera fluida y directa.

Los puntos más significativos son:

- No ofrece un mecanismo para el descubrimiento del hub en la red, por el contrario siempre asume un hub único o con igual dirección, si se piensa que los objetos inteligentes pueden ir variando su ubicación y con esto cambiando la red a la que se conectan se hace imprescindible ir redescubriendo la dirección del hub que servirá esa red.

- No ofrece una creación de canales bajo demanda de los objetos, por el contrario se crean bajo archivo de configuración o creación explícita en el servidor.

- No ofrece la posibilidad de taggear los canales según los intereses de los dispositivos, de hecho no provee ningún tipo de taggeo de los canales, la forma de identificación de los mismos es a través del nombre asignado.

- Al no ofrecer tags en los canales, deja afuera la posibilidad de utilizar algún sistema un poco más inteligente o complejo para utilizar los canales de forma más natural al tener topics que los identifiquen según el tipo de información que transmiten.

	jWebSockets	EIoT
Descubrimiento de hub	NO	SI
Creación dinámica de canales de comunicación	NO	SI
Canales tagueados	NO	SI
Uso de diccionario para sinónimos de tags	NO	SI

Figura 5.11: Tabla comparativa entre JWebSockets y EIoT
Tabla comparativa sobre las diferencias más importantes entre JWebSockets y EIoT

5.12. Casos de uso

La aplicación funciona exitosamente acorde a los parámetros enviados para la creación de canales, pero como esto depende de si el diccionario de wordNet contiene la relación entre las diferentes palabra utilizadas para dicho canal. Veremos casos de éxito y casos fallidos según los parámetros utilizados para los canales.

5.12.1 Caso de éxito

Los casos de éxito se producirán cuando tengamos un servidor corriendo y atendiendo las conexiones y los mensajes correspondientes a los clientes.

Para este ejemplo de caso de éxito utilizamos dos dispositivos móviles cada uno con la aplicación cliente instalada y con los tags de canales “cold” y “heat” respectivamente. El primer dispositivo inicia sin conexión e intenta encontrar el server de webSockets para iniciar la comunicación.

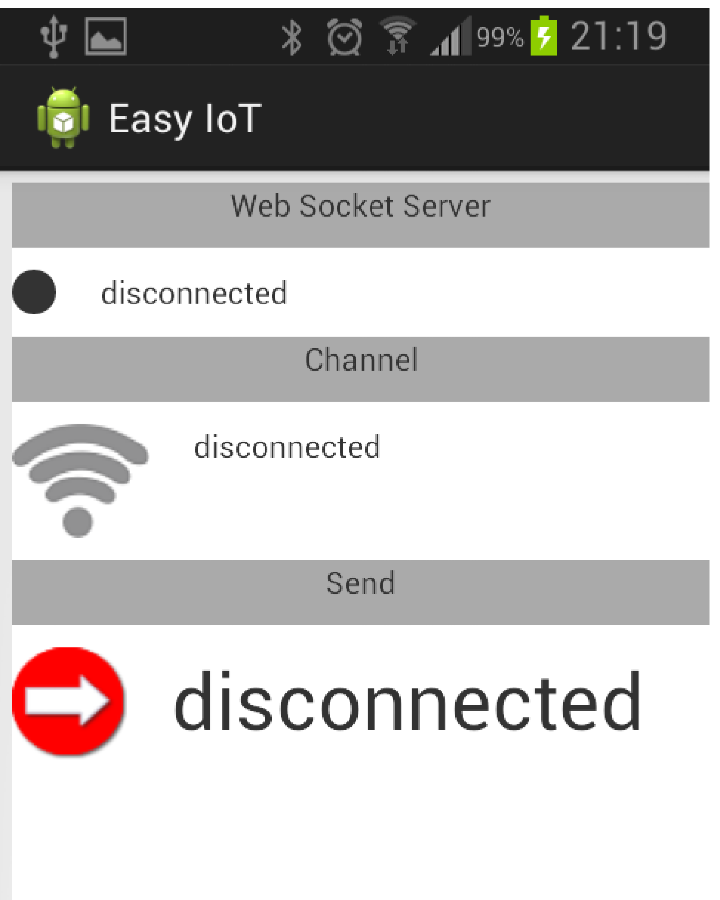

Figura 5.12: Aplicación cliente desconectada.
Captura de pantalla de la aplicación cliente antes de conectarse con el servidor.

Una vez que la conexión está establecida inicia el envío de datos por el canal correspondiente. En este caso el diccionario de wordNet asignó también al tag original: “hot” y el sinónimo: “heat energy”, por lo tanto ahora el canal estará taggeado con: “heat” y “heat energy”.

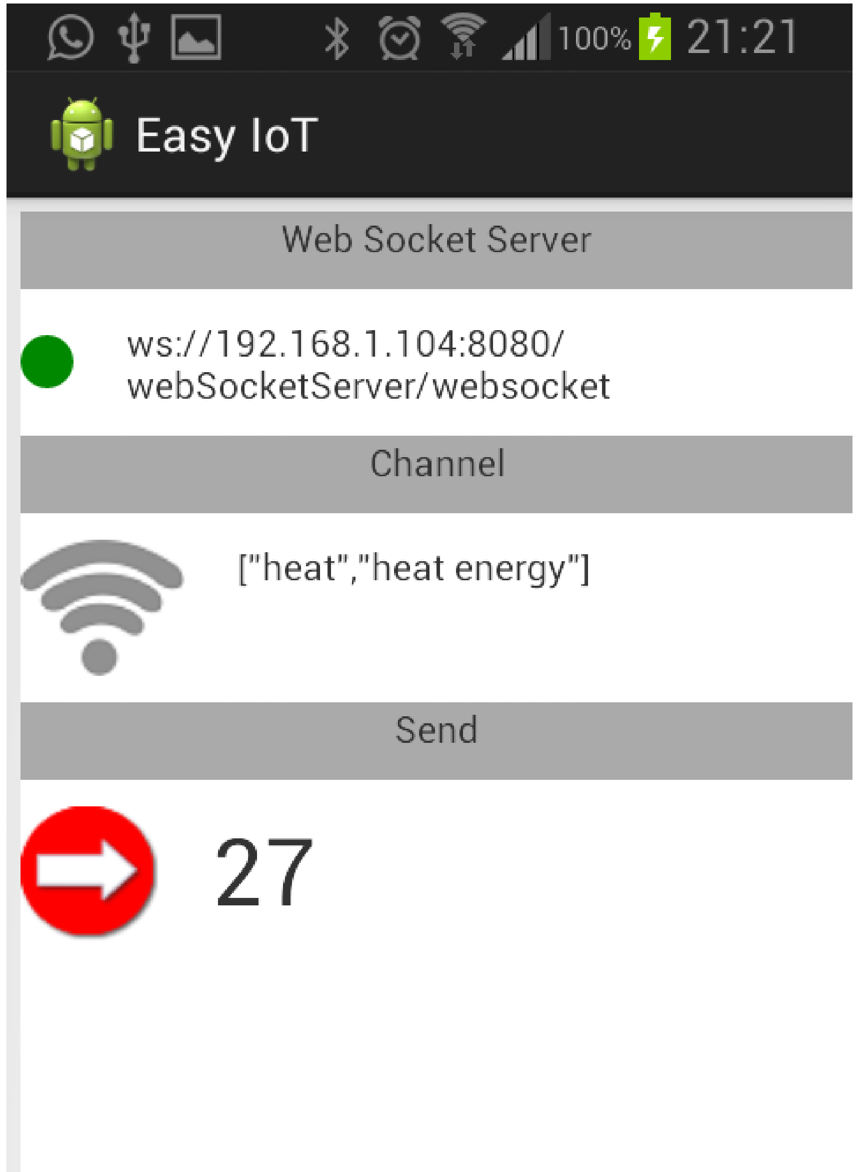

Figura 5.13: Aplicación cliente conectada y enviando datos.
Captura de pantalla de la aplicación cliente antes de conectarse con el servidor.

El el servidor que inicialmente está sin canales al recibir la conexión del cliente, con la información para crear el canal con el tag “heat energy” contrasta con el diccionario de wordNet los sinónimos y crea el nuevo canal con los tags “heat” y “heat energy” y en la aplicación dashboard se puede ver la información que circula por el canal.

Figura 5.14: Aplicación dashboard conectada.
Captura de pantalla de la aplicación dashboard monitoreando la actividad de los canales.

Si un segundo cliente inicia una conexión con el servidor y utiliza para crear el canal los tags "heat" o "heat energy" se le asignará el mismo canal y no se creará uno nuevo asumiendo que los objetos están interesados en compartir información del mismo tipo.

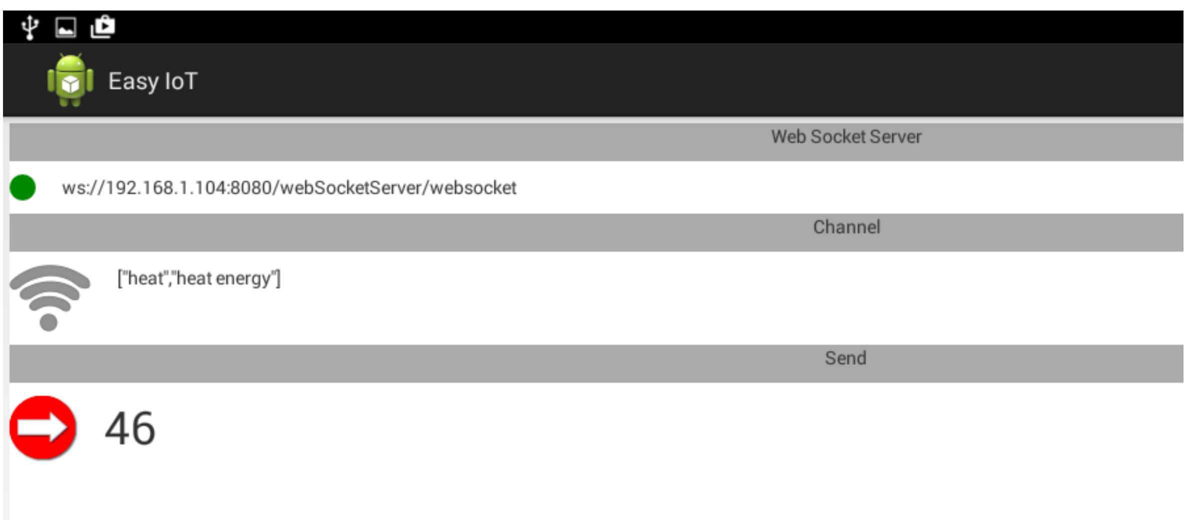

Figura 5.15: Aplicación cliente conectada.
Captura de pantalla de la aplicación cliente utilizando el mismo canal existente acorde al tag enviado.

5.12.2. Caso de fallo

El caso de fallo más trivial es no tener un servidor en la Lan que atienda los pedidos y conexiones de los clientes.

Pero el caso no trivial y más confuso se da a la par de una de las ventajas de la aplicación y es cuando el diccionario de wordNet unifica tags y provoca que el servidor no cree un nuevo canal de comunicación y se unifiquen los mensajes de los clientes productores.

O de manera análoga a la anterior wordNet no encuentra que dos palabras son sinónimas y provoca que el servidor cree dos canales que deberían ser uno solo.

Para este ejemplo de caso de fallo utilizamos dos dispositivos móviles cada uno con la aplicación cliente instalada y con los tags de canales “core” y “heart” respectivamente. El primer dispositivo inicia la conexión y con el server y crea el canal con el tag “core” a lo que el server utilizando wordNet le asigna además los tags: “core”, “nucleus”, “core group”, "kernel", "substance", "center", "essence", "gist", "heart", "heart and soul", "inwardness", "marrow", "meat", "nub", "pith", "sum", "nitty-gritty", "Congress of Racial Equality", "CORE", "effect", "essence", "burden", "gist".

Ahora el canal creado estará taggeado con tantos tags que será difícil poder saber si un objeto que quiere transmitir por un canal con un tag “heart” tiene la misma connotación que el anterior. Si tiene la misma connotación semántica no hay problema, pero si la connotación es diferente entonces se creará un solo canal con significados diferentes generando problemas fundamentalmente en los receptores de la información.

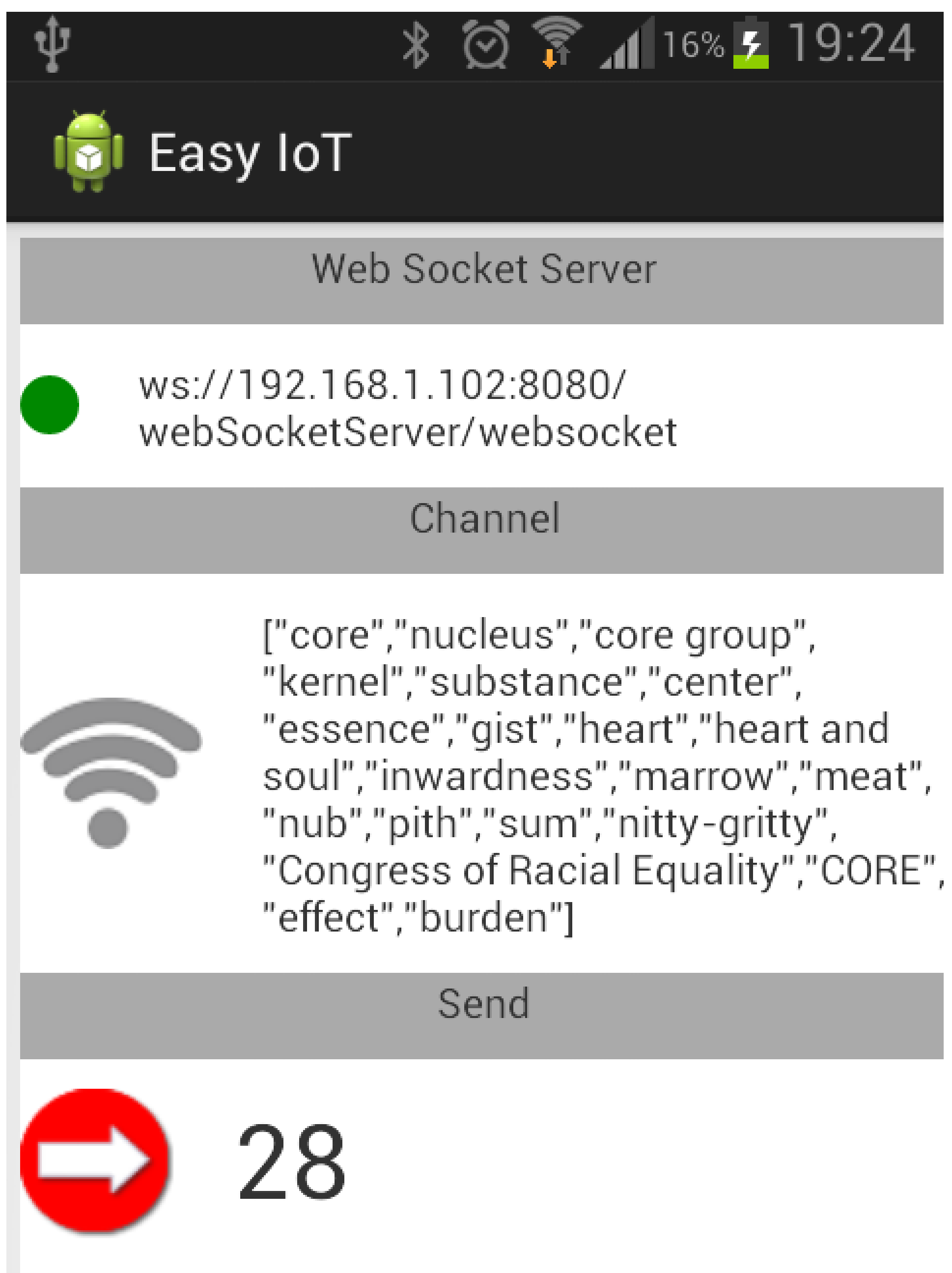

Figura 5.16: Aplicación cliente conectada con el tag “core”.
Captura de pantalla de la aplicación cliente utilizando el mismo canal creado bajo el tag “core”

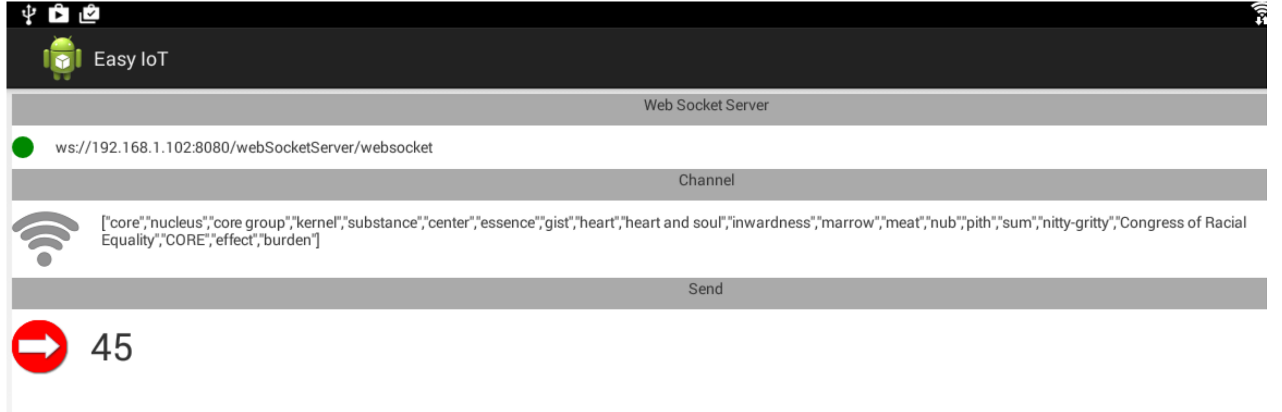

Figura 5.17: Aplicación cliente conectada con el tag “heart”.
Captura de pantalla de la aplicación cliente utilizando el mismo canal creado bajo el tag “heart”.

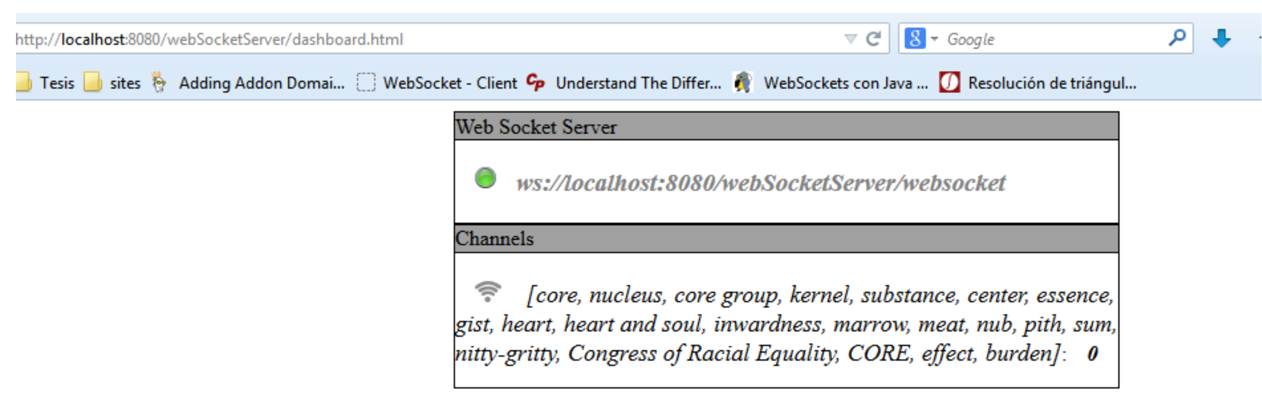

Figura 5.18: Aplicación dashboard inspeccionando los canales creados.
Captura de pantalla de la aplicación dashboard controlando el único canal creado para dos tags diferentes.

Capítulo 6: Trabajos futuros

Teniendo en cuenta las tecnologías utilizadas y lo nuevas que son, para futuros trabajos o ampliaciones del mismo, resultaría interesante evaluar la posibilidad de que mediante código javascript una aplicación web sea capaz al igual que lo es una de android de poder mediante descubrimiento de servidores, identificar el server de webSockets que atiende la red. Alguna aproximaciones se hicieron con aplicaciones chrome nativas pero aun no esta la tecnología necesaria para implementarlo en el resto de los navegadores.

También evaluando los puntos débiles del sistema resulta interesante abordar la creación de canales mediante tags, ya que si bien la colaboración de wordNet es de ayuda fundamental a la hora de tagear canales, también puede resultar en creaciones de canales con distinta connotación cosa que nos gustaría evitar, con lo cual un aporte interesante es ver la manera de cargar semánticamente la creación de canales para evitar o reducir al mínimo la posibilidad de colisiones no deseadas.

Si bien la aplicación cliente se realizó en Android sería interesante llevarla la implementación a dispositivos Arduinos, para poner en práctica el modelo de solución en dispositivos de menor capacidad, aproximando más aún los requerimientos de IoT

Conclusión

El objetivo de este trabajo de tesis era diseñar e implementar un mecanismo para la comunicación flexible entre dispositivos físicos, aplicaciones y servicios de software. Para complir este objetivo se demostro en forma empírica que una variación del esquema publish-subscribe, cumplía con lo necesario para llevar a cabo la comunicación deseada. La variación corresponde a un esquema publish-to-hub que ayuda a dispositivos con capacidades limitadas a llevar a cabo la comunicación entre ellos. Este trabajo está íntimamente ligado a la importancia que se le está dando hoy en día a Internet de las cosas, donde todos los objetos cotidianos tienden a estar comunicados unos con otros y necesitan de una infraestructura para llevar esto a cabo.

Bibliografía

- [Introduction IoT 2013] http://www.cisco.com/web/solutions/trends/iot/introduction_to_IoT_november.pdf
- [Gartner 2013] http://www.gartner.com/newsroom/id/2621015
- [Atzori 2010] The Internet of Things: A survey, Luigi Atzori, Antonio Lera, Giacomo Morabito
- [Santander ciudad inteligente 2013] http://blogthinkbig.com/santander-ciudad-inteligente/
- [IETF RFC768 2013] http://tools.ietf.org/html/rfc768
- [RFC2616 1999] http://tools.ietf.org/html/rfc2616
- [Wordnet 2014] http://wordnet.princeton.edu. Princeton University "About WordNet."
- [Fette 2011] https://tools.ietf.org/html/rfc6455
- [Arduino] http://www.arduino.cc/
- [W3 Websockets 2014] http://dev.w3.org/html5/websockets
- [Publish-subscribe 2014] http://en.wikipedia.org/wiki/Publish%E2%80%93subscribe_pattern
- [Glass 2014] https://www.google.com/glass/start
- [PubSubHubHub 2014] http://en.wikipedia.org/wiki/PubSubHubbub
- [Machine-To-Machine 2010] http://whatis.techtarget.com/definition/machine-to-machine-M2M
- [RFID 2014] http://es.wikipedia.org/wiki/RFID
- [Ashton 2009] http://www.rfidjournal.com/articles/view?4986
- [Raunio 2009] The Internet of things, Bjórn Raunio
- [Smart Objects 2014] http://www.springer.com/gp/book/9783319004907
- [Gartner 12-2013] http://www.gartner.com/newsroom/id/2636073
- [Dunkel 2008] https://www.sics.se/~adam/dunkels08ipso.pdf
- [Smart Cities 2015] http://www.smartcities.es/
- [M2M 2014] http://es.wikipedia.org/wiki/M2M
- [M2M-IoT 2014] http://www.chemicalprocessing.com/articles/2014/understand-the-difference-between-iot-and-m2m/
- [jWS 2014] https://jwebsocket.org/
- [W3 UDP 2015] http://www.w3.org/2012/sysapps/tcp-udp-sockets/
- [W3 AJAX] http://www.w3schools.com/ajax/
- [Comet - Server Push] http://es.wikipedia.org/wiki/Comet
- [Pub-Sub] http://en.wikipedia.org/wiki/Publish-subscribe_pattern
- [W3 Discovery API] http://www.w3.org/TR/discovery-api/
- [SSDP 2014] http://en.wikipedia.org/wiki/Simple_Service_Discovery_Protocol
- [UPnP 2014] http://www.upnp.org/
- [mDNS 2014] http://www.multicastdns.org/
- [DIAL 2015] http://www.dial-multiscreen.org/
- [Ajax 2014] http://www.w3schools.com/ajax/ajax_intro.asp
- [Long Polling 2014] http://www.pubnub.com/blog/http-long-polling/
- [WS 2014] https://www.websocket.org/
- [Tomcat 2015] https://tomcat.apache.org/tomcat-7.0-doc/web-socket-howto.html
- [Android 2014] https://www.android.com/
- [HTML5 2014] http://www.html5rocks.com/es/

- [DatagramPacket 2010] http://download.java.net/jdk7/archive/b123/docs/api/java/net/DatagramPacket.html
- [DatagramSocket 2010] http://download.java.net/jdk7/archive/b123/docs/api/java/net/DatagramSocket.html